HAW HAMBURG

Bachelorarbeit

Moritz Höwer

Effiziente GPU-basierte Klassifizierung von Fahrspuren auf eingebetteten Echtzeitsystemen

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Moritz Höwer

Effiziente GPU-basierte Klassifizierung von Fahrspuren auf eingebetteten Echtzeitsystemen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Franz-Josef Korf
Zweitgutachter: Prof. Dr. Wolfgang Fohl

Eingereicht am: 14. Juni 2019

Moritz Höwer

Thema der Arbeit

Effiziente GPU-basierte Klassifizierung von Fahrspuren auf eingebetteten Echtzeitsystemen

Stichworte

CUDA, Echtzeit, eingebettetes System, Fahrspurerkennung, GPU, LiDAR, Optimierung

Kurzzusammenfassung

In dieser Arbeit wird ein Algorithmus zur Erkennung und Klassifizierung von Fahrspurmarkierungen aus LiDAR-Daten entwickelt. Damit dieser für die Umsetzung eines Autobahnpiloten genutzt werden kann, muss er robust sein und außerdem in Echtzeit auf einer NVIDIA AGX Xavier Platform laufen. Um dies zu erreichen wird der Algorithmus auf einer GPU beschleunigt und optimiert. Die mittlere Laufzeit konnte dabei auf 7 ms reduziert werden, ohne Robustheit einzubüßen.
Es wird darauf geachtet, dass der Algorithmus leicht erweiterbar ist und schnell an veränderte Voraussetzungen angepasst werden kann.

Moritz Höwer

Title of Thesis

Efficient GPU based classification of lanes on embedded realtime systems

Keywords

CUDA, embedded system, GPU, lane-detection, LiDAR, optimization, realtime

Abstract

In this thesis an algorithm to detect and classify lane markings based on lidar data is developed. To be used for realizing a highway pilot application, it has to be robust and run in realtime on an NVIDIA AGX Xavier platform. To achieve this, the algorithm is accelerated and optimized on a GPU. The average runtime was reduced to 7 ms, without compromising on robustness.

Care is beeing taken that the algorithm can easily be extended and quickly be adapted to updated requirements.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

ACC Adaptive Cruise Control.

ADAS Fahrassistenzsysteme – *engl. advanced driver-assistance systems* –.

CPU Central Processing Unit.

gcc GNU Compiler Collection.

GPGPU General Purpose Computation on Graphics Processing Unit.

GPU Graphics Processing Unit.

KPI Key Performance Indicator.

LKA Lane Keeping Assist.

nvcc NVIDIA CUDA Compiler.

RISC reduced instruction set.

SIMD Single Instruction Multiple Data.

SM Streaming Multiprocessor.

SOC System-on-a-Chip.

1 Einleitung

Die häufigste Unfallursache im Straßenverkehr ist menschliches Versagen [23]. Zunehmend mehr Fahrzeuge werden daher mit Fahrassistenzsystemen – *engl. advanced driver assistance systems* – (ADAS) ausgestattet. Diese können durch permanente Wahrnehmung der Umgebung und schnellere Reaktionszeiten einige der typischen Fehler menschlicher Fahrer vermeiden und so für mehr Sicherheit sorgen. Ein Beispiel für ein ADAS ist Adaptive Cruise Control (ACC) – zu deutsch Tempomat mit Abstandsregelung. Dabei wird der Tempomat um eine Funktion zur Erkennung des vorausfahrenden Fahrzeugs erweitert und regelt nun nicht nur die Geschwindigkeit sondern auch den Abstand. Bremst das vorausfahrende Fahrzeug, drosselt ACC die Geschwindigkeit automatisch. Ein weiteres Beispiel ist Lane Keeping Assist (LKA) – zu deutsch Spurhalteassistent. Dieses System erkennt die Fahrspurmarkierungen im Umfeld des Fahrzeugs. Wenn diese überfahren werden ohne den Blinker einzuschalten, warnt das System den Fahrer oder greift selbst in die Steuerung ein.

Um sicher fahren zu können benötigen ADAS, oder, als weitere Entwicklung, komplett autonom fahrende Fahrzeuge, genaue Informationen über die Position des Fahrzeugs. Als globale Referenz dient üblicherweise eine satellitenbasierte Positionsbestimmung, wie beispielsweise GPS. Weitere Informationen über die Umgebung stehen in einer Karte zur Verfügung. Die Genauigkeit von GPS reicht jedoch nicht aus, um auf mehrspurigen Straßen bestimmen zu können, auf welcher Fahrspur sich das Fahrzeug befindet. Auch die Karte hilft nur bedingt, da nicht gesichert ist, dass diese noch aktuell ist und zum Beispiel die dort hinterlegten Fahrspuren so noch existieren. Zudem muss das Fahrzeug auf die aktuelle Situation reagieren können.
Ein menschlicher Fahrer ermittelt die dafür notwendigen Informationen über sein Sehvermögen. Ein autonomes Fahrzeug nutzt Sensoren wie Kameras oder LiDAR (engl. light detection and ranging), die gemeinsam die *Perception* (Wahrnehmung) bilden. Ein Teil dieser Wahrnehmung ist die Erkennung von Fahrspuren. Damit die Fahrspurdaten für die Steuerung des Fahrzeugs zur Verfügung stehen, müssen diese in Echtzeit im Auto

bestimmt werden. Die dafür notwendige Rechenleistung wird in dieser Arbeit durch den Einsatz einer Graphics Processing Unit (GPU) als Rechenbeschleuniger bereitgestellt.

Im Bereich der Fahrspurerkennung gibt es bereits verschiedene Ansätze, sowohl rein auf LiDAR- oder Kamera-Basis, als auch auf Basis einer Fusion beider Systeme. Ziel dieser Arbeit ist die Entwicklung eines Systems, welches nur auf der Basis von LiDAR Daten die Fahrspurmarkierungen um das Fahrzeug herum erkennt und als durchgezogen oder gestrichelt klassifiziert. Das System soll in Echtzeit auf einer Xavier-Plattform von NVIDIA lauffähig sein.
Die Arbeit wurde bei der Ibeo Automotive Systems GmbH erstellt. Sämtliche Daten stammen daher aus Messsystemen von Ibeo.

In Kapitel 2 werden einige Grundlagen erklärt, die im weiteren Verlauf als bekannt vorausgesetzt werden. Kapitel 3 erläutert die Ausgangssituation und geht dabei auf verwandte Arbeiten sowie die Anforderungen und Voraussetzungen für das in Kapitel 4 beschriebene Konzept ein. Dieses wird in Kapitel 5 zunächst mathematisch und in Kapitel 6 dann als Implementierung eines Programms umgesetzt. Kapitel 7 behandelt die Optimierung des Algorithmus zur Echtzeitfähigkeit. In Kapitel 8 werden Konfigurationen des Algorithmus sowie die Ausführung auf der Zielplattform evaluiert. Kapitel 9 bildet schließlich den Abschluss dieser Arbeit und enthält ein Fazit sowie einen Ausblick auf mögliche Weiterentwicklungen des Systems.

2 Grundlagen

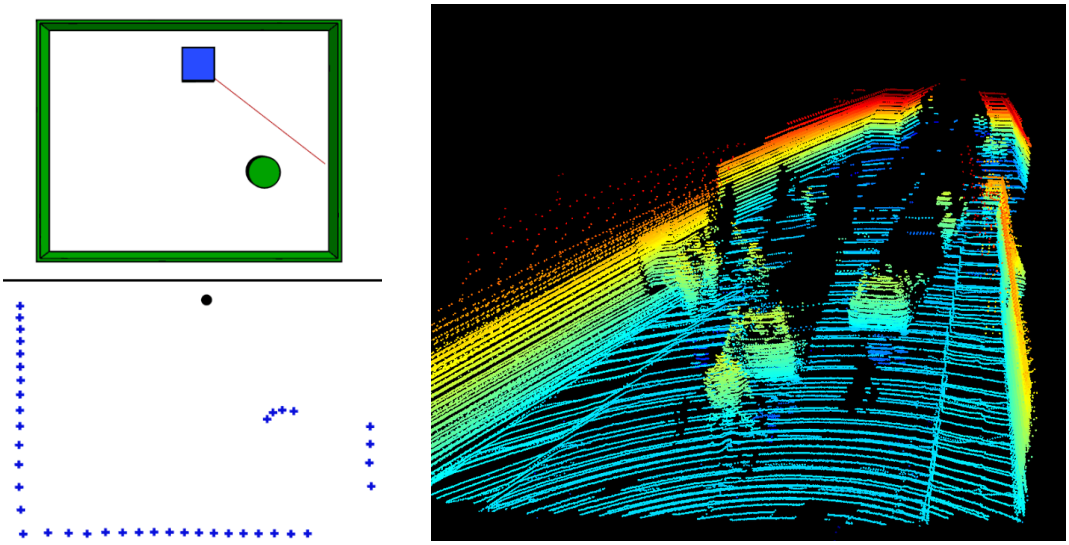
Im folgenden Kapitel werden grundlegende Begriffe und Konzepte eingeführt, auf denen im weiteren Verlauf aufgebaut wird. Abschnitt 2.1 behandelt die Funktionsweise eines LiDAR-Sensors. Abschnitt 2.2 geht darauf ein, wie man Fahrspurmarkierungen mit einem LiDAR erkennt. Abschnitt 2.3 erklärt die grundlegende Funktionsweise einer GPU sowie das zugehörige Programmiermodell.

2.1 Funktionsweise eines LiDAR

Ein LiDAR ist ein optischer Entfernungssensor. Er funktioniert nach dem *time-of-flight*-Prinzip, misst also die Laufzeit eines ausgesendeten Laserpulses bis zum Eintreffen von dessen Reflektion. Dies geschieht nicht nur einmal, sondern mehrmals für verschiedene Winkel, sodass als Ausgabe mehrere Entfernungswerte bereitstehen, die zusammen mit den jeweiligen Winkeln eine Punktwolke ergeben.

Bei einem 2D-LiDAR ergibt die Punktwolke einen Schnitt der Umgebung auf Höhe der Sende-/Empfangseinheit. In Abbildung 2.1a ist dies schematisch dargestellt. Im oberen Teil ist die Welt dargestellt, im unteren Teil der LiDAR-Scan. Der LiDAR (blaue Box oben, schwarzer Punkt unten) misst im Gegenuhrzeigersinn in festen Winkelabständen die Entfernung zum nächsten Objekt. Die dabei entstehenden Punkte ergeben die Box, in der sich der Sensor befindet. Da der LiDAR nicht durch Objekte hindurchsehen kann wirft der grüne Kreis einen Schatten und verdeckt die dahinterliegende Ecke der Box.

Das Prinzip ist bei einem 3D-LiDAR grundsätzlich dasselbe. Allerdings misst dieser zusätzlich noch schräg nach oben und unten in verschiedenen Ebenen. Dadurch ergibt sich ein dreidimensionales Bild. Ein Beispiel dafür ist in Abbildung 2.1b abgebildet.
Auf dem Boden sind dort die Kreisbögen zu sehen, in denen der LiDAR misst. Ebenfalls zu sehen sind die Schatten hinter den Autos.



(a) 2D-LiDAR (Quelle: [29])

(b) 3D-LiDAR (Quelle: [27])

Abbildung 2.1: Beispielausgabe LiDAR

2.2 Fahrbahnmarkierungen und LiDAR

Fahrbahnmarkierungen haben eine hohe Retroreflexion, um vor allem nachts eine gute Sichtbarkeit zu gewährleisten. Diese betrifft sowohl das sichtbare Licht, als auch das bei LiDAR-Sensoren häufig eingesetzte Infrarotlicht. Erzeugt wird diese Retroreflexion beispielsweise über kleine Glasperlen, die in die Fahrbahnmarkierung integriert sind [24]. Abbildung 2.2 zeigt eine solche Glasperle. Die Glasperle funktioniert ähnlich wie eine Linse und fokussiert die einfallenden Lichtstrahlen auf die Rückwand der Glasperle. Die Rückseite wirkt spiegelnd und reflektiert einen Großteil des Lichts wieder zurück in die Richtung, aus der es gekommen ist. Da dieser Effekt prinzipiell für alle Einfallswinkel funktioniert, sind die Markierungen immer gut zu sehen.

Der Asphalt oder Beton, der den Rest der Straße bildet, absorbiert viel Licht. Schaut man sich nun das Intensitätsbild des LiDARs bei einem Scan der Straße an, sieht man darin die Fahrbahnmarkierung deutlich hervorgehoben. Abbildung 2.3 zeigt ein Beispiel. Um das 2D-Intensitätsbild aus den dreidimensionalen Daten zu erzeugen, wurden die einzelnen Messungen wie bei einem Kamerabild als rechteckiges Gitter angeordnet. Jeder Pixel korrespondiert zu einer Messung und zeigt als Grauwert die gemessene Intensität.

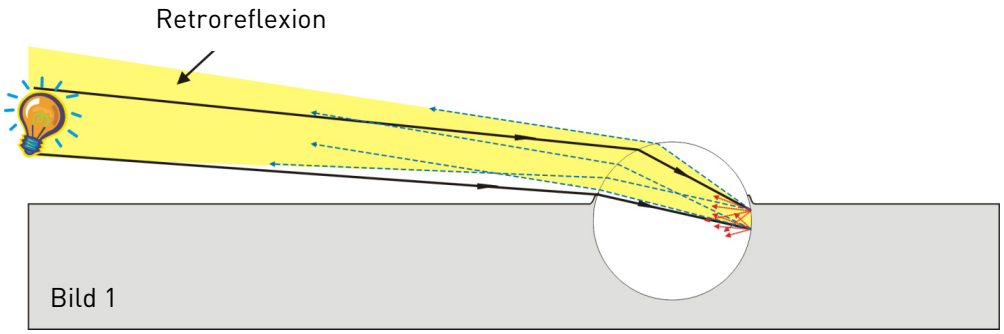


Abbildung 2.2: Glasperlen in der Fahrbahnmarkierung (Quelle: [24, S. 4])

Abbildung 2.3: Beispielhaftes Intensitätsbild eines LiDAR-Sensors

2.3 Funktionsweise einer GPU

Eine GPU wurde als Beschleuniger entwickelt, um aufwändige 3D-Grafiken generieren und anzeigen zu können. Mittlerweile werden GPUs jedoch immer mehr für *General Purpose Computation on Graphics Processing Unit (GPGPU)* eingesetzt. Dies bedeutet, dass die für das Erstellen von Grafiken nötige parallelisierte Architektur auch für nicht grafische Berechnungen genutzt werden kann. Ermöglicht wird dies durch Frameworks wie CUDA [19] oder OpenCL [22]

Da für die Erstellung dieser Bachelorarbeit auf NVIDIA GPUs gearbeitet wurde, wird der Aufbau anhand dieser erläutert. Die GPUs anderer Hersteller sind jedoch grundsätzlich ähnlich aufgebaut.

2.3.1 Architektur und Recheneinheiten einer GPU

Eine GPU ist nach dem Single Instruction Multiple Data (SIMD) Prinzip aufgebaut. Dies bedeutet, dass eine einzelne (Rechen-)Operation auf viele unterschiedliche Daten gleichzeitig angewendet wird.

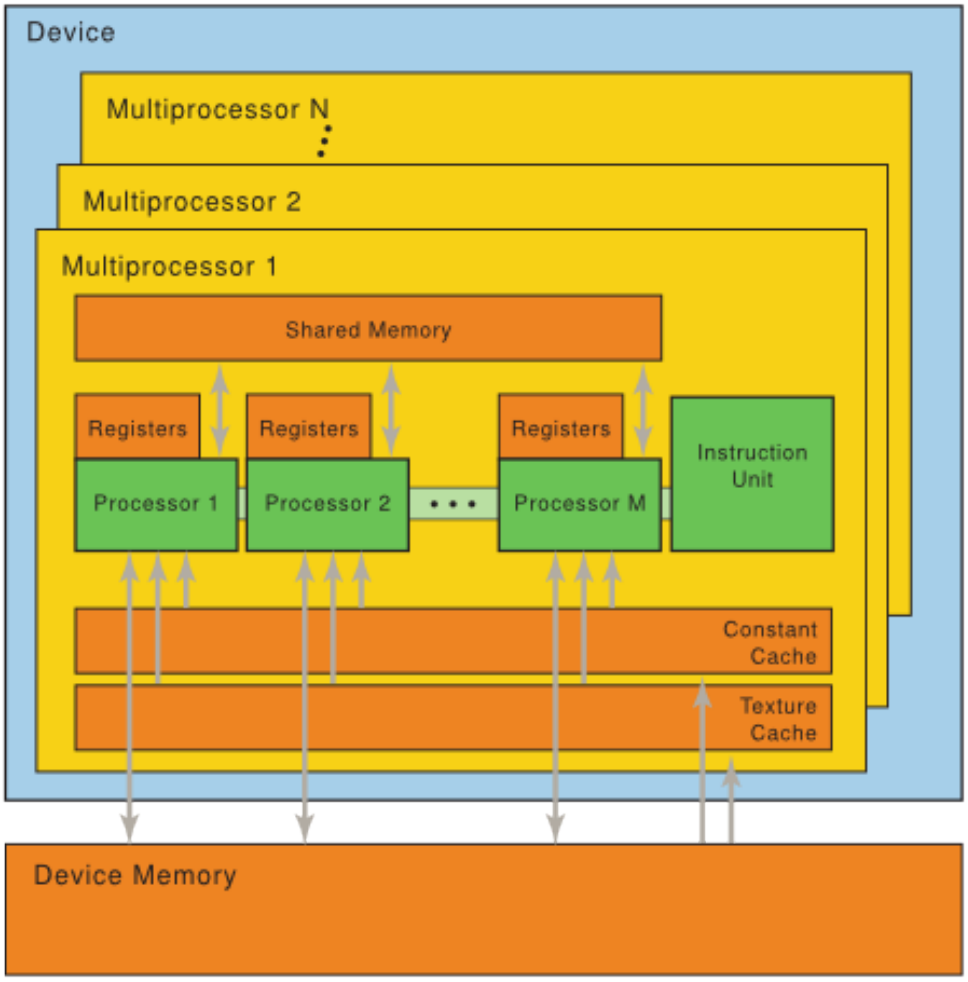


Abbildung 2.4: Hardwaremodell einer NVIDIA-GPU (Quelle: [25])

Die GPU ist dazu in mehrere Streaming Multiprocessors (SMs) aufgeteilt (vgl. Abbildung 2.4), welche jeweils Speicher, Caches und mehrere CUDA-Prozessoren enthalten. Letztere sind vergleichbar mit einer *reduced instruction set (RISC)*-Central Processing Unit (CPU), haben also nur einen kleinen Satz an Instruktionen, welche aber sehr effizient umgesetzt sind. Koordiniert werden diese Prozessoren von einer Instruction Unit.

Um die GPU auszulasten wird eine hohe Anzahl paralleler Operationen benötigt. Unterabschnitt 2.3.2 beschreibt, wie dies unter CUDA umgesetzt wird. Eine weitere Besonderheit bilden die verschiedenen Speicherklassen, welche für effiziente Implementierungen auf GPUs ausgenutzt werden müssen. Dies wird in Unterabschnitt 2.3.3 näher erklärt.

2.3.2 Das Threadmodell von CUDA

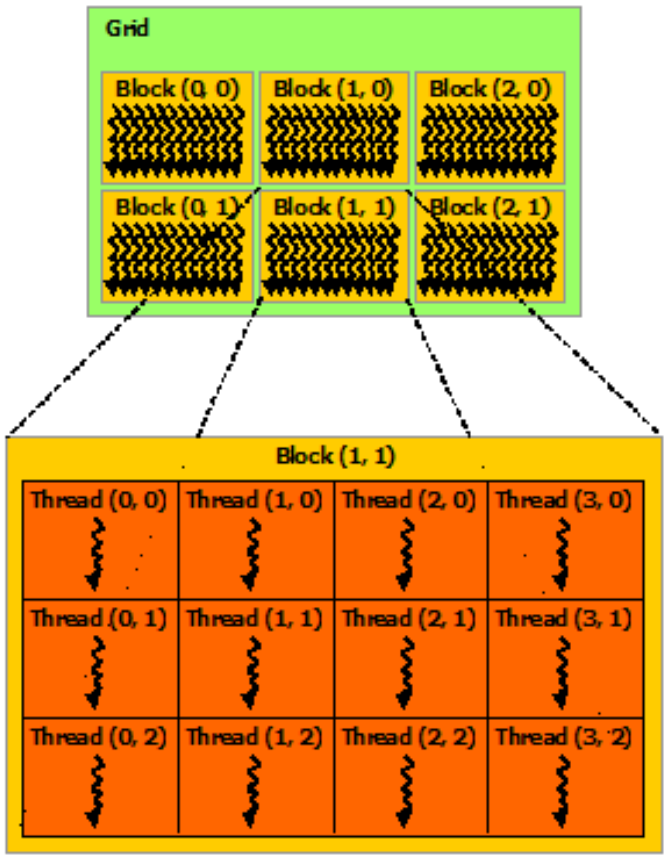


Abbildung 2.5: CUDA Threadmodell (Quelle: [26])

CUDA Programme werden auf der GPU von mehreren Threads ausgeführt. Das Threadmodell von CUDA (vgl. Abbildung 2.5) besteht aus drei Komponenten – den Threads, den Blocks und dem Grid. Darüber gibt es noch den Kernel, welcher hier aber nicht abgebildet ist. Der Kernel ist Teil des Programms und enthält den auszuführenden Programmcode. Zur Laufzeit wird dieser dann von mehreren Threads ausgeführt. Alle Threads führen dabei dieselben Instruktionen aus – sie haben jedoch eindeutige IDs,

welche im Programm als Variablen zur Verfügung stehen. Dadurch können die Parameter der Instruktionen (beispielsweise die Adresse eines Speicherzugriffs oder die Bedingung eines *if*) bei jedem Thread anders sein.
Threads werden in (Thread-)Blocks zusammengefasst, die Blöcke wiederum im Grid. Die Konfiguration dieses Grids wird dem Kernel beim Aufruf mitgegeben. CUDA ermöglicht bei der Konfiguration bis zu drei Dimensionen für die Thread- und Block-IDs, um die häufig auftretende Mehrdimensionalität der Operationen abbilden zu können und so die Programmierung zu vereinfachen.

Ausgeführt werden die Blöcke auf den SMs immer in Warps (Gruppen von 32 Threads), die nach dem SIMD-Prinzip alle gleichzeitig dieselbe Instruktion ausführen. Falls der Kontrollfluss innerhalb des Programms dafür sorgt, dass nicht alle 32 Threads dieselbe Instruktion ausführen (zum Beispiel durch unterschiedliche Zweige eines *if*, aufgrund der unterschiedlichen Parameter), kommt es zu *Divergence*. Dies bedeutet, dass alle 32 Threads erst die eine Instruktion ausführen und dann alle 32 Threads die andere. Die Instruction Unit sorgt in diesem Fall dafür, dass immer nur die jeweils gültigen Ergebnisse weiterverwendet werden, es verdoppelt sich aber die Laufzeit. Dieses Verhalten gilt es also zu vermeiden.
Bei der Ausführung verteilt ein Scheduler die Blöcke auf die SMs. Die Blöcke werden nebenläufig und unabhängig voneinander ausgeführt. Es kann also passieren, dass ein Block komplett fertig berechnet ist, bevor ein anderer gestartet wurde. Mit dem Aufruf `cudaDeviceSynchronize()` im CPU-Programmteil kann aber darauf gewartet werden, dass der Kernel (und damit alle enthaltenen Blöcke) beendet wurde.

2.3.3 Speicherklassen einer NVIDIA-GPU

Auf einer CPU gibt es nur eine Speicherklasse (Hauptspeicher). Daten werden aus diesem Hauptspeicher über einen hierarchischen Cache geladen. Die GPU hat hingegen verschiedene Speicherklassen. Diese sind in der Dokumentation von NVIDIA [12] ausführlich beschrieben und werden hier daher nur kurz erläutert. Eine ungünstige Nutzung der Speicherklassen kann das Programm um mehrere Größenordnungen langsamer machen. Es ist daher wichtig, sich Gedanken zu machen, wo welche Daten abgelegt werden. Auch die Art, wie auf den Speicher zugegriffen wird, hat großen Einfluss auf die Geschwindigkeit des Programms.

Abbildung 2.4 zeigt die verschiedenen Speicher und die spezialisierten Caches.

Grundsätzlich gibt es auf der GPU einen großen globalen Speicher (Device Memory), der für alle zugreifbar ist. Dieser umfasst typischerweise mehrere Gigabyte, der Zugriff darauf ist jedoch vergleichsweise langsam.
Ein Teil des globalen Speichers steht als Constant Memory zur Verfügung und wird über den Constant Cache stark beschleunigt. Auf einen weiteren Teil des globalen Speichers kann über eine Texture Unit und den Texture Cache zugegriffen werden.
Weiterhin gibt es direkt auf dem Chip ein Shared Memory. Dieses ist deutlich schneller als der globale Speicher, allerdings nur einige hundert Kilobyte groß.

Die Speicher sind darauf ausgelegt von mehreren Threads gleichzeitig gelesen zu werden. Lesen beispielsweise alle Threads von derselben Adresse im globalen Speicher, erzeugt dies nur einen Speicherzugriff. Das Ergebnis steht dann gleichzeitig allen Threads zur Verfügung (*broadcast*). Wird der Wert aus dem Constant Memory gelesen, liegt dieser sogar direkt auf dem Chip im Cache vor.
Auf den globalen Speicher wird sonst in 32-, 64- oder 128-Byte Blöcken zugegriffen. Wenn beispielsweise jeder Thread eines Warps jeweils einen 16-Bit Integer liest, und diese hintereinander im Speicher liegen, erzeugt dies nur einen einzigen 64-Byte Speicherzugriff. (*memory-coalescing*). Liegen die Adressen nicht nebeneinander, müssen mehrere getrennte Speicherzugriffe durchgeführt werden – auf Kosten der Ausführungsgeschwindigkeit.

Das Shared Memory ist in Speicherbänke (sogenannte *banks*) aufgeteilt, auf die jeweils gleichzeitig in einem GPU-Takt zugegriffen werden kann. Wenn beispielsweise alle Threads eines Warps jeweils einen Wert lesen oder schreiben, und diese Werte alle in unterschiedlichen Speicherbänken sind, können alle Zugriffe gleichzeitig erfolgen. Liegen zwei Zugriffe in derselben Speicherbank, erzeugt dies einen *bank-conflict* und die Speicherzugriffe müssen nacheinander ausgeführt werden – auf Kosten der Ausführungsgeschwindigkeit.

Da bei der Bearbeitung von Bildern sehr häufig Operationen vorkommen, welche im Bild nebeneinander liegende Pixel bearbeiten, gibt es den Texture Cache. Dieser ist darauf ausgelegt, Zugriffsmuster zu beschleunigen, die auf Texture Elements (Texels) zugreifen, welche im Bild nah beieinander liegen. Abbildung 2.6 zeigt ein Beispiel für ein 2D-Bild – im zugrunde liegenden Speicher, welcher zeilen- oder spaltenweise angelegt ist, liegen die Texel weit auseinander.

Die mit dem Texture Cache verbundene Texture Unit bietet zudem noch weitere Operationen zur Bildverarbeitung an. Sie ermöglicht es zum Beispiel, zwischen den Texeln zu interpolieren – zwar mit niedriger Genauigkeit (die Gewichtungsfaktoren haben nur

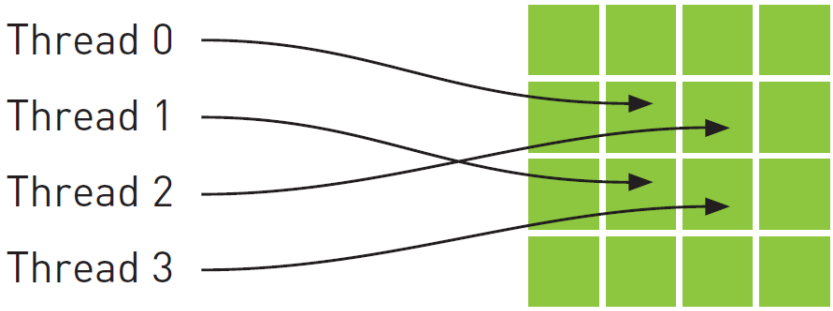


Abbildung 2.6: CUDA Texture Memory (Quelle: [28])

8bit), aber dafür in Hardware während des Speicherzugriffs.
Des Weiteren sind Zugriffe auf Texel außerhalb des Bildes möglich. In normalem Speicher wären diese undefiniert, die Texture Unit bietet dafür jedoch verschiedene Randbedingungen an. Beispielsweise kann bei Zugriffen außerhalb des Bildes immer der Wert 0 zurückgegeben werden.

Kapitel 7 beschreibt, wie Prinzipien aus diesem Abschnitt angewandt werden, um den Algorithmus auf der Grafikkarte zu beschleunigen.

3 Ausgangssituation

In diesem Kapitel werden zunächst verwandte Arbeiten vorgestellt. Danach werden die Anforderungen und Voraussetzungen erläutert, auf deren Basis im nächsten Kapitel das Konzept entwickelt wird.

3.1 Verwandte Arbeiten

Im Bereich der Fahrspurerkennung gibt es verschiedene Ansätze. Diese lassen sich bezüglich der verwendeten Sensorik in drei Arten einteilen: Kamera-basiert, LiDAR-basiert und basierend auf einer Fusion von Kamera und LiDAR.

Ein häufig als Basis genutzter Algorithmus ist die Hough-Transformation [8]. Diese ist ein globales Verfahren, um Geraden (in der generalisierten Variante auch andere Geometrische Modelle) in Bildern zu finden.

Bei den Kamera-basierten Verfahren [10, 17] werden üblicherweise zuerst die Kanten über eine Filterung ermittelt und dann mithilfe der Hough-Transformation Linien extrahiert, welche dann die erkannten Markierungen darstellen. Nachteil ist hier, dass die Position eines Pixels im Kamerabild in der realen 3D-Welt nicht genau bestimmt werden kann. Zudem ist das System passiv und daher abhängig von Umweltfaktoren – insbesondere der Beleuchtungssituation.

Bei den LiDAR-basierten Verfahren [13, 18] wird aus den 3D-Daten zunächst eine 2D Projektion der Straße von oben erstellt. Nun wird die Intensität der Messpunkte betrachtet und als Bild interpretiert. Darauf wird dann eine Hough-Transformation angewandt, welche die Linien extrahiert. Im weiteren Verlauf werden die erkannten Markierungen mithilfe eines Kalman Filters verfolgt. Nachteil ist hier, dass die Farbe der Linien (gelb oder weiß) von einem LiDAR nicht unterschieden werden kann. Da es sich bei einem

LiDAR um ein aktives System handelt (der Sensor empfängt Laserpulse, die er vorher selbst ausgesendet hat), ist dieser unabhängig von der Beleuchtungssituation.

Um die Nachteile des jeweils anderen Verfahrens auszugleichen, gibt es Ansätze, die auf einer Fusion von Kamera und LiDAR basieren. Dabei wird die Farbe der Markierung aus den Kameradaten und die Position aus den LiDAR-Daten bestimmt. An welcher Stelle des Algorithmus die Fusion stattfindet ist unterschiedlich. In [16] findet diese vor der Fahrspurerkennung auf den Eingabedaten statt. Die Fahrspuren werden dann auf Basis der fusionierten Daten ermittelt. In [7] findet die Fusion nach der Fahrspurerkennung statt. Hier werden die Fahrspuren zunächst unabhängig auf Kamera Daten und LiDAR Daten extrahiert und die Ergebnisse dann über einen Kalman Filter fusioniert.

In Kapitel 4 dieser Arbeit wird ein LiDAR-basiertes Verfahren vorgestellt, welches nicht auf Basis der Hough-Transformation funktioniert, sondern auf einem Verfahren von Florian Homm et. al. [7].
Anders als die Hough-Transformation ist dieses kein globales Verfahren, sondern nutzt nummerische Optimierung. Dies bringt Vorteile im Bezug auf den Rechenaufwand und ermöglicht ein genaueres Ermitteln der Parameter, da es nicht auf eine feste, diskrete Aufteilung der Parameter setzt. Nachteilig ist jedoch, dass das Verfahren nicht immer die global besten Parameter findet, sondern ggf. in einem lokalen Maximum endet. Es kann, wie die generalisierten Hough-Transformation, die Parameter für beliebige geometrische Modelle ermitteln.

3.2 Anforderungen

Der Kunde möchte einen Autobahnpiloten entwickeln, um auf der Autobahn autonom fahren zu können. Die verwendeten Sensoren liefern Messungen mit 25Hz und sind an eine Plattform mit einem NVIDIA Xavier System-on-a-Chip (SOC) angebunden. Das System soll flexibel und erweiterbar sein. Als Ausgabe werden die Parameter des Straßenmodells sowie die erkannten Fahrspurmarkierungen benötigt. Diese sollen jeweils mit einer Konfidenz versehen sein, die angibt, wie sicher das Ergebnis ist.
Folgende funktionale (F) und nicht-funktionale (NF) Anforderungen sind von dem System zu erfüllen:

ID	Bezeichnung	Anforderung
F1	Einsatzgebiet	Die Fahrspurerkennung soll für autonome Fahrten auf der Autobahn eingesetzt werden. (Autobahnpilot)
F2	Ausgabeformat	Die Fahrspurerkennung muss die Parameter des Straßenmodells sowie Position und Typ der erkannten Fahrspurmarkierungen ausgeben.
F3	Konfidenzen	Die Fahrspurerkennung muss ihre Ausgabe (Anforderung F2) mit Konfidenzen versehen.
NF1	Laufzeit	Die Fahrspurerkennung muss während der Fahrt im Auto möglich sein. *(online)*. Die Sensoren liefern mit 25Hz (alle 40ms) neue Messungen – die Laufzeit der Fahrspurerkennung muss daher im Mittel unter 40ms bleiben.
NF2	Austauschbarkeit Straßenmodell	Das verwendete Straßenmodell soll beim Kompilieren des Programms leicht austauschbar sein.
NF3	Austauschbarkeit Berechnungsplattform	Die zu benutzende Berechnungsplattform (CPU / GPU) soll beim Kompilieren des Programms leicht austauschbar sein.
NF4	Zielplattform	Der Algorithmus soll auf der Xavier-Plattform von NVIDIA laufen.

Tabelle 3.1: Anforderungsliste

3.3 Daten und Vorverarbeitung

Der in Kapitel 4 vorgestellte Algorithmus arbeitet auf zweidimensionalen Projektionsbildern aus der Vogelperspektive. Diese werden aus einer Karte erstellt, in deren Mittelpunkt sich das Auto befindet. Die Karte zeigt einen Ausschnitt der Welt im Umfeld des Autos und bewegt sich mit diesem. Neue Messungen werden dabei kontinuierlich an ihrer entsprechenden Position in der Welt eingetragen. Alte Messungen, die nicht mehr im Umfeld des Autos liegen, weil dieses sich weiterbewegt hat, werden gelöscht.

Eine Messung beginnt mit dem Eintreffen neuer Rohdaten des Sensors. Die Rohdaten werden zunächst von einem Sensormodell verarbeitet. Dieses enthält Informationen über die physikalischen Zusammenhänge im Sensor (beispielsweise Linsenverzerrung des Objektivs) und kann diese korrigieren. Die Ausgabe davon ist eine dreidimensionale Punktwolke.

Aus dieser Punktwolke sollen im nächsten Schritt die Punkte extrahiert werden, die zu einer Fahrspurmarkierung gehören. Würde man die Punkte ungefiltert in die Karte eintragen, enthielte diese (und das daraus generierte Projektionsbild) neben ungewolltem Rauschen auch Spuren von dynamischen Objekten. Diese Spuren entstehen, wenn Objekte sich zwischen den Messungen bewegen, da diese dann mehrfach an verschiedenen Stellen eingetragen werden. Abbildung 3.1 zeigt ein Beispiel.

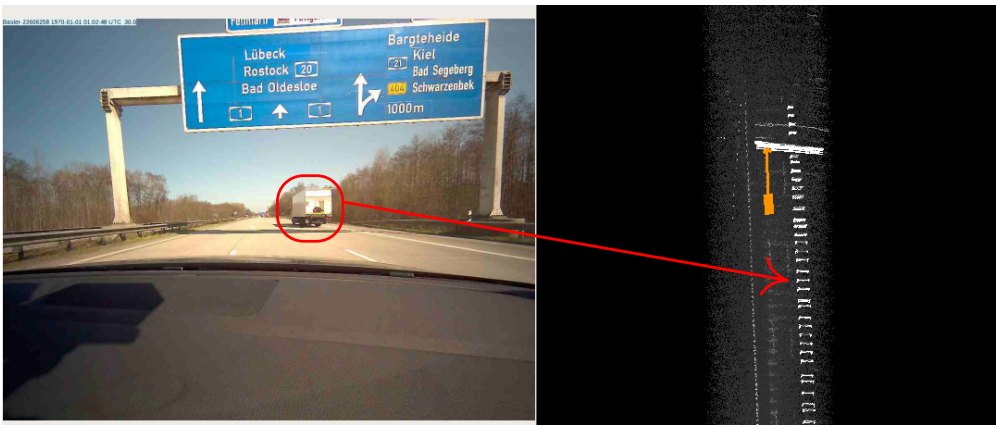


Abbildung 3.1: Spur eines LKWs

Da der vorgestellte Algorithmus versuchen würde, diese Spuren als Fahrspurmarkierung zu erkennen, wird die Punktwolke gefiltert. Der Filter ordnet jedem Punkt zu, wie wahrscheinlich dieser die Reflektion einer Fahrspurmarkierung darstellt [4]. Dazu betrachtet er das Intensitätsbild (vgl. Abbildung 2.3) des Sensors zeilenweise und sucht nach zusammenhängenden, hohen Intensitäten, deren Breite der Breite einer Fahrspurmarkierung entspricht. Denkbar wären hier auch andere Filteransätze, dies ist jedoch nicht Gegenstand dieser Arbeit.

In die Karte werden dann die Wahrscheinlichkeiten eingetragen, und im Projektionsbild als Grauwert dargestellt. Ein Beispiel für ein solches Projektionsbild zeigt Abbildung 3.2. Zur besseren Veranschaulichung wurden die Farben invertiert.
Das Auto bewegte sich zum Zeitpunkt der Erstellung des Projektionsbildes nach oben. Daher werden die Fahrspurmarkierungen nach oben schwächer – die Punkte dort sind noch neu und weit weg. Die untere Hälfte des Bildes befindet sich zwar nicht mehr im Blickfeld des Sensors, die Daten bleiben aber zunächst in der Karte, bis das Auto sich soweit fortbewegt hat, dass sie am unteren Rand des Projektionsbildes abgeschnitten werden.

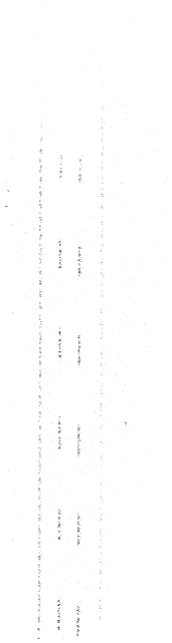


Abbildung 3.2: Projektionsbild (invertiert)

4 Konzept

In diesem Kapitel wird das Konzept für die Erkennung der Fahrspuren entwickelt.

4.1 Grundidee

Basierend auf einem Paper von Florian Homm et al. [7] wird ein Algorithmus vorgestellt, um die Fahrbahnmarkierungen aus Eingabedaten wie Abbildung 3.2 zu extrahieren.

4.1.1 Schritt 1 – Ermitteln der Straßenrichtung

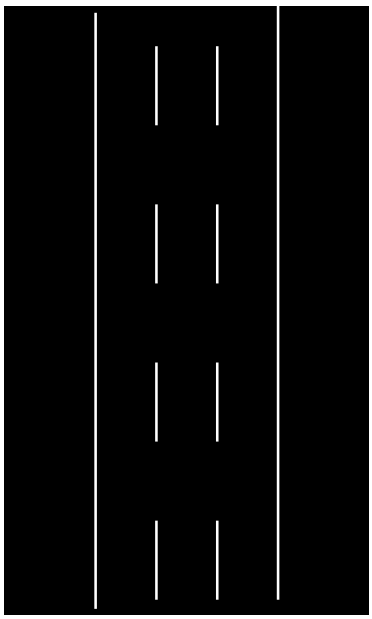


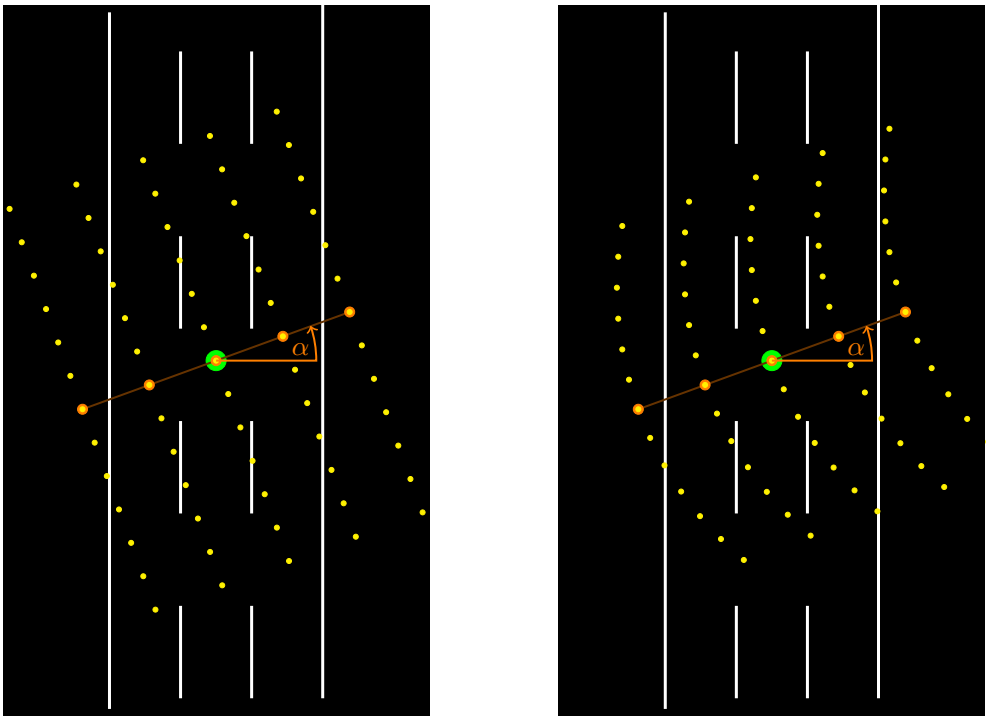
Abbildung 4.1: Ideales Bild

Grundannahme des Algorithmus ist, dass alle Fahrspuren parallel zueinander verlaufen. Für das geplante Einsatzgebiet Autobahnpilot (Anforderung F1) ist dies eine gültige Annahme. Einen Sonderfall bilden Auf- und Abfahrten sowie manche Baustellen. Diese werden in Abschnitt 4.3 näher betrachtet. Ausgehend von dieser Grundannahme wird zunächst ein parametrierbares Modell für den Verlauf der Straße definiert. Ziel des Algorithmus ist im ersten Schritt das Einstellen der Parameter dieses Modells.

Als Straßenmodell dient im einfachsten Fall eine Linie (Parameter: Winkel zum Fahrzeug (α)) – etwas flexibler ist ein Kreissegment (Parameter: Winkel zum Fahrzeug (α) und Krümmung (k)) oder ein Polynom (Parameter: Winkel zum Fahrzeug (α), Koeffizienten ($a_1 \dots a_n$)).

Das Modell einer Klothoide wird absichtlich nicht betrachtet. Zwar werden im Straßenbau häufig Klothoidenformen eingesetzt [1], diese sind jedoch mathematisch deutlich komplexer als die oben genannten Modelle. Für das Einsatzgebiet Autobahnpilot (Anforderung F1) reicht eine Approximierung über ein Polynom.

Zur Erklärung des Konzepts wird im Folgenden der Einfachheit halber ein Linienmodell gewählt. Im optimalen Fall sieht die Eingabe für den Algorithmus so aus wie Abbildung 4.1. Über dieses Bild wird ein Gitter gelegt, dessen Größe und Auflösung einstellbar sind. Anhand dieses Gitters wird das Bild dann abgetastet. Beispiele für solche Gitter sind in Abbildung 4.2 dargestellt. Der Parameter α beschreibt dabei die globale Rotation des Gitters. Das Straßenmodell bestimmt die Form der Kurven, die orthogonal zu einer Baseline sind und gleichmäßig entlang dieser Baseline verteilt werden. Abschnitt 5.1 beschreibt dies mathematisch.



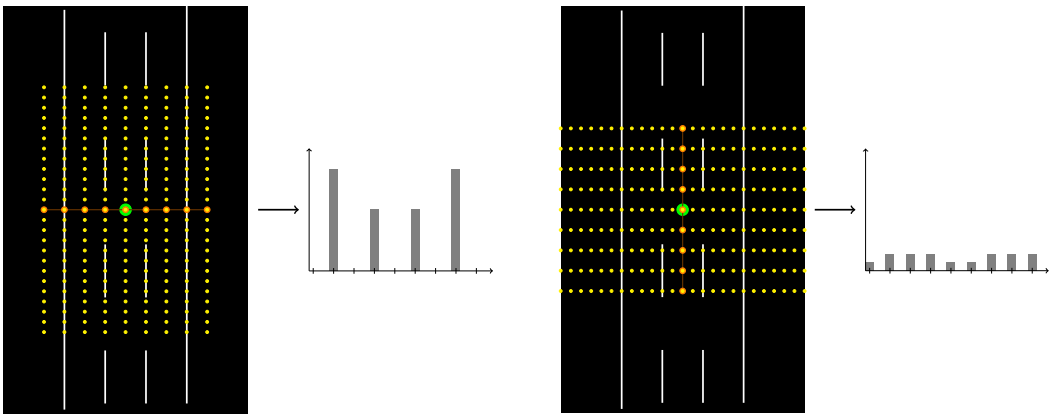
(a) Linienmodell (b) Kreismodell

Abbildung 4.2: Beispiele für verschiedene Gitter

Im nächsten Schritt wird ein Histogramm entlang der Baseline gebildet. Jede Kurve bildet dabei eine Klasse, deren Wert aus dem Bild berechnet wird. Dazu werden die Grauwerte der Pixel, die von Punkten auf der Kurve getroffen werden, aufsummiert.

Wenn ein Punkt auf der Kurve nicht genau einen Bildpixel trifft, wird der Grauwert über eine bilineare Interpolation [8] aus den benachbarten Pixeln ermittelt.

Wenn Straßenmodell und Parameter zu der echten Straße passen, enthält das Histogramm, wie in Abbildung 4.3a zu sehen, hauptsächlich leere Klassen – mit Ausnahme der Fahrbahnmarkierungen, die als Spitzen herausstechen. Passen die Parameter nicht, wie in Abbildung 4.3b, dann sind die Klassen im Histogramm ungefähr gleich verteilt.



(a) „Gute“ Parameter (b) „Schlechte“ Parameter

Abbildung 4.3: Vergleich der Histogramme bei „guten“ und „schlechten“ Parametern

Die Eigenschaft des Histogramms, dass die Werte sich auf wenige Klassen konzentrieren, kann über die Quadratsumme aller Klassen auf einen einzigen Wert abgebildet werden. Dessen Verlauf ist in Abbildung 4.4 beispielhaft für die Winkel von 0° bis 180° aufgetragen und wird maximal, wenn die korrekten Parameter gefunden wurden.

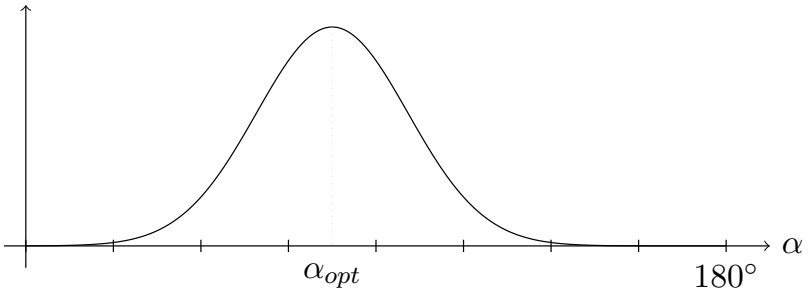


Abbildung 4.4: Verlauf der Quadratsumme für Winkel zwischen 0° und 180°

Da die Berechnung dieses Wertes eine rechenintensive Operation ist und es zudem unendlich viele mögliche Parameter gibt, kann das Optimum nicht durch vollständige Berechnung (*brute-force*) ermittelt werden. In dieser Arbeit wird das Maximum der Zielfunktion

(und damit die optimalen Parameter) über Verfahren der nummerischen Optimierung ermittelt.

Dabei gibt es verschiedene Ansätze, die sich zunächst grundlegend darin unterscheiden, ob nur die Funktionswerte betrachtet werden oder ob zusätzlich noch die Ableitung der Zielfunktion einbezogen wird [15].

Ein Vorteil ableitungsbasierten Verfahren sind mehr Informationen über die Zielfunktion. Dies führt in der Nähe des Optimums zu besserer Konvergenz auf den Zielwert als bei ableitungsfreien Verfahren [11]. Die Berechnung der Ableitung erfordert jedoch üblicherweise einen erhöhten Rechenaufwand, welcher den Vorteil bei der Konvergenz wieder aufheben kann. Zudem gibt es Funktionen, zu denen keine Ableitung berechnet werden kann. Dann können lediglich ableitungsfreie Verfahren eingesetzt werden.

Da die Zielfunktion differenzierbar ist, werden in dieser Arbeit zwei erprobte Verfahren aus der nummerischen Optimierung genutzt und verglichen. Das ableitungsfreie Downhill-Simplex-Verfahren [11] – nachfolgend nach den beiden Autoren als Nelder-Mead bezeichnet – und das ableitungsbasierte BFGS-Verfahren [11] (benannt nach den vier Autoren).
Damit geht diese Arbeit über das zugrunde liegenden Paper von Florian Homm et al. [7] hinaus, wo lediglich das Nelder-Mead-Verfahren eingesetzt wurde.

Um die Berechnung der Ableitung zu vereinfachen, kann die Zielfunktion aus mathematischer Sicht als eine Verkettung mehrerer Teilfunktionen gesehen werden. Zunächst werden die Parameter des Straßenmodells auf ein Abtastgitter abgebildet (g). Mit diesem wird dann das Bild abgetastet, wodurch es auf einen Vektor von Intensitätswerten abgebildet wird (ι). Dieser Vektor wird dann entlang der Kurven zu einem Histogramm summiert (s), welches schließlich über die Quadratsumme auf einen einzigen Wert abgebildet wird (r).

Mit

P – Anzahl der Parameter,

B – Anzahl der Histogrammklassen (Baseline),

H – Anzahl der Punkte pro Histogrammklasse.

ergeben sich folgende Abbildungen

$$\mathbb{R}^P \xrightarrow{g} \mathbb{R}^{2BH} \xrightarrow{\iota} \mathbb{R}^{BH} \xrightarrow{s} \mathbb{R}^B \xrightarrow{r} \mathbb{R} \tag{4.1}$$

welche zusammengenommen

$$\mathbb{R}^P \xrightarrow{h} \mathbb{R} \tag{4.2}$$

ergeben.

Abschnitt 5.3 beschreibt die Funktion im Detail mathematisch. Die Zerlegung vereinfacht zudem die in Anforderung NF2 geforderte Austauschbarkeit des Straßenmodells, da dieses nur die Teilfunktion g betrifft.

4.1.2 Schritt 2 – Ermitteln der Fahrspurmarkierungen

Nachdem die Straßenrichtung erfolgreich ermittelt und die korrekten Parameter gefunden wurden, ergibt sich ein Histogramm wie in Abbildung 4.5. Jede Klasse des Histogramms stellt eine Hypothese für eine mögliche Fahrspurmarkierung dar. Im nächsten Schritt müssen daraus die tatsächlichen Fahrspurmarkierungen extrahiert werden, welche sich dadurch auszeichnen, dass sie als Spitzen aus dem Histogramm herausragen.

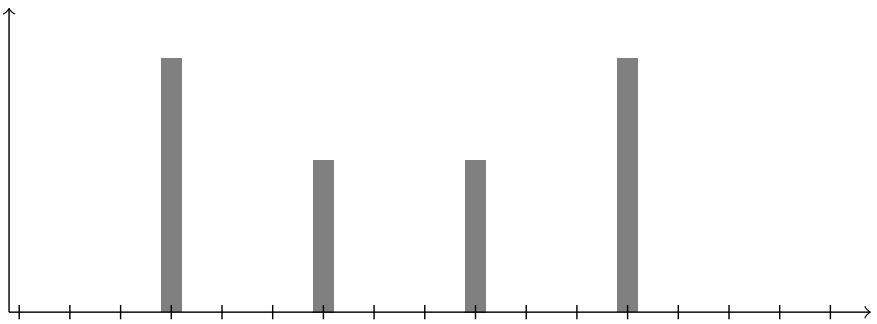
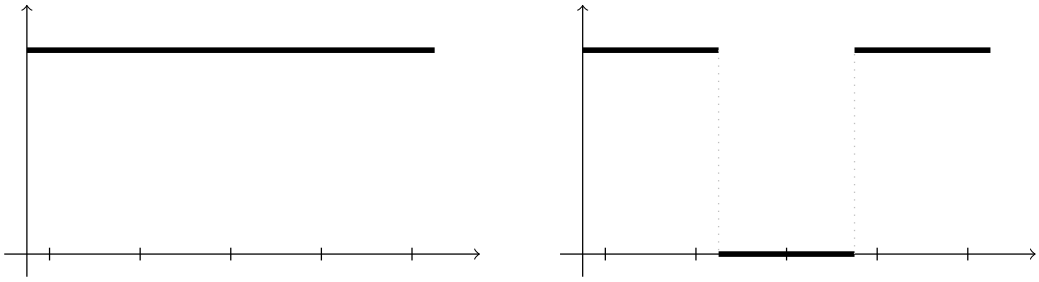


Abbildung 4.5: Ideales Histogramm

4.1.3 Schritt 3 – Klassifizierung der Fahrspurmarkierungen

Nachdem die Fahrspurmarkierungen ermittelt wurden, müssen diese als durchgezogen oder gestrichelt klassifiziert werden.

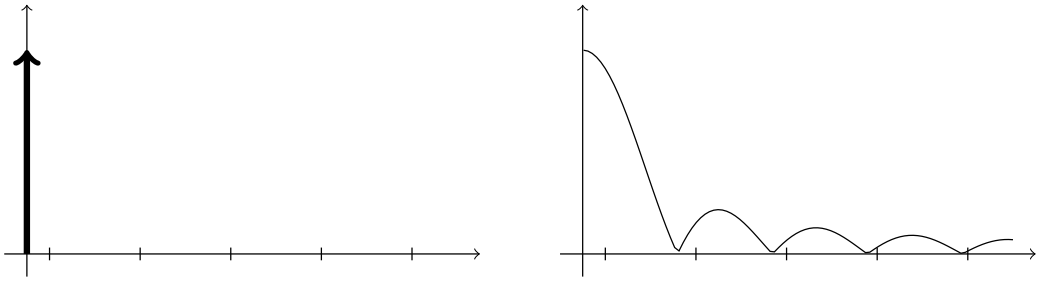
Jede der als Fahrspurmarkierung erkannten Klassen im Histogramm korrespondiert zu einer Kurve im Bild. Die Intensitäten entlang der Kurve, die in Summe den Wert der Klasse im Histogramm gebildet haben, werden nun als ein diskretes Signal entlang der Kurve betrachtet. Im Idealfall ergeben sich dann die in Abbildung 4.6 dargestellten Signale.



(a) Durchgezogen (b) Gestrichelt

Abbildung 4.6: Ideale Signale

Indem man eine Fouriertransformation anwendet, erhält man daraus die in Abbildung 4.7 dargestellten Signale im Frequenzraum. Beide Signale haben dabei einen konstanten Anteil bei Frequenz 0, das gestrichelte Signal enthält jedoch auch noch höhere Frequenzen. Die Verteilung dieser Frequenzen ist dabei abhängig von der Breite der Lücken.



(a) Durchgezogen (b) Gestrichelt

Abbildung 4.7: Ideale Fouriertransformierte

4.1.4 Schritt 4 – Ausgabe der Fahrspuren

Gemäß Anforderung F2 muss der Algorithmus die Parameter des Straßenmodells sowie die Position und den Typ der Markierungen ausgeben.

Die Parameter werden als Vektor ausgegeben, die Fahrspurmarkierungen als Liste. Dabei enthält jedes Element der Liste die Position als Index im Diskretisierungsgitter sowie den Typ der Markierung als Aufzählungstyp mit den möglichen Werten „gestrichelt“ oder „durchgezogen“.

4.2 Plausibilität

Nach der Ermittlung der Straßenrichtung (Unterabschnitt 4.1.1) wird diese mit der aktuellen Fahrtrichtung verglichen. Weicht die Straßenrichtung zu stark ab, ist dies ein Hinweis auf falsche Parameter – der Optimierer hat hier vermutlich ein lokales Nebenmaximum anstatt des globalen Maximums gefunden. In diesem Fall meldet der Algorithmus für den aktuellen Aufruf eine Konfidenz von 0 und setzt den Startwert für die Straßenrichtung im nächsten Aufruf auf die aktuelle Fahrtrichtung. Die Startwerte aller weiteren Parameter bleiben unverändert.

Nach der Ermittlung der Fahrspurmarkierungen (Unterabschnitt 4.1.2) wird bei den akzeptierten Hypothesen der Abstand der daraus resultierenden Fahrspurmarkierungen geprüft. Da in Baustellen die minimale Breite von Behelfsfahrstreifen 2,5 m beträgt [3, S. 113], wird die Hypothese mit der niedrigeren Konfidenz (Unterabschnitt 4.2.1) verworfen, wenn der Abstand zur nächsten Hypothese 2,3 m (2,5 m abzüglich Toleranz aufgrund der Auflösung) unterschreitet. Ein schmalerer Fahrstreifen wäre unplausibel.

Im Klassifikationsschritt wird nach Anwendung der Fouriertransformation das Spektrum im Frequenzraum F geprüft. Hier wird sowohl bei gestrichelten als auch insbesondere bei durchgezogenen Fahrspurmarkierungen ein signifikanter konstanter Anteil ($F(0)$) erwartet. Ist dieser nicht vorhanden, wurde die Hypothese vermutlich fälschlicherweise als Fahrspurmarkierung ermittelt. In diesem Fall wird die Konfidenz der Hypothese verringert und eine Klassifikation findet nicht statt.

4.2.1 Konfidenzen

Zusätzlich zu den Plausibilitätsprüfungen wird gemäß Anforderung F3 jede Fahrspurmarkierung mit zwei Konfidenzwerten versehen. Sie erhält zunächst eine *existence-confidence*, die beschreibt, wie sicher der Algorithmus ist, die Fahrspurmarkierung korrekt erkannt zu haben. Außerdem erhält sie eine *classification-confidence*, die beschreibt, wie sicher der Algorithmus ist, den Typ der Markierung korrekt erkannt zu haben.

Um die *existence-confidence* zu berechnen wird das arithmetische Mittel $\overline{x}$ sowie die Standardabweichung σ des Histogramms benötigt. Die *existence-confidence* e einer Hypothese mit dem Wert x_h berechnet sich als

$$e = \frac{x_h - \overline{x}}{\sigma}. \tag{4.3}$$

Die *classification-confidence* wird im Frequenzraum F berechnet. Benötigt werden dazu der konstante Anteil des Signals ($x_0 = F(0)$) sowie der Anteil der Wellenlänge der gestrichelten Linie (x). Die *classification-confidence* c berechnet sich als

$$c = \frac{x}{x_0}. \tag{4.4}$$

Die Weiterverarbeitung kann die Ausgabe der Fahrspurerkennung auf Basis dieser Konfidenzwerte mit Information aus anderen Modulen kombinieren, um daraus ein Gesamtbild der Situation zu erzeugen.

4.3 Grenzen

Aus der Grundannahme, dass alle Fahrspuren parallel verlaufen (vgl. Unterabschnitt 4.1.1), folgt direkt die Einschränkung, dass der Algorithmus nicht für Straßenkreuzungen oder Kreisverkehre funktioniert. Auch Gabelungen, die z.B. bei Auf- und Abfahrten oder Baustellen auftreten, können nicht dargestellt werden. Aufgrund der Beschaffenheit der Daten, deren Intensität mit zunehmender Entfernung zum Fahrzeug abnimmt, ergibt sich für Parameter, die auf die aktuelle Spur passen, ein höherer Funktionswert als für Parameter, die auf die abzweigende Spur passen. Letztere wird zudem häufig teilweise

verdeckt (z.B. durch Leitplanken oder Betonwände), was den Funktionswert für die Parameter der abzweigenden Spur weiter verringert. Der Algorithmus folgt daher bevorzugt der aktuellen Spur des Fahrzeugs.

Es werden keine Informationen aus vorherigen Ergebnissen verwendet (kein Tracking). Einzig die Startwerte für die Optimierung werden aus dem vorherigen Ergebnis übernommen, der Algorithmus selbst verwendet dann jedoch nur die Daten aus dem aktuellen Projektionsbild. Sind diese ungünstig, kann es passieren, dass eine Fahrspurmarkierung, welche zuvor eine hohe Konfidenz hatte, in diesem Durchlauf nicht mehr erkannt wird. Man könnte die Ausgabe des Algorithmus beispielsweise mithilfe eines Kalman Filters [2] filtern – das ist jedoch nicht Gegenstand dieser Arbeit.

Wenn Markierungen kurz verdeckt werden, beeinflusst dies den Algorithmus aufgrund der zugrundeliegenden Karte nicht. Bleiben sie aber länger verdeckt, weil beispielsweise gerade ein langer LKW überholt wird, können die Markierungen aus der Karte und damit auch aus der Ausgabe verschwinden. Ebenso können verrauschte Daten (aufgrund schlechter / fehlerhafter Vorverarbeitung) den Algorithmus – insbesondere den Klassifikationsschritt – beeinflussen.

Für das Einsatzgebiet Autobahnpilot (Anforderung F1) sind die beschriebenen Einschränkung jedoch akzeptabel. Einige der Einschränkungen können durch weitere Systeme (z.B. Offline-Karte, Dynamische Objekterkennung) kompensiert werden – dies ist jedoch nicht Gegenstand dieser Arbeit.

4.4 Erfüllung der nicht-funktionalen Anforderungen

Um die Laufzeitanforderung NF1 zu erfüllen – insbesondere auf der in NF4 geforderten Zielplattform – wird der Algorithmus teilweise auf einer GPU implementiert. Zudem wird der Optimierer bezüglich der Anzahl der Iterationen begrenzt. Dies ist in Kapitel 7 beschrieben.

Um die Anforderungen NF2 und NF3 zu erfüllen wird in der Architektur darauf geachtet, an geeigneter Stelle Abstraktionen vorzusehen. Diese werden in Abschnitt 6.1 erläutert.

Auf die Besonderheiten der Zielplattform (NF4) geht Abschnitt 8.3 ein.

5 Mathematisches Modell

In diesem Kapitel wird das mathematische Modell eingeführt, um das Konzept aus Kapitel 4 umzusetzen. Es bildet die Basis der Implementierung, welche in Kapitel 6 beschrieben wird.

5.1 Koordinatensysteme

Um den Algorithmus besser beschreiben zu können werden verschiedene Koordinatensystem eingeführt. Diese sind in Abbildung 5.1 dargestellt.

Basierend auf einem Weltkoordinatensystem (w) wird ein Koordinatensystem (b) für die Baseline definiert. Die Baseline kann über den Parameter α um den Ursprung von b rotiert werden. Gleichmäßig entlang der Baseline verteilt gibt es für jede Fahrspurmarkierungshypothese ein eigenes Koordinatensystem (c_i) sowie ein dazu verdrehtes Koordinatensystem c_i'.

Die Baseline ist mit einer Schrittweite s_b diskretisiert. Die Anzahl $2n_b + 1$ der Disktretisierungspunkte bestimmt somit auch die Anzahl der Koordinatensysteme c_i und $c_i', i = -n_b, \ldots, n_b$.

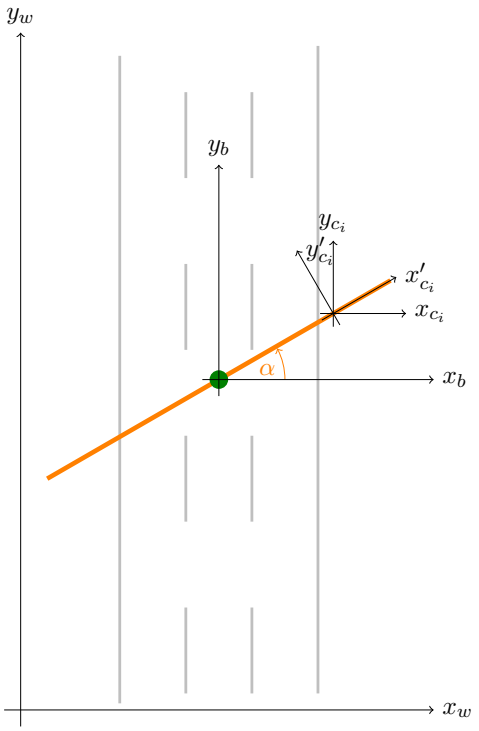
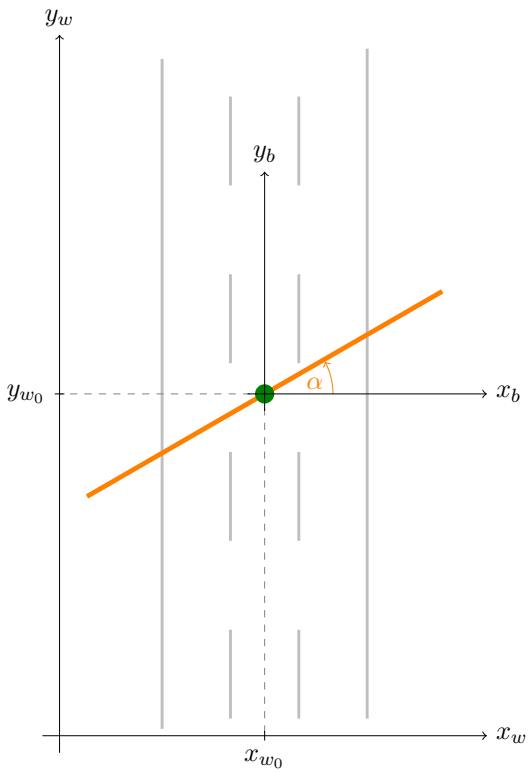
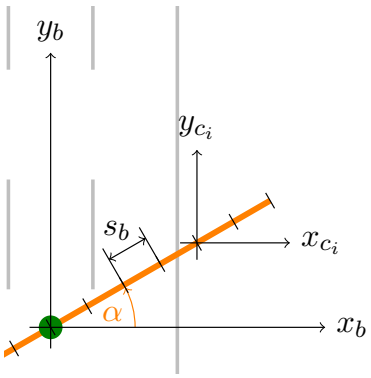


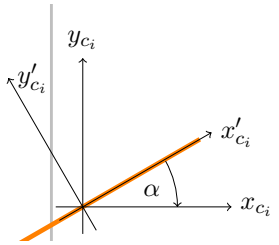
Abbildung 5.1: Koordinatensysteme



(a) Verschiebung von b nach w



(b) Verschiebung von c_i nach b



(c) Rotation von c'_i nach c_i

Abbildung 5.2: Transformationen zwischen den Koordinatensystemen

Die Verschiebung V_b, die einen Punkt aus dem Koordinatensystem b in das Koordinatensystem w überführt (vgl. Abbildung 5.2a), ist abhängig von der Position der Baseline $(x_{w_0}, y_{w_0})^T$ und gegeben durch

$$V_b : \mathbb{R}^2 \to \mathbb{R}^2, \begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} x_{w_0} \\ y_{w_0} \end{pmatrix}. \tag{5.1}$$

Die Verschiebung V_{c_i}, die einen Punkt aus dem Koordinatensystem c_i in das Koordinatensystem b überführt (vgl. Abbildung 5.2b) ist abhängig von α, der Schrittweite s_b sowie vom Diskretisierungspunkt i und gegeben durch

$$V_{c_i} : \mathbb{R}^2 \to \mathbb{R}^2, \begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} is_b \cos(\alpha) \\ is_b \sin(\alpha) \end{pmatrix}, i = -n_b, \ldots, n_b. \tag{5.2}$$

Die Rotation $R_{c_i'}$, die einen Punkt aus dem Koordinatensystem c_i' in das Koordinatensystem c_i überführt (vgl. Abbildung 5.2c) ist abhängig von α und gegeben durch

$$R_{c_i'} : \mathbb{R}^2 \to \mathbb{R}^2, \begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}. \tag{5.3}$$

Für die Transformation $W = V_b \circ V_{c_i} \circ R_{c_i'}$ von Koordinaten in c_i' zu Weltkoordinaten w gilt also

$$W : \mathbb{R}^2 \to \mathbb{R}^2, \begin{pmatrix} x_{c_i'} \\ y_{c_i'} \end{pmatrix} \mapsto \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \cdot \begin{pmatrix} x_{c_i'} \\ y_{c_i'} \end{pmatrix} + \begin{pmatrix} is_b \cos(\alpha) \\ is_b \sin(\alpha) \end{pmatrix} + \begin{pmatrix} x_{w_0} \\ y_{w_0} \end{pmatrix}. \tag{5.4}$$

Mit der Abbildung W können Punkte aus jedem c_i'-Koordinatensystem direkt in Weltkoordinaten transformiert werden. Dies wird im Folgenden für die Definition der Straßenmodelle genutzt.

5.2 Straßenmodelle

Im Koordinatensystem c_i' werden nun verschiedene Straßenmodelle definiert. Ähnlich wie bei der Baseline werden diese mit einer Schrittweite s_c in insgesamt $2n_c + 1$ Punkte diskretisiert.

5.2.1 Straßenmodell Gerade

Für eine Gerade als Straßenmodell bietet es sich an, die Gerade $x = 0$ im c_i'-Koordinatensystem zu verwenden. Wie in Abschnitt 5.2 beschrieben diskretisiert ergeben sich die Punkte

$$\begin{pmatrix} x_j \\ y_j \end{pmatrix} := \begin{pmatrix} 0 \\ js_c \end{pmatrix}, j = -n_c, \ldots, n_c. \tag{5.5}$$

Mit W aus Gleichung 5.4 ergeben sich die Punkte $\begin{pmatrix} x_w^{i,j} & y_w^{i,j} \end{pmatrix}^T$ des Abtastgitters

$$G = \left\{ \begin{pmatrix} x_w^{i,j} \\ y_w^{i,j} \end{pmatrix} \;\middle|\; i = -n_b, \ldots, n_b; \quad j = -n_c, \ldots, n_c \right\} \tag{5.6}$$

in Weltkoordinaten als

$$\begin{aligned} \begin{pmatrix} x_w^{i,j} \\ y_w^{i,j} \end{pmatrix} &:= W\left(\begin{pmatrix} x_j & y_j \end{pmatrix}^T \right) \\ &= \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \cdot \begin{pmatrix} 0 \\ js_c \end{pmatrix} + \begin{pmatrix} is_b \cos(\alpha) \\ is_b \sin(\alpha) \end{pmatrix} + \begin{pmatrix} x_{w_0} \\ y_{w_0} \end{pmatrix} \\ &= \begin{pmatrix} -js_c \sin(\alpha) + is_b \cos(\alpha) + x_{w_0} \\ js_c \cos(\alpha) + is_b \sin(\alpha) + y_{w_0} \end{pmatrix} \\ &\text{mit } i = -n_b, \ldots, n_b \text{ und } j = -n_c, \ldots, n_c. \end{aligned} \tag{5.7}$$

Mit Gleichung 5.7 und Gleichung 5.6 werden die beiden Diskretisierungsparametern n_c und n_b sowie dem Parameter α auf ein Abtastgitter in Weltkoordinaten abgebildet.

5.2.2 Straßenmodell Parabel

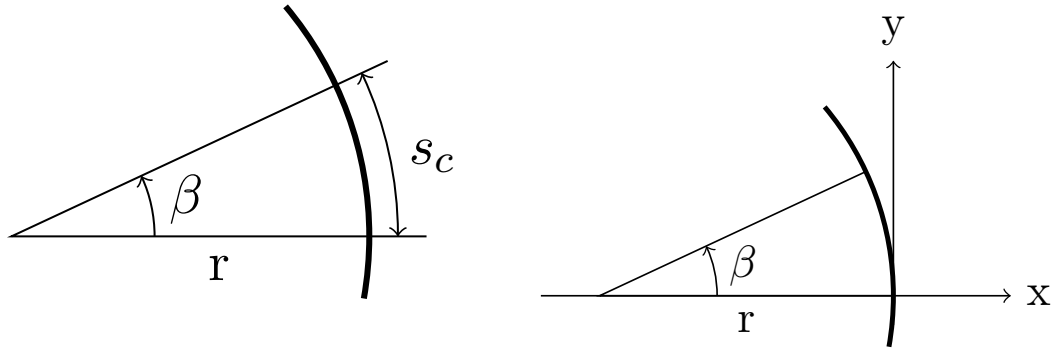
Für eine Parabel als Straßenmodell wird $x = a \cdot y^2$ im c_i'-Koordinatensystem verwendet. Dies entspricht einer gestreckten Normalparabel, bei der x und y vertauscht sind. Analog zu Unterabschnitt 5.2.1 ergeben sich die Punkte

$$\begin{pmatrix} x_j \\ y_j \end{pmatrix} := \begin{pmatrix} a(js_c)^2 \\ js_c \end{pmatrix}, j = -n_c, \ldots, n_c. \tag{5.8}$$

Auch hier kann ein Abtastgitter in Weltkoordinaten definiert werden. Dessen Punkte ergeben sich analog zu Gleichung 5.7 als

$$\begin{pmatrix} x_w^{i,j} \\ y_w^{i,j} \end{pmatrix} = \begin{pmatrix} a(js_c)^2\cos(\alpha) - js_c\sin(\alpha) + is_b\cos(\alpha) + x_{w_0} \\ a(js_c)^2\sin(\alpha) + js_c\cos(\alpha) + is_b\sin(\alpha) + y_{w_0} \end{pmatrix} \tag{5.9}$$
$$\text{mit } i = -n_b, \ldots, n_b \text{ und } j = -n_c, \ldots, n_c.$$

5.2.3 Straßenmodell Kreissegment



(a) Schrittweite im Kreismodell

(b) Kreissegment

Abbildung 5.3: Veranschaulichungen für das Kreismodell

Für ein Kreissegment als Straßenmodell wird die Definition der Schrittweite geändert. Wie in Abbildung 5.3a zu sehen bezieht sich s_c hier nicht mehr auf eine Koordinatenachse, sondern auf die Länge des Kreisbogens. Der korrespondierende Winkel β im Bogenmaß lässt sich wie folgt berechnen:

$$s_c = 2\pi r\frac{\beta}{2\pi} = r\beta \Rightarrow \beta = \frac{s_c}{r} \text{ für } r \neq 0 \tag{5.10}$$

Gemäß dem in Abbildung 5.3b dargestellten Modell ergeben sich analog zu Unterabschnitt 5.2.1 die Punkte

$$\begin{pmatrix} x_j \\ y_j \end{pmatrix} := \begin{pmatrix} r\cos(j\beta) - r \\ r\sin(j\beta) \end{pmatrix}, j = -n_c, \ldots, n_c. \tag{5.11}$$

Als Parameter soll im Folgenden nicht der Radius r, sondern die Krümmung k dienen.

Grund dafür ist die bessere Anschaulichkeit für das geplante Einsatzgebiet („Die Straße hat wenig/viel Krümmung“ statt „Die Straße hat einen großen/kleinen Radius“).

Mit

$$r = \frac{1}{k} \text{ für } k \neq 0 \tag{5.12}$$

gilt für den Winkel β aus Gleichung 5.10:

$$\beta = s_c k \tag{5.13}$$

Eingesetzt in Gleichung 5.11 folgt

$$\begin{pmatrix} x_j \\ y_j \end{pmatrix} = \begin{pmatrix} \frac{1}{k}\left(\cos(js_ck) - 1\right) \\ \frac{1}{k}\sin(js_ck) \end{pmatrix}, j = -n_c, \ldots, n_c \text{ und } k \neq 0. \tag{5.14}$$

Wie in Unterabschnitt 5.2.1 wird ein Abtastgitter in Weltkoordinaten definiert. Dessen Punkte ergeben sich analog zu Gleichung 5.7 als

$$\begin{pmatrix} x_w^{i,j} \\ y_w^{i,j} \end{pmatrix} = \begin{pmatrix} \frac{1}{k}\cos(\alpha)\left(\cos(js_ck) - 1\right) - \frac{1}{k}\sin(\alpha)\sin(js_ck) + is_b\cos(\alpha) + x_{w_0} \\ \frac{1}{k}\sin(\alpha)\left(\cos(js_ck) - 1\right) + \frac{1}{k}\cos(\alpha)\sin(js_ck) + is_b\sin(\alpha) + y_{w_0} \end{pmatrix} \tag{5.15}$$

$$\text{mit } i = -n_b, \ldots, n_b,\ j = -n_c, \ldots, n_c \text{ und } k \neq 0.$$

Durch die Anwendung von Additionstheoremen ergibt sich

$$\begin{pmatrix} x_w^{i,j} \\ y_w^{i,j} \end{pmatrix} = \begin{pmatrix} \frac{1}{k}\left(\cos(\alpha + js_ck) - \cos(\alpha)\right) + is_b\cos(\alpha) + x_{w_0} \\ \frac{1}{k}\left(\sin(\alpha + js_ck) - \sin(\alpha)\right) + is_b\sin(\alpha) + y_{w_0} \end{pmatrix} \tag{5.16}$$

$$\text{mit } i = -n_b, \ldots, n_b,\ j = -n_c, \ldots, n_c \text{ und } k \neq 0.$$

Vermeidung der Definitionslücke bei $k = 0$

Um die Definitionslücke der Funktion bei einer Krümmung $k = 0$ – also einer Geraden – zu vermeiden kann der Sinus Cardinalis (sinc) eingesetzt werden, der an seiner Definitionslücke stetig fortgesetzt wird:

$$\operatorname{sinc}(x) = \begin{cases} 1 & x = 0 \\ \frac{\sin(x)}{x} & \text{sonst} \end{cases} \tag{5.17}$$

Durch Anwendung weiterer Additionstheoreme auf Gleichung 5.16 ergibt sich zunächst

$$\begin{pmatrix} x_w^{i,j} \\ y_w^{i,j} \end{pmatrix} = \begin{pmatrix} \frac{1}{k} \cdot 2 \sin\left(\frac{js_c k}{2}\right) \sin\left(\frac{js_c k}{2} + \alpha\right) + is_b \cos(\alpha) + x_{w_0} \\ \frac{1}{k} \cdot 2 \sin\left(\frac{js_c k}{2}\right) \cos\left(\frac{js_c k}{2} + \alpha\right) + is_b \sin(\alpha) + y_{w_0} \end{pmatrix} \tag{5.18}$$
$$\text{mit } i = -n_b, \ldots, n_b,\ j = -n_c, \ldots, n_c \text{ und } k \neq 0,$$

durch Einsetzen von Gleichung 5.17 schließlich

$$\begin{pmatrix} x_w^{i,j} \\ y_w^{i,j} \end{pmatrix} = \begin{pmatrix} js_c \text{sinc}\left(\frac{js_c k}{2}\right) \sin\left(\frac{js_c k}{2} + \alpha\right) + is_b \cos(\alpha) + x_{w_0} \\ js_c \text{sinc}\left(\frac{js_c k}{2}\right) \cos\left(\frac{js_c k}{2} + \alpha\right) + is_b \sin(\alpha) + y_{w_0} \end{pmatrix} \tag{5.19}$$
$$\text{mit } i = -n_b, \ldots, n_b \text{ und } j = -n_c, \ldots, n_c.$$

5.3 Gesamtfunktion

Die Gesamtfunktion h aus Gleichung 4.1 wird hier zum besseren Verständnis noch einmal eingefügt:

$$h : \mathbb{R}^P \xrightarrow{g} \mathbb{R}^{2BH} \xrightarrow{\iota} \mathbb{R}^{BH} \xrightarrow{s} \mathbb{R}^B \xrightarrow{r} \mathbb{R}.$$

Sie kann für verschiedenen Straßenmodelle definiert werden – dabei gilt $B = 2n_b + 1$ und $C = 2n_c + 1$.

Exemplarisch wird dies nun für das Linienmodell aus Unterabschnitt 5.2.1 gezeigt. Dieses hat einen Eingangsparameter ($P = 1$) und definiert in Gleichung 5.7 die einzelnen Elemente des Abtastgitters (Gleichung 5.6). Auch dieses zum besseren Verständnis noch einmal eingefügt:

$$G = \left\{ \begin{pmatrix} x_w^{i,j} \\ y_w^{i,j} \end{pmatrix} \;\middle|\; i = -n_b, \ldots, n_b; \quad j = -n_c, \ldots, n_c \right\}$$

Die Menge wird für die Abbildung g der Gesamtfunktion in lexikographischer Ordnung als Vektor aufgeschrieben. Dieser enthält in der oberen Hälfte die x_w-Komponenten und in der unteren Hälfte die zugehörigen y_w-Komponenten.
Zur besseren Veranschaulichung der Struktur wurde die Notation absichtlich etwas ausführlicher gewählt.

Für g im Linienmodell gilt:

$$g : \mathbb{R} \to \mathbb{R}^{2BC}, x \mapsto \left(\begin{array}{c} \left. \begin{array}{c} -(-n_c)s_c \sin(x) + (-n_b)s_b \cos(x) + x_{w_0} \\ -(-n_c+1)s_c \sin(x) + (-n_b)s_b \cos(x) + x_{w_0} \\ \cdots \\ -(+n_c)s_c \sin(x) + (-n_b)s_b \cos(x) + x_{w_0} \\ -(-n_c)s_c \sin(x) + (-n_b+1)s_b \cos(x) + x_{w_0} \\ \vdots \\ -(+n_c)s_c \sin(x) + (+n_b)s_b \cos(x) + x_{w_0} \end{array} \right\} x_w\text{-Komponenten} \\ \hdashline \left. \begin{array}{c} (-n_c)s_c \cos(x) + (-n_b)s_b \sin(x) + y_{w_0} \\ \vdots \\ (+n_c)s_c \cos(x) + (+n_b)s_b \sin(x) + y_{w_0} \end{array} \right\} y_w\text{-Komponenten} \end{array} \right) \tag{5.20}$$

Es folgt die Abbildung ι, welche den Graustufenwert an einer Stelle im Bild ermittelt. Das Bild wird hier als kontinuierliche Abbildung $I : \mathbb{R}^2 \to \mathbb{R}$ angenommen.

Für ein diskretes Bild endlicher Größe liefert diese für alle Punkte außerhalb des Bildes den Wert 0. Innerhalb des Bildes wird der Wert kontinuierlich aus den Grauwerten der benachbarten Pixeln interpoliert.

Basierend auf der Aufteilung der x- und y-Komponenten in g gilt:

$$\iota : \mathbb{R}^{2BC} \to \mathbb{R}^{BC}, x \mapsto \begin{pmatrix} I\left(x_1, x_{(BC+1)}\right) \\ \vdots \\ I\left(x_{(BC)}, x_{(2BC)}\right) \end{pmatrix} \tag{5.21}$$

Aus diesen Intensitäten wird nun über die Abbildung s ein Histogramm entlang der Baseline gebildet:

$$s : \mathbb{R}^{BC} \to \mathbb{R}^{B}, x \mapsto \begin{pmatrix} \sum_1^C x_i \\ \sum_{C+1}^{2C} x_i \\ \vdots \\ \sum_{((B-1)C+1)}^{BC} x_i \end{pmatrix} \tag{5.22}$$

Dieses Histogramm wird schließlich mit der Abbildung r auf seine Quadratsumme reduziert:

$$r : \mathbb{R}^B \to \mathbb{R}, x \mapsto \sum_{1}^{B} (x_i)^2 . \tag{5.23}$$

Durch die Verkettung dieser Teilfunktionen entsteht die Gesamtfunktion:

$$h : \mathbb{R} \to \mathbb{R}, x \mapsto r\left(s\left(i\left(g(x)\right)\right)\right). \tag{5.24}$$

5.4 Gradienten

Um die optimalen Parameter der in Gleichung 5.24 definierten Gesamtfunktion h zu finden wird ein nummerisches Optimierungsverfahren angewendet. Einige Verfahren benötigen über den Funktionswert hinaus auch den Gradienten der Funktion. Um diesen für die Gesamtfunktion zu berechnen, kann, wie in Abschnitt 5.3, die Funktion zunächst in Teilen betrachten werden. Diese können jeweils einzeln abgeleitet und dann durch Anwendung der Kettenregel zum Gradienten der Gesamtfunktion zusammengesetzt werden.

Für die Darstellung der Ableitung wird die Jacobi-Matrix [9] verwendet.

Reduktion

Für die Ableitung der Reduktion r (Gleichung 5.23) gilt:

$$J_r(x) := \begin{pmatrix} \frac{\partial r}{\partial x_1} & \frac{\partial r}{\partial x_2} & \dots & \frac{\partial r}{\partial x_B} \end{pmatrix} = \begin{pmatrix} 2x_1 & 2x_2 & \dots & 2x_B \end{pmatrix}. \tag{5.25}$$

Summierung zum Histogramm

Für die Ableitung der Summierung s (Gleichung 5.22) zum Histogramm gilt:

$$\begin{aligned} J_s(x) := & \begin{pmatrix} \frac{\partial s_1}{\partial x_1} & \cdots & \frac{\partial s_1}{\partial x_{(BC)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial s_B}{\partial x_1} & \cdots & \frac{\partial s_B}{\partial x_{(BC)}} \end{pmatrix} \\ = & \begin{pmatrix} X_1 & & & \\ & X_2 & & \\ & & \ddots & \\ & & & X_B \end{pmatrix} \\ & \text{mit } X_i = \begin{pmatrix} x_{((i-1)C+1)} & \cdots & x_{(iC)} \end{pmatrix}, i = 1, \ldots, B. \end{aligned} \tag{5.26}$$

Bildabbildung

Für die Ableitung der Abbildung ι (Gleichung 5.21) im Bild wird der Bildgradient ∇I benötigt. Die resultierende Jacobi-Matrix J_ι enthält in der linken Hälfte die Bildableitung in x-Richtung und in der rechten Hälfte die Bildableitung in y-Richtung.

Es gilt:

$$\begin{aligned} J_\iota(x) := & \begin{pmatrix} \frac{\partial \iota_1}{\partial x_1} & \cdots & \frac{\partial \iota_1}{\partial x_{(2BC)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \iota_{(BC)}}{\partial x_1} & \cdots & \frac{\partial \iota_{(BC)}}{\partial x_{(2BC)}} \end{pmatrix} \\ = & \left(\begin{array}{ccc:ccc} \nabla I_x(x_1, x_{BC+1}) & & & \nabla I_y(x_1, x_{BC+1}) & & \\ & \ddots & & & \ddots & \\ & & \nabla I_x(x_{(BC)}, x_{(2BC)}) & & & \nabla I_y(x_{(BC)}, x_{(2BC)}) \end{array}\right) \end{aligned} \tag{5.27}$$

Straßenmodell

Die Ableitung der Funktion g (Gleichung 5.20) ist (auch wieder mit etwas ausführlicherer Notation):

$$J_g(x) := \begin{pmatrix} \frac{\partial g_1}{\partial x} \\ \vdots \\ \frac{\partial g_{(2n_c n_b)}}{\partial x} \end{pmatrix} = \left(\begin{array}{c} -(-n_c)s_c\cos(x) - (-n_b)s_b\sin(x) \\ -(-n_c+1)s_c\cos(x) - (-n_b)s_b\sin(x) \\ \cdots \\ -(+n_c)s_c\cos(x) - (-n_b)s_b\sin(x) \\ -(-n_c)s_c\cos(x) - (-n_b+1)s_b\sin(x) \\ \vdots \\ -(+n_c)s_c\cos(x) - (+n_b)s_b\sin(x) \\ \hdashline -(-n_c)s_c\sin(x) + (-n_b)s_b\cos(x) \\ \vdots \\ -(+n_c)s_c\sin(x) + (+n_b)s_b\cos(x) \end{array}\right) \tag{5.28}$$

Die Anzahl der Spalten in J_g ist abhängig von der Anzahl der Parameter P (hier beim Linienmodell 1). Die Anzahl der Zeilen ist unabhängig vom Straßenmodell immer $2BC$. Bei einem Wechsel des Straßenmodells ändert sich lediglich diese Ableitung. Die anderen Ableitungen sind unabhängig vom Straßenmodell und gelten weiterhin unverändert.

5.4.1 Gesamtgradient

Der Gesamtgradient ∇h der Funktion h (Gleichung 5.24) wird über die Kettenregel zusammengesetzt. Mit den Jacobi-Matrizen aus Abschnitt 5.4 gilt:

$$\nabla h = J_h(x) := J_r \cdot J_s \cdot J_\iota \cdot J_g \tag{5.29}$$

6 Umsetzung

Auf Basis des in Kapitel 4 eingeführten Konzepts und der in Kapitel 5 beschriebenen mathematischen Grundlage wird im Folgenden auf die Implementierung des Algorithmus eingegangen.

6.1 Straßenmodell

Die in Abschnitt 5.2 vorgestellten mathematischen Straßenmodelle müssen gemäß Anforderung NF2 so implementiert werden, dass sie leicht austauschbar sind. Da alle Modelle bis auf die Anzahl der Parameter dieselben Ein- und Ausgaben haben – es werden Parameter (α, a, c) sowie Konstanten (s_c, s_b, x_{w_0}, y_{w_0}) und Indizes (i, j) auf 2D-Koordinaten abgebildet – wird folgende Schnittstelle erstellt:

```
Vector<2> indicesToWorldCoordinates(const ParameterVector& params,
                                    const GridConstants& gridConstants,
                                    const Indices& indices)
```

`ParameterVector` ist dabei ein Alias, der von dem jeweils konkreten Straßenmodell gemäß der Anzahl der Parameter definiert wird. Andere Programmteile, die das Straßenmodell generisch nutzen, verwenden diesen Alias, um die korrekte Speichergröße bereitzustellen.
`GridConstants` enthält die Konstanten und ist für alle Straßenmodelle gleich.
`Indices` enthält die beiden Indizes und ist ebenfalls für alle Straßenmodelle gleich.
Der Rückgabewert `Vector<2>` beschreibt einen Punkt mit x- und y-Komponente im Weltkoordinatensystem (w).

Alle Implementierungen der Straßenmodelle bieten diese Schnittstelle an. Über einen zentralen Alias wird vor dem Kompilieren festgelegt, welches konkrete Straßenmodell eingesetzt wird. Zur Laufzeit ist dieses nicht austauschbar.

Die Anforderung NF3 fordert die Austauschbarkeit der Berechnungsplattform. Damit die Funktion auf der GPU in einem CUDA-Kernel genutzt werden kann, muss sie mit dem Attribut `__device__` gekennzeichnet werden (soll die Funktion außerdem auch auf der CPU verfügbar sein, muss zusätzlich noch das Attribut `__host__` hinzugefügt werden). Diese Attribute sind jedoch spezifisch für den NVIDIA CUDA Compiler (nvcc). Damit auch ein von CUDA unabhängiges, reines, C++ Programm mit der GNU Compiler Collection (gcc) kompiliert werden kann, welche diese Attribute nicht kennt, werden die Attribute durch Präprozessor-Makros hinzugefügt. Diese prüfen den verwendeten Compiler indem die Präprozessor-Direktive `__CUDACC__` geprüft wird, welche nur vom nvcc gesetzt wird.

Für die Funktionsauswertung wurde parallel zur CUDA-Version eine Referenz in C++ gepflegt. Da die beiden Implementierungen dieselbe Schnittstelle bereitstellen, können diese ausgetauscht werden. Für die Fourriertransformation im Klassifikationsschritt existiert jedoch aufgrund des begrenzten zeitlichen Rahmens dieser Arbeit nur eine Version mit der CUDA-Bibliothek cuFFT [20]. Diese Funktionalität könnte zukünftig jedoch auch mithilfe einer anderen Bibliothek auf der CPU bereitgestellt werden.

6.2 Funktionsauswertung

Die in Abschnitt 5.3 mathematisch beschriebene Funktion wurde zunächst auf der CPU umgesetzt.

6.2.1 Basis

Die Berechnung des Funktionswerts erfolgt gemäß des Pseudocodes in Listing 6.1.

In zwei geschachtelten Schleifen wird jeder Punkt des Abtastgitters betrachtet. Dabei wird der Abtastpunkt zuerst über die in Abschnitt 6.1 beschriebene Schnittstelle der Straßenmodelle in das Weltkoordinatensystem abgebildet (entspricht g aus Gleichung 5.20). Dann wird der Graustufenwert an dieser Stelle im Bild ermittelt – wenn der berechnete Bildpunkt nicht exakt auf einem Pixel liegt wird der Intensitätswert über eine bilineare Interpolation aus den benachbarten Pixeln ermittelt (entspricht ι aus Gleichung 5.21). In einer Variablen werden zunächst alle Intensitätswerte entlang der Fahrspurhypothese aufsummiert (entspricht s aus Gleichung 5.22). In einer weiteren

Variable werden die quadrierten Ergebnisse der einzelnen Fahrspurhypothesen aufsummiert (entspricht r aus Gleichung 5.23). Am Ende wird diese Summe zurückgegeben.

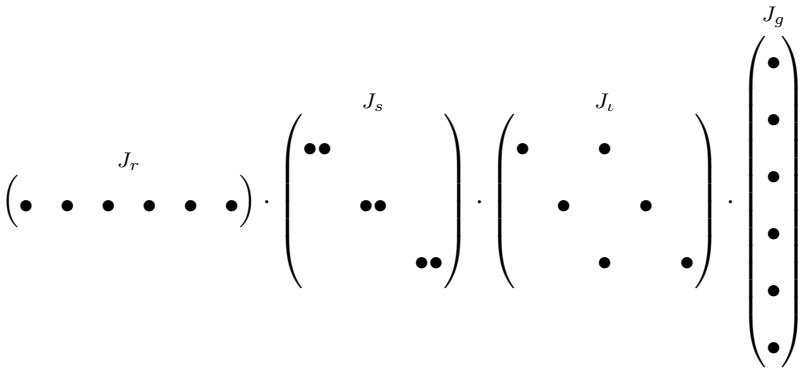
```
1  funktionswert = 0
2  for (alle baseline Punkte)
3  {
4    hypothesenwert = 0
5    for (alle Punkte auf dieser Fahrspurhypothese)
6    {
7      weltkoordinaten = indicesToWorldCoordinates(...) // g
8      grauwert = imageAt(...)                          // i
9      hypothesenwert += grauwert                       // s
10   }
11   funktionswert += hypothesenwert^2                  // r
12 }
13 return funktionswert
```

Listing 6.1: Pseudocode der Funktionswertberechnung

6.2.2 Berechnung des Gradienten

Im Abschnitt 5.4 wurde der Gesamtgradient als Multiplikation von vier Jacobi-Matrizen definiert (Gleichung 5.29). Diese Art der Berechnung benötigt allerdings sehr viel Speicher und Rechenkapazität. Basierend auf dem Ansatz von Rühaak et al. [14] wird die Berechnung im Folgenden matrixfrei durchgeführt.

Betrachtet man die Struktur der Jacobi-Matrizen, ergibt sich folgendes Bild:



Aufgrund der Diagonalstruktur von J_s und J_ι ergibt die Multiplikation $J_{rs\iota} = J_r \cdot J_s \cdot J_\iota$ einen Vektor, dessen Elemente unabhängig voneinander berechnet werden können. J_r und J_s enthalten lediglich Konstanten. J_ι enthält Bildgradienten, die – solange sich das Bild nicht ändert – ebenfalls konstant sind. Zudem können auch diese unabhängig voneinander berechnet werden. Insgesamt ergibt sich somit ein Vektor aus unabhängig voneinander berechenbaren Konstanten.
Bei der Multiplikation $J_{rs\iota} \cdot J_g$ wird jedes Element aus J_g zunächst mit einer dieser Konstanten multipliziert, was ebenfalls jeweils unabhängig voneinander berechnet werden kann. Danach folgt abschließend nur noch die Summation dieser Produkte.

Die Unabhängigkeit in der Berechnung von Teilen des Gradienten wird im Folgenden bei der Erweiterung des Programs aus Unterabschnitt 6.2.1 ausgenutzt. Unabhängig bedeutet ebenfalls, dass die Berechnungen parallelisiert werden können. Dies wird in Kapitel 7 ausgenutzt.

Der erweiterte Pseudocode ist in Listing 6.2 dargestellt. Der Algorithmus ist bis zur Zeile 11 wie Listing 6.1, danach wird jedoch noch für jeden Punkt der Bildgradient (Element von J_ι aus Gleichung 5.27) sowie der Gradient des Straßenmodells (Element von J_g aus Gleichung 5.28) berechnet. Diese beiden Teile des Gesamtgradienten werden dann schon zusammengefasst und in den Teilgradienten der Linie aufsummiert (Element von J_s aus Gleichung 5.26). Der Gradient jeder Hypothese wird dann schließlich mit deren Funktionswert skaliert und aufsummiert (J_r aus Gleichung 5.25), bevor er zurückgegeben wird.

Zu beachten ist dabei, dass es zwischen der Berechnung des Funktionswerts und der Berechnung des Gradienten Synergieeffekte gibt. Beispielsweise werden sowohl zur Interpolation als auch zur Ermittelung des Bildgradienten dieselben Pixel benötigt. Deren Grauwerte müssen also nur einmal aus dem Speicher geladen werden.
Zudem geht der Funktionswert in Zeile 18 in den Gradienten ein – zur Berechung des Gradienten wird also immer eine vorherige Berechnung des Funktionswerts benötigt.

```
1  funktionswert = 0
2  gradient = 0
3  for (alle baseline Punkte)
4  {
5    hypothesenwert = 0
6    hypothesengradient = 0
7    for (alle Punkte auf dieser Fahrspurhypothese)
8    {
9      weltkoordinaten = indicesToWorldCoordinates(...)      // g
10     grauwert = imageAt(...)                               // i
11     hypothesenwert += grauwert                            // s
12
13     bildgradient = imageGradient(...)                     // Ji
14     modellgradient = berechneGradient(...)                // Jg
15     hypothesengradient += modellgradient * bildgradient   // Js
16   }
17   funktionswert += hypothesenwert^2                       // r
18   gradient += hypothesenwert * hypothesengradient         // Jr
19 }
20 return funktionswert
21 return gradient
```

Listing 6.2: Pseudocode der Funktionswertberechnung mit Gradienten

6.3 Extraktion der Fahrspurhypothesen

Unterabschnitt 4.1.2 fordert eine Extraktion der Spitzen aus dem Histogramm, da diese zu den Fahrspurmarkierungen korrespondieren.

In der Realität sieht das Histogramm jedoch nicht so ideal aus, wie in Abbildung 4.5 dargestellt. Die Spitzen sind etwas breiter und es enthält Rauschen. Abbildung 6.1 zeigt ein Beispiel. Die Extraktion der Fahrspuren erfolgt daher in mehreren Schritten. Im ersten Schritt werden zunächst alle Spitzen aus dem Histogramm extrahiert. Da diese

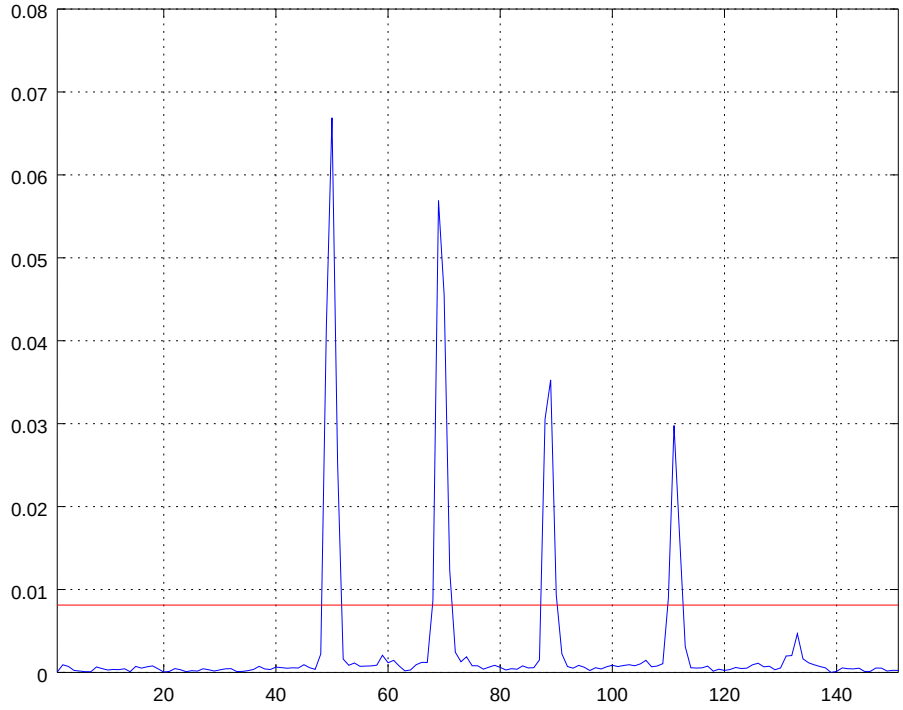


Abbildung 6.1: Reales Histogramm

häufig auch im Rauschen auftreten, werden im zweiten Schritt nur noch signifikante Spitzen betrachtet. In den vorliegenden Daten hat sich dafür ein Schwellwert von

$$\overline{x} + \frac{1}{2}\sigma_x \tag{6.1}$$

mit

$\overline{x}$: Mittelwert des Histogramms,

σ_x : Standardabweichung des Histogramms,

bewährt. Dieser ist in Abbildung 6.1 als rote Linie eingetragen.

Zuletzt werden noch gemäß Abschnitt 4.2 Spitzen entfernt, die zu nah beieinander liegen. Das Ergebnis sind die erkannten Fahrspurmarkierungen.

6.4 Klassifikation der Fahrspurmarkierungen

Die Klassifikation nutzt aus, dass die Länge der Striche und Lücken bekannt ist. Auf deutschen Autobahnen werden hier zum Beispiel 6m Strichlänge mit 12m Lücken erwartet [5, Teil 1, S. 5] – als kontinuierliches Signal gedacht also eine Wellenlänge von $\lambda = 18\text{m}$.

Im Frequenzraum wird demnach bei gestrichelten Linien eine Spitze bei der zugehörigen Frequenz $f = \frac{1}{\lambda}$ erwartet. Da hier die diskrete Fouriertransformation eingesetzt wird, welche nicht alle Wellenlängen abbilden kann, und um robust gegen leichte Variationen zu sein, wird ein Toleranzfenster eingesetzt. Bei den vorliegenden Daten hat sich dafür eine Größe von $\pm 2m$ bewährt.

Um Rauschen auszuschließen, wird auch hier wieder eine signifikante Spitze erwartet. In den vorliegenden Daten hat sich als Schwellwert

$$\frac{1}{2}F(0) \tag{6.2}$$

bewährt. $F(0)$ repräsentiert dabei den konstanten Anteil des Signals. Überschreitet die Spitze den Schwellwert, wird die Linie als gestrichelt klassifiziert.

Validierung des Extraktionsschritts

Abschnitt 4.2 beschreibt eine Validierung der Fahrspurhypothese im Frequenzraum über einen signifikanten konstanten Anteil.

Als Schwellwert dafür hat sich bei den vorliegenden Daten

$$4\overline{F} \tag{6.3}$$

mit

$\overline{F}$: Mittelwert des Frequenzspektrums,

bewährt.

6.5 Struktur des Gesamtalgorithmus

Das Konzept aus Kapitel 4 beschreibt für den Gesamtalgorithmus mehrere Schritte. Diese sind in Abbildung 6.2 nochmals grafisch dargestellt.

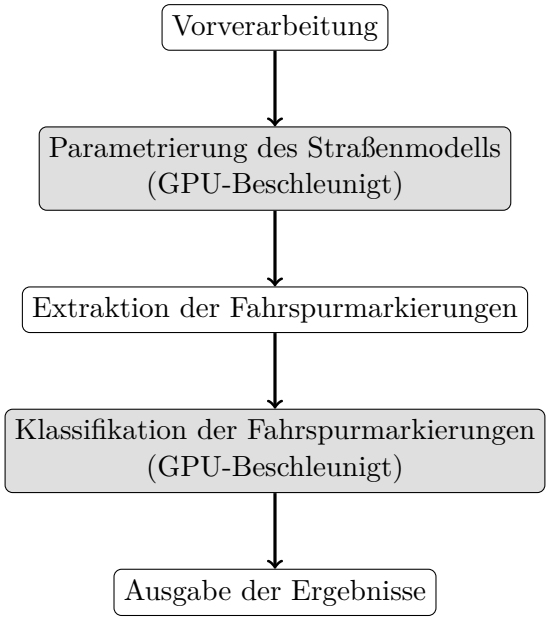


Abbildung 6.2: Ablauf des Gesamtalgorithmus

Der Vorverarbeitungsschritt wurde in dieser Arbeit nicht weiter betrachtet. Er wird ausschließlich auf der CPU ausgeführt.

Im Parametrierungsschritt wird eine nummerische Optimierung ausgeführt (vgl Unterabschnitt 4.1.1). Auch dies passiert primär auf der CPU, jedoch wird der parallelisierbare und rechenaufwändige Schritt der Berechnung des Funktionswertes der Zielfunktion in Kapitel 7 auf die GPU ausgelagert.

Der Extraktionsschritt (siehe Unterabschnitt 4.1.2) wird ausschließlich auf der CPU ausgeführt. Er ist nicht rechenaufwändig und lässt sich schlecht parallelisieren, daher wäre es nicht sinnvoll hier eine GPU-Beschleunigung anzustreben.

Der Klassifikationsschritt nutzt die parallelisierbare und rechenintensive Fouriertransformation (siehe Unterabschnitt 4.1.3). Diese wurde mithilfe einer Bibliothek auf der GPU beschleunigt.

Auch die Ausgabe der Ergebnisse (vgl Unterabschnitt 4.1.4) ist nicht rechenaufwändig. Es werden im wesentlichen nur die Resultate der vorherigen Schritte gesammelt und strukturiert – auch dafür wäre eine GPU-Beschleunigung nicht angebracht.

6.6 Qualitätssicherung

Die Güte des Verfahrens wurde von Florian Homm et al. im dieser Arbeit zugrunde liegenden Paper [7] bereits evaluiert. Da für den hier genutzten Datensatz keine *ground-truth* Daten für eine automatisierte Validierung zur Verfügung stehen, erfolgt die Qualitätssicherung in dieser Arbeit „per Augenmaß“. In Abschnitt 8.4 wurden zudem einige Szenarien evaluiert.

Über eine Visualisierung und ein Referenzkamerabild wird geprüft, ob das Ergebnis hinsichtlich der Anzahl der Fahrspurmarkierungen und der Klassifizierung korrekt ist. Dabei können jedoch keine Aussagen über die Genauigkeit der Positionen und Parameter getroffen werden. Dies ist in Abbildung 6.3 dargestellt. Die grünen Linien zeigen dabei die erkannten Fahrspurmarkierungen sowie den Typ, und verblassen bei niedriger Konfidenz. Die Orange Box mit Pfeil zeigt das Fahrzeug und die Fahrtrichtung als Referenz an.

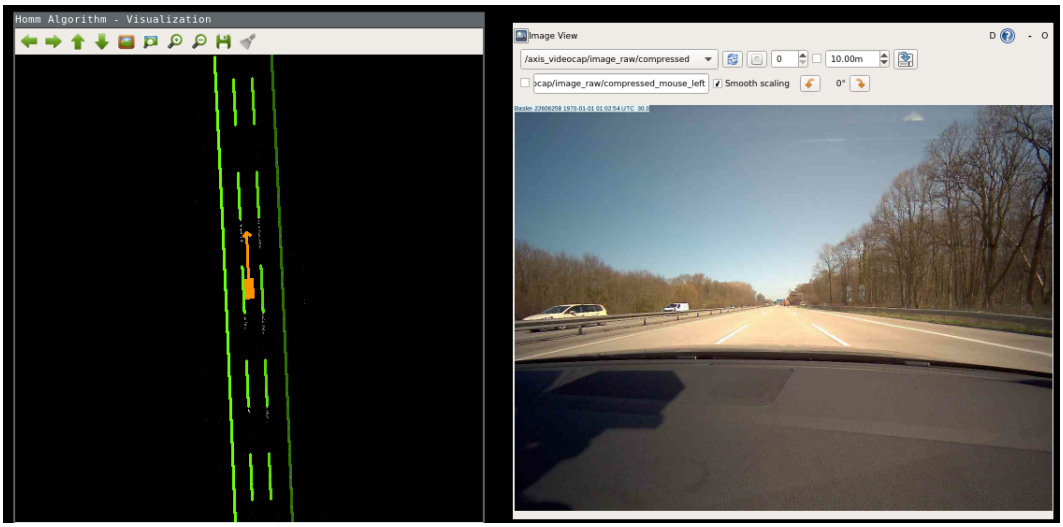


Abbildung 6.3: Visualisierung der Ausgabe

7 Optimierung

In Abschnitt 6.2 wurde die Umsetzung des Algorithmus anhand von Pseudocode erklärt. Um die Laufzeitanforderung NF1 zu erfüllen muss dieser nun optimiert werden.

In Abbildung 7.1 sind die Einstellungen dargestellt, die genutzt wurden, um die Laufzeiten in diesem Abschnitt zu ermitteln. Die dargestellte Laufzeit entspricht dabei jeweils dem Durchschnittswert aus 10 Durchläufen.
Die Auswertung wurde auf dem Entwicklungslaptop (NVIDIA GeForce 940MX) durchgeführt.

(a) Genutztes Bild

Straßenmodell	Linie
Berechnung bei	$\alpha = 0.1$[rad]
Berechnung von	Funktionswert
Bildgröße	500x500 Pixel
Schrittweite Baseline	0,2m
Schrittweite Hypothese	0,2m
Diskretisierung Baseline	151 Punkte
Diskretisierung Hypothese	501 Punkte

(b) Genutzte Parameter

Abbildung 7.1: Einstellungen bei der Ermittlung der Laufzeiten

Um den Algorithmus aus Unterabschnitt 6.2.2 zu beschleunigen kann man zunächst die Berechnung im Straßenmodell verändern. Dort werden bei jedem Aufruf der Sinus und der Cosinus der Straßenrichtung benötigt, deren Berechnung relativ teuer ist. Indem man diese Werte vorberechnet kann die Berechnungszeit fast halbiert werden (Tabelle 7.1).

	Laufzeit	Speedup
Basisversion	10,92 ms	
Vorberechnet	6,35 ms	1.7

Tabelle 7.1: Speedup bei Vorberechnung

Um noch schneller zu werden soll der Algorithmus auf einer GPU ausgeführt werden. Überträgt man ihn dafür 1:1 läuft dieser jedoch zunächst deutlich langsamer (Tabelle 7.2). Um von der Leistung der GPU zu profitieren muss der Algorithmus parallelisiert werden.

	Laufzeit	Speedup
CPU (Vorberechnet)	6,35 ms	
GPU (1:1)	137,5 ms	-21,6

Tabelle 7.2: GPU Implementierung 1:1

Als ersten Ansatz wird die äußere Schleife aus Zeile 3 in Listing 6.2 parallelisiert. Es ergibt sich ein Block, der so viele Threads enthält wie Abtastpunkte auf der Baseline (Fahrspurhypothesen) vorhanden sind. Jeder dieser Threads berechnet dabei eine einzelne Fahrspurhypothese. Am Ende werden alle Threads synchronisiert und der erste Thread berechnet die Summe bevor der Wert zurückgegeben wird.
Tabelle 7.3 zeigt, dass der Algorithmus dadurch um mehr als zwei Größenordnungen schneller ist, als die naive 1:1-Variante – und mehr als fünf mal schneller als die CPU-Variante.

	Laufzeit	Speedup
GPU (1:1)	137,5 ms	
GPU (Baseline parallelisiert)	1,18 ms	116,5
CPU (Vorberechnet)	6,35 ms	
GPU (Baseline parallelisiert)	1,18 ms	5,4

Tabelle 7.3: Speedup bei Parallelisierung der Baseline

Als Nächstes werden die Speicherbereiche angepasst. Die Diskretisierungsparameter sind aus Sicht des Algorithmus konstant und werden daher im Constant Memory abgelegt. Das Projektionsbild kann im Texture Memory abgelegt und über die Texture Unit abgefragt werden. Somit profitiert man von den spezialisierten Caching-Lösungen.
Die Texture Unit kann auch in Hardware zwischen den Pixeln interpolieren. Diese Funktionalität wird hier nicht genutzt, da sie für die Gewichtung der Pixel nur 8bit Genauigkeit hat. Dies führt zu kleinen Fehlern, die sich im Gesamtergebnis soweit akkumulieren, dass dieses inakzeptabel stark (in der Größenordnung 10^{-1}) von der Referenz auf der CPU abweicht.
Tabelle 7.4 zeigt, dass die Laufzeit hier nochmal auf fast ein Viertel reduziert werden kann – im Vergleich zur CPU ist der Algorithmus jetzt fast um den Faktor 20 schneller.

	Laufzeit	Speedup
GPU (Baseline parallelisiert)	1,18 ms	
GPU (Speicherzuweisung)	0,32 ms	3,7
CPU (Vorberechnet)	6,35 ms	
GPU (Speicherzuweisung)	0,32 ms	19,8

Tabelle 7.4: Speedup bei sinnvoller Speicherzuweisung

Trotz aller Verbesserungen kann das Potential der GPU jedoch nicht ganz ausgeschöpft werden. Um Latenzen zu verstecken werden mehr Threads benötigt. Dies erfordert grundlegende Änderungen am Algorithmus.

Die Fahrspurhypothesen werden in Abschnitte aus je 32 Abtastpunkten aufgeteilt – passend zu der Anzahl Threads in einem Warp. Die Berechnung erfolgt nun Abschnittsweise,

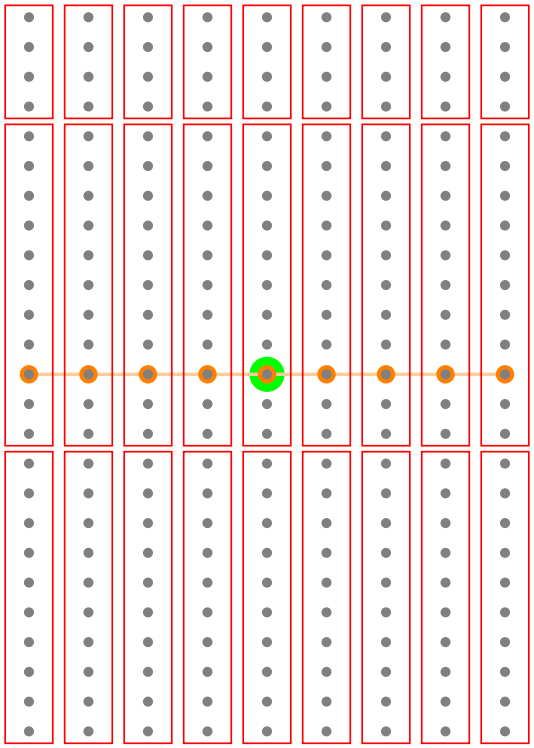


Abbildung 7.2: Prinzip der Aufteilung in Abschnitte

parallelisiert in mehreren Blöcken zu je 32 Threads. Veranschaulicht ist dies in Abbildung 7.2. Für jeden Abschnitt wird die Summe der Intensitätswerte des Abschnitts im globalen Speicher abgelegt.
Ist die Anzahl der Punkte pro Linie kein Vielfaches von 32 ist der letzte Abschnitt zwar trotzdem 32 Punkte lang, die Ergebnisse der ungültigen Punkte werden jedoch nicht verwendet.

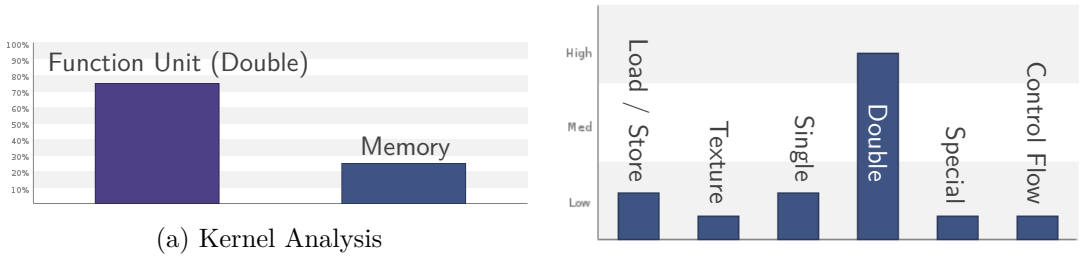
Da Blöcke unabhängig voneinander sind – und in der hier verwendeten CUDA-Version innerhalb eines Kernels noch nicht synchronisiert werden können – muss die Berechnung auf zwei Kernel aufgeteilt werden. Im ersten Kernel werden, wie oben beschrieben, Zwischenwerte für jeden Abschnitt berechnet, in einem Zweiten werden diese Zwischenwerte dann wie zuvor mit einem Block und so vielen Threads wie Fahrspurhypothesen zu den einzelnen Gesamtwerten für jede Fahrspurhypothese und schließlich zur Quadratsumme reduziert.

Tabelle 7.5 zeigt nochmals eine Reduktion der Laufzeit um ein Drittel – verglichen mit der CPU ist der Algorithmus nun mehr als 26 mal schneller.

	Laufzeit	Speedup
GPU (Speicherzuweisung)	0,32 ms	
GPU (Abschnitte)	0,24 ms	1,3
CPU (Vorberechnet)	6,35 ms	
GPU (Abschnitte)	0,24 ms	26,5

Tabelle 7.5: Speedup bei Aufteilung in Abschnitte

Zur weiteren Optimierung kann der NVIDIA Visual Profiler eingesetzt werden. Das Tool ermöglicht eine detaillierte Analyse der Ressourcenauslastung auf der GPU. Dabei zeigt sich, dass die Geschwindigkeit des Algorithmus durch eine hohe Auslastung der double-precision unit begrenzt ist (vgl. Abbildung 7.3).



(a) Kernel Analysis

(b) Compute Analysis

Abbildung 7.3: NVIDIA Visual Profiler

Durch Ersetzung mit single-precision floating point kann nochmals etwas mehr Geschwindigkeit herausgeholt werden (siehe Tabelle 7.6).

	Laufzeit	Speedup
GPU (Abschnitte - double)	0,24 ms	
GPU (Abschnitte - float)	0,18 ms	1,3

Tabelle 7.6: Speedup bei Reduktion auf float

Der NVIDIA Visual Profiler (vgl. Abbildung 7.4) zeigt nun eine gute Balance zwischen verschiedenen Recheneinheiten (*Compute*) und der Speicherauslastung.

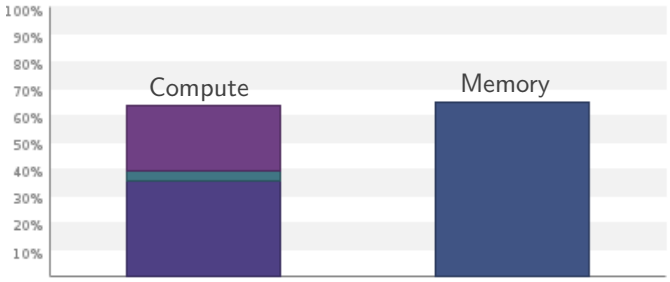


Abbildung 7.4: NVIDIA Visual Profiler - Kernel Analysis

Ein abschließender Vergleich der CPU mit der GPU zeigt, dass die GPU etwa 35 mal schneller ist. Im Vergleich der ersten, langsamsten Implementierung mit der schnellsten konnte insgesamt sogar mehr als das 60-fache an Geschwindigkeit erreicht werden (vgl. Tabelle 7.7).

	Laufzeit	Speedup
CPU (Vorberechnet)	6,35 ms	
GPU (Abschnitte - float)	0,18 ms	35,3
CPU (Basis)	10,92 ms	
GPU (Abschnitte - float)	0,18 ms	60,7

Tabelle 7.7: Speedup der GPU

7.1 Laufzeit des Gesamtalgorithmus

Für den in Abschnitt 6.5 dargestellten Gesamtalgorithmus gibt Anforderung NF1 insgesamt eine Maximallaufzeit von 40ms vor.

Auf die verwendete Version der Vorverarbeitung entfallen dabei im Mittel 20ms. Für den restlichen Algorithmus bleiben somit 20ms Laufzeit.

Die Laufzeit des Ausgabe- sowie des Extraktionsschritts ist vernachlässigbar. Die Laufzeitbeschränkung von 20ms beeinflusst nur die Klassifikation sowie die Parametrierung.

7.1.1 Laufzeit der Klassifikation

Zur Berechnung der Fouriertransformation wird die cuFFT-Bibliothek [20] eingesetzt. Mit den Diskretisierungsparametern aus Kapitel 7 ergeben sich auf dem Entwicklungslaptop für verschiedene Anzahlen von Fahrspuren in diesem Schritt folgende Laufzeiten:

Anzahl	1	2	3	5	10	100	151
Laufzeit	2.57 ms	3.33 ms	3.35 ms	3.40 ms	3.83 ms	4.70 ms	5.33 ms

Tabelle 7.8: Laufzeiten der Klassifikation

Um eine Maximallaufzeit abschätzen zu können, wird die Anzahl der zu klassifizierenden Fahrspurhypothesen auf 10 begrenzt. Der Verarbeitungsschritt hat dadurch eine geschätzte Maximallaufzeit von 4ms – es bleiben 16ms für die weiteren Schritte.

Zwar würde eine Verzehnfachung der Begrenzung die Laufzeit nur um ein Viertel erhöhen, die gewählte Beschränkung übersteigt jedoch bereits deutlich die zu erwartende Anzahl von Markierungen in der Anwendung Autobahnpilot (Anforderung F1). Somit wäre eine höhere Grenze nicht sinnvoll.

Sollte der vorherige Schritt mehr Hypothesen ermitteln, werden diese nach der Konfidenz sortiert und lediglich die 10 mit der höchsten Konfidenz klassifiziert. Die übrigen werden mit einer classification-confidence von 0 versehen und an den Ausgabeschritt weitergegeben.

7.1.2 Laufzeit der Parametrierung

Bei der Parametrierung des Straßenmodells wird ein Verfahren aus der nummerischen Optimierung angewendet. Dieses benötigt pro Iteration den Funktionswert. Die Laufzeit einer Funktionsauswertung wurde zuvor als 0,18ms bestimmt. In den verbliebenen 16ms können also

$$n_{max} = \left\lfloor \frac{16ms}{0,18ms} \right\rfloor = \lfloor 88.89 \rfloor = 88 \tag{7.1}$$

Iterationen durchgeführt werden. Diese Begrenzung wird im Optimierungsverfahren als Parameter eingestellt.

Sollte das Optimierungsverfahren innerhalb dieser Begrenzung nicht zu einem Ergebnis kommen, wird für den aktuellen Schritt eine Konfidenz von 0 zurückgemeldet. Die Werte für die Parameter aus der letzten Iteration bleiben als Startwert für den nächsten Durchlauf.

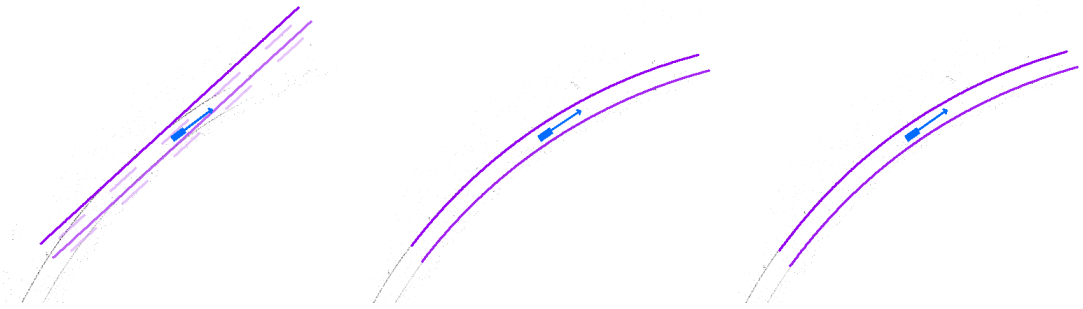
8 Evaluation

In diesem Kapitel werden verschiedene Konfigurationsoptionen des Algorithmus betrachtet sowie die Ergebnisse in einigen Szenarien ausgewertet. Zudem wird die Ausführung auf der Zielplattform betrachtet.

8.1 Auswahl des Straßenmodells

Auf Autobahnen reicht grundsätzlich das Geradenmodell. Zwar ist die Straße nicht perfekt gerade, aber Kurven sind meist so sanft, dass sie ohne Probleme durch eine Gerade approximiert werden können.
Um jedoch flexibler zu sein – und auch Ausfahrten abbilden zu können – werden ein Modell mit Kreisbögen und ein Quadratisches Modell betrachtet.

Abbildung 8.1 zeigt beispielhaft die Ergebnisse der verschiedenen Straßenmodelle in der Kurve einer Ausfahrt. Vergleicht man diese zeigt sich das Linienmodell erwartungsgemäß gänzlich ungeeignet. Die Ergebnisse von Kreismodell und Quadratischem Modell sind durch Anschauen nicht zu unterscheiden.



(a) Linienmodell (b) Kreismodell (c) Quadratisches Modell

Abbildung 8.1: Ergebnisse der Straßenmodelle in einer Ausfahrt (Farben invertiert)

Da keine Referenzdaten vorliegen, werden die Modelle anhand der Visualisierung und nach Augenmaß bewertet. Grundsätzlich lässt sich hier sagen, dass Kreismodell und Quadratisches Modell für die Anwendung vergleichbare Ergebnisse liefern. Das Kreismodell ist mathematisch komplexer und benötigt dadurch auch etwas mehr Laufzeit in der Funktionsauswertung. Daher wird das Quadratische Modell bevorzugt.

8.2 Auswahl des Optimierungsverfahrens

Es wurden die beiden in Unterabschnitt 4.1.1 vorgestellten Verfahren zur nummerischen Optimierung verglichen – das BFGS-Verfahren mit Armijo-Liniensuche und das Nelder-Mead-Verfahren.

Da die Funktionsauswertung eine teure Operation ist (vgl. Kapitel 7), wurde zunächst verglichen, wie viele solche Funktionsauswertungen die Verfahren benötigen bis sie konvergieren.
Erwartet wird hier, dass das BFGS-Verfahren weniger Auswertungen benötigt, da es den Gradienten nutzt und damit mehr Informationen über das Problem zur Verfügung hat, als das Nelder-Mead-Verfahren.

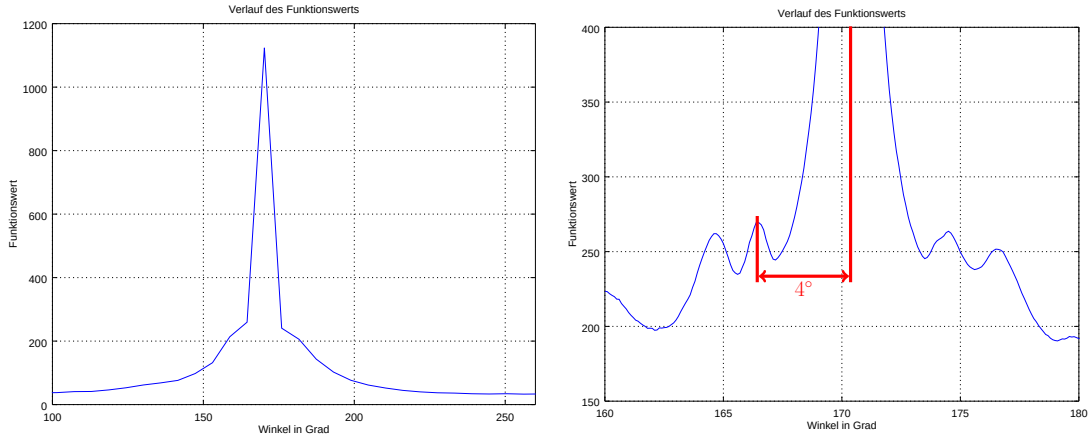
	Funktionswert	Funktionswert + Gradient	Laufzeit
Nelder-Mead	16	0	7 ms
BFGS	0	8	5 ms

Tabelle 8.1: Vergleich der Optimierungsverfahren - Funktionsauswertungen

Tabelle 8.1 zeigt Durchschnittswerte, die auf einem Stück Autobahn mit anschließender Ausfahrt ermittelt wurden. Es zeigt sich, dass das BFGS-Verfahren tatsächlich mit der Hälfte der Funktionsauswertungen auskommt. Zwar wird die teurere Operation mit der Berechnung des Gradienten genutzt – in der Laufzeit ergibt sich allerdings trotzdem eine Ersparnis von 2 ms, bzw. ein Speedup von 1,4.

In der Anwendung zeigt sich jedoch, dass die Zielfunktion sich nicht gut für das BFGS-Verfahren eignet. Sie enthält viele lokale Maxima – auch relativ nah am globalen Maximum. Abbildung 8.2 zeigt ein Beispiel mit dem Linienmodell – Datenbasis ist das Bild aus Kapitel 7.

Zu sehen ist, dass schon 4° neben dem globalen Maximum die ersten lokalen Nebenmaxima existieren.



(a) Alle Winkel

(b) Umgebung des Globalen Maximums

Abbildung 8.2: Lokale Nebenmaxima

Dass weniger Informationen bekannt sind, kommt dem Nelder-Mead-Verfahren hier zugute. Das Verfahren approximiert den Gradienten intern und untersucht dabei auch etwas weiter entfernte Parameter – dadurch überspringt es hier die kleinen lokalen Maxima und findet das globale Optimum deutlich robuster.

Im Ausblick (Abschnitt 9.2) wird ein Ansatz vorgeschlagen, wie die Robustheit des BFGS-Verfahrens für diesen Algorithmus verbessert werden könnte.

8.3 Ausführung auf der Zielplattform

Die Zielplattform (Anforderung NF4) des in dieser Arbeit vorgestellten Algorithmus ist der Xavier-SOC von NVIDIA. Die Tests wurden auf der Jetson AGX Xavier Entwicklungsplattform von NVIDIA durchgeführt.

Vergleicht man die Spezifikationen des Xavier-SOC mit dem Entwicklungslaptop (siehe Tabelle 8.2) ist die Grafikeinheit der Xavier-Plattform leistungsfähiger. Sie verfügt über mehr CUDA-Kerne – diese sogar aus einer neueren Generation – und mehr SMs. Der Prozessor ist jedoch deutlich langsamer.

	Laptop (940MX)	Xavier SOC
Generation der Grafikeinheit	Maxwell (5.0)	Volta (7.2)
CUDA-Kerne	384	512
Streaming Multiprocessors	3	8
GPU-Frequenz	1,19 GHz	1,5 GHz
Speicherbandbreite	64 bit	256 bit
Speicher-Frequenz	2505 MHz	1377 MHz
Prozessor	Intel Core i7	ARMv8
Architektur	x86	RISC
Kerne (Hyperthreading)	4 (8)	8 (8)
CPU-Frequenz	3,9 GHz	2,26 GHz

Tabelle 8.2: Vergleich der Spezifikationen (Laptop ↔ Xavier)

Testet man basierend auf der Konfiguration aus Kapitel 7 verschiedene Varianten des Algorithmus (siehe Tabelle 8.3) zeigt sich das Jetson-Board etwa 1,2-mal schneller, als die Laptop-Plattform. Betrachtet man ausschließlich die Laufzeiten der CUDA-Kernel (siehe Tabelle 8.4) wird der Unterschied zwischen der 940MX und der Grafikeinheit des Xavier deutlich. Letztere ist hier etwa 2,7-mal schneller.

	Laptop	Jetson
Linienmodell	0,18 ms	0,15 ms
Linienmodell (Gradient)	0,21 ms	0,16 ms
Quadratisches Modell	0,16 ms	0,15 ms
Quadratisches Modell (Gradient)	0,22 ms	0,18 ms

Tabelle 8.3: Laufzeitvergleich der Funktionsauswertung

	Laptop	Jetson
Linienmodell	75 μs	27 μs
Linienmodell (Gradient)	91 μs	37 μs
Quadratisches Modell	77 μs	26 μs
Quadratisches Modell (Gradient)	107 μs	42 μs

Tabelle 8.4: Laufzeitvergleich der Kernel

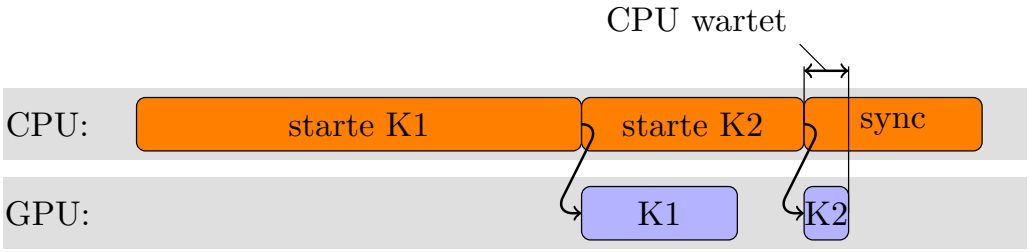
Die Laufzeit der Kernel scheint auf der Xavier-Plattform nur etwa 20% zur Gesamtlaufzeit beizutragen. Um herauszufinden, warum der Anteil so gering ist, wird der NVIDIA Visual Profiler eingesetzt.

In Abbildung 8.3 sind die Ergebnisse dargestellt. Die Laufzeit wird fast ausschließlich vom overhead der CUDA-API-Aufrufe auf der CPU bestimmt, welche unter anderem die Kommunikation zwischen der CPU und der GPU koordinieren. Die Ausführung des ersten Kernels passiert vollständig nebenläufig zur Startvorbereitung des zweiten Kernels (und hat somit keinen Einfluss auf die Gesamtlaufzeit). Lediglich während der Ausführung des zweiten Kernels muss die CPU kurz warten.

Betrachtet man den Ausführungspfad auf dem Laptop (vgl. Abbildung 8.4) zeigt sich ein komplett anderes Bild.
Die Laufzeiten der CUDA-API-Aufrufe sind deutlich kürzer, die Kernel Laufzeiten deutlich länger. Die CPU hat hier schon lange bevor der erste Kernel endet den zweiten vorbereitet und wartet danach nur noch auf die GPU.

Dies bestätigt die Vermutung aus dem Vergleich der Spezifikationen, dass zwar die Grafikeinheit des Xavier-SOC leistungsstärker ist, als die Laptop-GPU, der Prozessor jedoch langsamer.

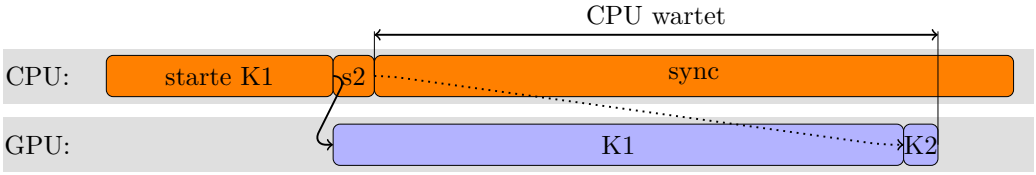
(a) Darstellung im NVIDIA Visual Profiler



(b) Schematische Darstellung

Abbildung 8.3: Ausführungspfad auf dem Jetson-Board

(a) Darstellung im NVIDIA Visual Profiler



(b) Schematische Darstellung

Abbildung 8.4: Ausführungspfad auf dem Laptop

8.3.1 Echtzeitfähigkeit des Gesamtalgorithmus auf der Zielplattform

In Abschnitt 7.1 wurden Laufzeitüberlegungen für den Algorithmus auf dem Laptop durchgeführt. Basierend auf den Ergebnissen aus Abschnitt 8.3 wird erwartet, dass der GPU-beschleunigte Algorithmus aus dieser Arbeit auf der Zielplattform etwas schneller laufen wird. Da die Vorverarbeitung jedoch nur auf der CPU ausgeführt werden kann, wird dafür eine höhere Laufzeit erwartet.

Die Auswertung in Tabelle 8.5 zeigt, dass genau dies der Fall ist. Die dargestellten Laufzeiten wurden als Durchschnittswerte bei der Verarbeitung eines Teilstücks einer Autobahnfahrt ermittelt. Die Laufzeiteinbußen in der Vorverarbeitung und die Laufzeitgewinne im Algorithmus heben sich dabei gegenseitig auf, sodass weiterhin dieselbe Gesamtlaufzeit benötigt wird. Diese bleibt mit 30 ms dabei deutlich unter der in Anforderung NF1 festgelegten Maximallaufzeit von 40 ms.

	Vorverarbeitung	Algorithmus	Gesamt
Laptop	20 ms	10 ms	30 ms
Xavier	23 ms	7 ms	30 ms

Tabelle 8.5: Laufzeitvergleich des Gesamtalgorithmus (gemittelt)

Für die Auswertung wurde als Straßenmodell, wie in Abschnitt 8.1 beschrieben, das Quadratische Model genutzt. Als Optimierungsverfahren wurde, wie in Abschnitt 8.2 beschrieben, das Nelder-Mead Verfahren eingesetzt. Die Diskretisierungsparameter waren weiterhin wie in Abbildung 7.1b.

In Unterabschnitt 7.1.2 wurde eine Begrenzung der Iterationen des Optimierungsverfahrens festgelegt. Aufgrund der verlängerten Laufzeit der Vorverarbeitung von 23 ms verbleiben auf der Xavier-Plattform nur 17 ms für den Algorithmus. Da die Laufzeit der Fouriertransformation weiterhin etwa 4 ms beansprucht verbleiben für die Parametrierung des Straßenmodels 13 ms. Mit der verbesserten Laufzeit einer Funktionsauswertung von 0,15 ms (vgl. Tabelle 8.3) ergibt sich damit eine Begrenzung auf

$$n_{max} = \left\lfloor \frac{13ms}{0,15ms} \right\rfloor = \lfloor 86.67 \rfloor = 86 \tag{8.1}$$

Iterationen – zwei weniger, als auf dem Laptop.

8.4 Typische Szenarien

Die Evaluation einiger typischer Szenarien kann aufgrund fehlender Referenzdaten nur „halb-automatisiert“ erfolgen. Dazu werden manuell feste Erwartungswerte für die Anzahl der Fahrspurmarkierungen sowie die zugehörige Klassifikation festgelegt, welche dann innerhalb des Szenarios automatisiert geprüft werden. Für die Validierung der korrekten Position der Fahrspurmarkierungen fehlt die Datengrundlage – die Breite der Fahrstreifen sollte aber unabhängig von der Position konstant sein und kann leicht ermittelt werden.

Tabelle 8.6 zeigt die Ergebnisse dieser Auswertung.

Szenario	Anzahl korrekt	Klassifizierung korrekt	Fahrspurbreite
Idealbedingungen	100%	100%	± 3,5%
Mitte + Verkehr	83,5%	100%	± 3,5%
Rechts + Verkehr	81%	96%	± 3,1%
Ausfahrt	95,5%	100%	± 16%

Tabelle 8.6: Auswertung verschiedener Szenarien

Im Szenario „Idealbedingungen“ fährt das Fahrzeug auf der Mittelspur einer dreispurigen Autobahn. Die Fahrspurmarkierungen sind neu und gut sichtbar, außerdem ist kein Verkehr. Unter diesen Bedingungen kann der Algorithmus alle Markierungen korrekt erkennen und klassifizieren. Die Abweichung in der Fahrspurbreite entspricht etwa der Auflösung von 0,2m.

Auch im Szenario „Mitte + Verkehr“ fährt das Fahrzeug auf der Mittelspur einer dreispurigen Autobahn. Nun jedoch mit Verkehr. Welche Fahrspur gerade sichtbar ist wird in der vereinfachten Evaluation nicht unterstützt. Die verminderte Erkennungsrate von 83,5% ergibt sich daher aus der zeitweisen Verdeckung der Fahrspuren – beispielsweise wenn ein LKW überholt wird.

Im Szenario „Rechts + Verkehr“ fährt das Fahrzeug auf der rechten Spur einer dreispurigen Autobahn. Auch hier mit Verkehr. Dieser besteht jedoch nur aus PKWs, sodass jederzeit genug Fahrspurmarkierung sichtbar ist. Die verminderte Erkennungsrate wird hier durch das Versagen der Vorverarbeitung verursacht.

Zudem wird die linke Außenmarkierung in diesem Szenario nicht erkannt. Die Intensität der Messungen ist aufgrund des Abstandes zu niedrig und die Markierung geht im Rauschen unter. Da es sich bei der Datenquelle um einen Sensorprototypen handelt ist hier jedoch noch Besserung zu erwarten.

Im letzten Szenario „Ausfahrt“ fährt das Fahrzeug von der Autobahn ab. Die verminderte Erkennungsrate wird hier durch eine S-Kurve verursacht, welche nicht über das Straßenmodell abgebildet werden kann. Beim Durchfahren dieser scheitert in einigen Fällen die Plausibilitätsprüfung. Die Hohe Abweichung in der Fahrspurbreite wird durch einige Ausreißer im Bereich der S-Kurve verursacht.

9 Fazit

In diesem Kapitel werden die Ergebnisse der Arbeit zusammengefasst und im Ausblick wird auf Weiterentwicklungsmöglichkeiten eingegangen.

9.1 Zusammenfassung

In dieser Arbeit wurde gezeigt, wie eine Fahrspurerkennung auf LiDAR-Basis beschleunigt werden kann, sodass diese in Echtzeit auf einem eingebetteten System lauffähig ist. Dabei wurde darauf geachtet, den Algorithmus leicht erweiterbar zu gestalten. Insbesondere das Straßenmodell kann einfach ausgetauscht werden und auch neue Straßenmodelle können mit wenig Aufwand hinzugefügt werden.

Das zugrundeliegende Verfahren wurde um die Berechnung des Gradienten der Zielfunktion erweitert. Dies ermöglicht den Einsatz leistungsfähigerer, ableitungsbasierter Optimierungsverfahren.

In der Auswertung der Ausgabe zeigte sich unter Idealbedingungen ein perfektes Ergebnis. Zwar beeinflussen Verkehr, durch das Straßenmodell nicht abbildbare Straßenformen (S-Kurve in der Abfahrt) und Fehler in der Vorverarbeitung das Ergebnis des Algorithmus. Dieser liefert aber nach Ende der Störeinflüsse robust wieder die korrekten Ergebnisse.

Neben der Berechnung des Gradienten war die Optimierung der Laufzeit ein zentrales Thema dieser Arbeit. Das Resultat ist ein CUDA-Kernel, welcher die Rechen- und Speicherressourcen auf der GPU in der Waage hält und in der Lage ist, diese gut auszunutzen. Dadurch konnte die Laufzeit der Funktionsauswertung um den Faktor 60 verringert werden. Der finale Algorithmus benötigt im Mittel nur 7 ms und wird zukünftig in der Perception-Pipeline der Firma eingesetzt.

9.2 Ausblick

Im Folgenden soll ein Ausblick auf mögliche Weiterentwicklungen des Algorithmus gegeben werden – auch im Hinblick auf in Zukunft erwartete, größere Mengen von Rohdaten.

9.2.1 Optimierung der Vorverarbeitung

Die Vorverarbeitung, welche zur Zeit lediglich als Prototyp vorliegt, wurde in dieser Arbeit als gegeben hingenommen und nicht weiter betrachtet.

In Abschnitt 6.6 zeigten sich allerdings abschnittsweise gravierende Probleme in der Vorverarbeitung. Teilweise entfernte diese klar sichtbare Markierungen direkt neben dem Fahrzeug. Da es sich bei der Vorverarbeitung um einen Prototypen handelt, ist dieses Verhalten nicht überraschend – im aktuellen Stadium ist jedoch fast immer eine fehlerhafte Vorverarbeitung für nicht erkannte Fahrspurmarkierungen verantwortlich.

In der Auswertung der Laufzeit (Abschnitt 7.1 und Unterabschnitt 8.3.1) zeigte sich außerdem, dass die Vorverarbeitung den größten Anteil beansprucht. Insbesondere im Hinblick auf zukünftig größere Datenmengen scheint eine Parallelisierung – möglicherweise auch wieder GPU-beschleunigt – sinnvoll.

9.2.2 Verbesserungen des Algorithmus

In Abschnitt 8.2 verhinderten lokale Maxima der Zielfunktion einen erfolgreichen Einsatz des BFGS-Verfahrens. Ein möglicher Ansatz, diese zu vermeiden, ist der Multilevel-Ansatz [15].

Beim Multilevel-Ansatz wird die Bildauflösung zunächst stark verringert und das Bild zudem geglättet. Dadurch wird auch die Zielfunktion geglättet, sodass diese im Idealfall nur noch ein Maximum enthält. Dieses ist mit einem auf Gradienten basierenden Optimierungsverfahren sehr schnell zu finden. Weil dazu üblicherweise eine deutlich verringerte Auflösung benötigt wird, leidet im Gegenzug die Genauigkeit. Um diese wieder zu erhöhen wird nach der Optimierung in der niedrigsten Auflösung diese schrittweise wieder erhöht, bis die ursprüngliche Auflösung erreicht ist. In jedem Schritt findet eine

weitere Optimierung statt, deren Startwerte die Ergebnisse aus der vorherigen Auflösungsstufe sind. Diese liegen schon nah am tatsächlichen Optimum – idealerweise näher als das nächste lokale Maximum.

Ein sinnvoller nächster Schritt wäre daher die Implementierung eines solchen Multilevel-Ansatzes.

Um weitere Optimierungen auswerten zu können sollte zudem eine automatisierte Validierung über Referenzdaten angestrebt werden. Diese sollte außerdem Key Performance Indicators (KPIs) ermitteln, um verschiedene Versionen bzgl. ihrer Güte vergleichen zu können.

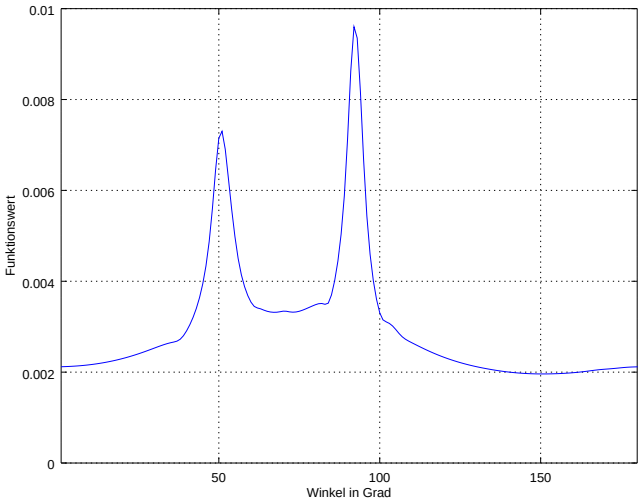
9.2.3 Erweiterung des Algorithmus

Eine fundamentale Einschränkung des Algorithmus ist die Grundannahme, dass alle Fahrspuren parallel verlaufen (siehe Abschnitt 4.3). In Abbildung 9.1 ist der Verlauf des Funktionswerts der Winkel von $0 - 180°$ für eine theoretisch konstruierte Gabelung dargestellt. Es sind deutlich zwei Maxima zu sehen – eines für die Hauptstraße sowie ein zweites kleineres für die abzweigende Straße.

In der Realität existiert eine solche Gabelung mit scharfen Knicken natürlich nicht, sondern die Straße würde sanft abzweigen. Dabei müsste sich langsam ein Nebenmaximum für die zweite Straße herausbilden. Prinzipiell sollte es möglich sein, dieses zu erkennen – und damit die Existenz einer Gabelung.

Eine weitere Möglichkeit der Erweiterung ist die Fusion mit Kameradaten [6, 7]. Neben einer Erhöhung der Robustheit gewinnt man dadurch auch Farbinformationen, um beispielsweise in Baustellen zwischen weißen und gelben Markierungen unterscheiden zu können.

(a) Konstruierte Gabelung



(b) Verlauf der Funktionswerte

Abbildung 9.1: Veranschaulichung der Zielfunktion bei mehreren Lösungen

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während der Anfertigung dieser Bachelorarbeit unterstützt und motiviert haben.

Zuerst gebührt mein Dank Florian und Jan, die sich stets meinen Fragen stellen mussten. Eure wertvollen Anregungen und die konstruktive Kritik haben wesentliche Teile dieser Arbeit geprägt.

Ich bedanke mich bei Herrn Prof. Dr. Korf für die Unterstützung bei der Themenfindung, die interessanten Diskussionen und hilfreichen Anmerkungen. Auch dafür, dass Sie mich von Anfang an stark gefordert haben möchten ich und insbesondere mein Zeitmanagement uns bedanken.

Ebenfalls bedanken möchte ich mich bei der Ibeo Automotive Systems GmbH für die Bereitstellung dieses interessanten Themas. Vielen Dank an Ruben und alle Kollegen für die herzliche Aufnahme ins Team und die Unterstützung bei allen Fragen.

Abschließend gilt mein besonderer Dank meiner Familie, insbesondere meinen Eltern, die mir mein Studium ermöglicht und mich in all meinen Entscheidungen unterstützt haben.

Literaturverzeichnis

[1] Bachmann, E.: Die Klothoide als Übergangskurve im Straßenbau. In: *Schweizerische Zeitschrift für Vermessung, Kulturtechnik und Photogrammetrie* 49 (1951), 6, Nr. 6, S. 133–140

[2] Brammer, Karl ; Siffling, Gerhard: *Kalman-Bucy-Filter : deterministische Beobachtung und stochastische Filterung*. 1994. – ISBN 978-3-486-22779-6

[3] Bundesministerium für Verkehr, Bau- und Wohnungswesen: *Richtlinien für die Sicherung von Arbeitsstellen an Straßen*. Februar 1995

[4] Drews, Lukas: *Private Kommunikation*. 2019

[5] Forschungsgesellschaft für Strassen- und Verkehrswesen e.V. (FGSV), Köln - Arbeitsgruppe Verkehrsführung und Verkehrssicherheit: *Richtlinien für die Markierung von Straßen*. September 1993

[6] Gu, Xiaodong ; Zang, Andi ; Huang, Xinyu ; Tokuta, Alade ; Chen, Xin: Fusion of Color Images and LiDAR Data for Lane Classification. In: *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*. New York, NY, USA : ACM, 2015 (SIGSPATIAL '15), S. 69:1–69:4. – URL http://doi.acm.org/10.1145/2820783.2820859. – ISBN 978-1-4503-3967-4

[7] Homm, F. ; Kaempchen, N. ; Burschka, D.: Fusion of laserscannner and video based lanemarking detection for robust lateral vehicle control and lane change maneuvers. In: *2011 IEEE Intelligent Vehicles Symposium (IV)*, Juni 2011, S. 969–974. – ISSN 1931-0587

[8] Jähne, Bernd: *Digitale Bildverarbeitung*. 2002. – ISBN 3540412603

[9] Königsberger, Konrad: *Analysis - Teil 2*. 2002. – ISBN 978-3-540-43580-8

[10] Low, C. Y. ; Zamzuri, H. ; Mazlan, S. A.: Simple robust road lane detection algorithm. In: *2014 5th International Conference on Intelligent and Advanced Systems (ICIAS)*, Juni 2014, S. 1–4

[11] Nocedal, George ; Wright, S.: *Numerical Optimization*. 2006 ISSN 1431-8598

[12] NVIDIA: *Programming Guide :: CUDA Toolkit Documentation*. – URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#device-memory-accesses

[13] Ogawa, T. ; Takagi, K.: Lane Recognition Using On-vehicle LIDAR. In: *2006 IEEE Intelligent Vehicles Symposium*, Juni 2006, S. 540–545. – ISSN 1931-0587

[14] Rühaak, Jan ; König, Lars ; Tramnitzke, Florian ; Köstler, Harald ; Modersitzki, Jan: A Matrix-Free Approach to Efficient Affine-Linear Image Registration on CPU and GPU. In: *Journal of Real-Time Image Processing* 13 (2017), Nr. 1, S. 205–225. – URL http://dx.doi.org/10.1007/s11554-016-0564-4. – ISSN 1861-8219

[15] Rühaak, Dr. J.: *Private Kommunikation*. 2019

[16] Shin, Seunghak ; Shim, Inwook ; Kweon, In S.: Combinatorial approach for lane detection using image and LIDAR reflectance. In: *2015 12th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, Oktober 2015, S. 485–487

[17] Tian, M. ; Liu, F. ; Zhu, W. ; Xu, C.: Vision Based Lane detection for Active Security in Intelligent Vehicle. In: *2006 IEEE International Conference on Vehicular Electronics and Safety*, Dec 2006, S. 507–511

[18] von Reyher, A. ; Joos, A. ; Winner, H.: A lidar-based approach for near range lane detection. In: *IEEE Proceedings. Intelligent Vehicles Symposium, 2005.*, Juni 2005, S. 147–152. – ISSN 1931-0587

[19] Web: *CUDA Zone | NVIDIA Developer*. – URL https://developer.nvidia.com/cuda-zone

[20] Web: *cuFFT::CUDA Toolkit Documentation*. – URL https://docs.nvidia.com/cuda/cufft/index.html

[21] Web: *Katalog der Deutschen Nationalbibliothek*. – URL http://d-nb.info/gnd/4464685-9

[22] Web: *OpenCL Overview*. – URL https://www.khronos.org/opencl/

[23] Web: *Die häufigsten Unfallursachen | Runter vom Gas*. 2019. – URL https://www.runtervomgas.de/unfallursachen/artikel/die-haeufigsten-unfallursachen.html

[24] Zehntner GmbH: *Grundlagen Retroreflexion RL (Nachtsichtbarkeit) von Strassenmarkierungen erzeugt durch Glasperlen*. Juli 2017. – URL http://www.zehntner.com/download/prospekt_retroreflektion_glasperlen_d.pdf

Bildquellen

[25] NVIDIA: *Figure 3. Hardware Model.* – URL https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#set-of-simt-multiprocessors

[26] NVIDIA: *Figure 6. Grid of Thread Blocks.* – URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#thread-hierarchy

[27] ReasearchGate: *TEDUSAR White Book - State of the Art in Search and Rescue Robots - Scientific Figure on ResearchGate.*. – URL https://www.researchgate.net/figure/3D-laser-scan-left-taken-with-a-HDL-64E-LiDAR-from-Velodyne-right-Photo-credit-IST_fig14_304987927

[28] Web: *Spatial Locality.* – URL http://cuda-programming.blogspot.com/2013/02/texture-memory-in-cuda-what-is-texture.html

[29] Wikimedia Commons: *LIDAR-scanned-SICK-LMS-animation.* – URL https://commons.wikimedia.org/wiki/File:LIDAR-scanned-SICK-LMS-animation.gif

Glossar

Cache Ein Cache ist ein schneller Pufferspeicher, welcher häufig verwendete Werte aus langsamen Quellen abruft und für zukünftige Berechnungen schneller zur Verfügung stellt.

Framework Ein Framework ist *eine Menge wiederverwendbarer Klassen, zusammen mit einer Systemarchitektur zur Erstellung von Anwendungen für ein Gebiet* [21].

Konfidenz Eine Konfidenz ist ein einheitenloser Wert, welcher bezeichnet, wie sicher sich der Algorithmus ist, ein korrektes Ergebnis geliefert zu haben.

LiDAR engl. **Li**ght **D**etection **A**nd **R**anging – ein Sensor, der Entfernungen mithilfe von Laserpulsen misst.

Präprozessor Der Präprozessor führt eine Vorverarbeitung der Dateien durch, bevor diese dem Compiler zur Übersetzung in Maschinencode weitergegeben werden. Der Präprozessor kann verschiedene textuelle Ersetzungen durchführen..

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Effiziente GPU-basierte Klassifizierung von Fahrspuren auf eingebetteten Echtzeitsystemen

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original