Bachelorarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Künstliche Intelligenz und Robotik

Optimierung von Algorithmen für dichtes Stereomatching

Sven Adfeldt

s.adfeldt@fu-berlin.de

Erstkorrektur: Prof. Dr. Raúl Rojas

Betreuer: Robert Spangenberg

Berlin, 21. Oktober 2013

Zusammenfassung

Stereo Vision ist eine passive Technologie, die es einem autonomen System ermöglicht, die Entfernung von Objekten in seiner Umgebung zu ermitteln. Das dichte Stereomatching erstellt dabei eine komplette Abstandskarte im Sichtfeld der Stereo Kameras, die dann anschließend für viele Anwendungen verarbeitet werden kann.
Das größte Problem des Verfahrens ist der hohe Rechenaufwand für sehr gute Ergebnisse, weswegen die meisten Lösungen auf einer GPU oder einem FPGA laufen. Es ist allerdings wünschenswert auch auf einem normalen Prozessor solche Verfahren mit ausreichender Qualität anwenden zu können.
In dieser Arbeit wird versucht zwei verbreitete Verfahren, Blockmatching und Semi Global Matching, auf einer CPU so zu implementieren, dass die Berechnungszeit ausreichend schnell für Anwendungen des Automotive Bereichs wie Fahrassistenzsysteme ist.
Dazu werden algorithmische Optimierungen, sowie SIMD Befehle auf einer Intel CPU (SSE) und Multithreading zur Parallelisierung auf Mehrkernsystemen verwendet.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, 28. Oktober 2013

. . . .

Sven Adfeldt

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation 1
1.2 Struktur der Arbeit 1
1.3 Aufgabenstellung 1
1.4 Kamerasystem 2
1.5 Verwandte Arbeiten 2

2 Dichtes Stereomatching **4**
2.1 Stereo Vision 4
2.2 Census Transformation 5
2.3 Blockmatching 6
2.4 Semi Global Matching 7
2.5 Verwendung der Kostenfunktion 8
2.6 Rechts-Links-Check 8
2.7 Üblicher Gesamtablauf mit Optimierungsschritten 9

3 Implementierung **10**
3.1 Kerntechnologie 10
3.2 Cassandra 10
3.3 Intel SSE 10
3.4 Multithreading 11
3.5 Blockmatching 11
3.6 Semi Global Matching 13

4 Ergebnisse **16**
4.1 Testkonfiguration 16
4.2 Blockmatching 16
4.3 Semi Global Matching 17

5 Diskussion und Ausblick **23**

Abbildungsverzeichnis **26**

Literatur **27**

1 Einleitung

1.1 Motivation

Die optische 3D-Wahrnehmung der Umgebung spielt eine entscheidende Rolle in der Robotik, da diese Informationen für die Interaktion eines eigenständigen Systems mit der Umwelt notwendig sind.
Dichtes Stereomatching ist ein Verfahren zur Gewinnung von Tiefeninformationen aus zwei parallelen Kamerabildern des gleichen Zeitpunkts. Dabei besitzen Stereokameras als eine passive Technologie – im Gegensatz zu aktiven wie LiDAR-Systemen (Light Detection and Ranging, benutzt aktiven Laser) oder Radar – den Vorteil, dass sie sehr kostengünstig und energiesparend sind [1]. Das größte Problem des Stereomatchings ist, dass es sehr rechenaufwändig ist, wenn man ein sehr gutes Ergebnis erzielen möchte. Deswegen werden oft GPU- (Graphics Processing Unit, Grafikkarte) oder FPGA- (Field Programmable Gate Array, ein integrierter Schaltkreis) Implementierungen für die Algorithmen gewählt. Jedoch verbrauchen GPUs eine Menge Energie und FPGAs lassen sich nicht so einfach in Laptop-Systeme integrieren [2]. Somit ist es wünschenswert auch auf einer CPU eine Implementierung zu finden, die ausreichend schnell und genau für Anwendungen wie Fahrassistenzsysteme ist. Dazu wäre eine Bildrate von wenigstens 10Hz notwendig.

1.2 Struktur der Arbeit

Diese Arbeit beginnt mit der Einführung in die Aufgabenstellung, stellt das Kamerasystem und verwandte Arbeiten vor. Anschließend werden die Grundlagen der Algorithmen und Verfahren behandelt. Danach werden die wichtigsten Implementierungsdetails und Technologien zusammengefasst. Zum Schluss folgt eine Zusammenstellung, sowie Diskussion und Einordnung der Ergebnisse.

1.3 Aufgabenstellung

Ziel des dichten Stereomatchings ist es, die Entfernung von Objekten zum Beobachter zu bestimmen. Verwendet man dazu ein Stereo-Kamerasystem, so ist dies dadurch möglich, dass man von der beobachteten Szene zwei Bilder besitzt, die lediglich um den Abstand der Linsen zueinander verschoben sind. Rechnet man diese Verschiebung heraus, so befinden sich die Objekte in beiden Bildern dennoch an unterschiedlichen Positionen, da das Licht in unterschiedlichen Winkeln eingefallen ist. Dabei ist die horizontale Verschiebung eines Objekts im rechten im Vergleich zum linken Bild proportional zu seiner Entfernung. Je näher das Objekt desto weiter verschoben ist es im Bild.
Es reicht also aus ein Objekt im linken und rechten Bild zu finden und seine Verschiebung zu bestimmen. Daraus lässt sich dann die Entfernung vom Beobachter mit Hilfe von Triangulierung bestimmen.

1.4 Kamerasystem

Die Arbeit findet im Rahmen des Projekts AUTONOMOS der FU Berlin statt.
Das aktuelle autonome Fahrzeug des Projektes *MadeInGermany* (siehe Abb. 1) besitzt neben vielen verschiedenen Sensoren auch ein Stereokamerasystem, das sich an der Windschutzscheibe befindet (siehe Abb. 2). Dieses System wurde verwendet, um Teile der Videoaufnahmen zu machen, die in dieser Arbeit zum Testen der implementierten Algorithmen verwendet werden.
Die Verfahren funktionieren aber auch mit anderen Stereokamerasystemen.

Abb. 1: Autonomes Fahrzeug MadeInGermany des Projekts AUTONOMOS der FU Berlin

1.5 Verwandte Arbeiten

In einigen Arbeiten wurden bereits Dichte Stereo Matching Algorithmen auf einer CPU implementiert, die eine hohe Qualität bei Laufzeiten von mehr als 10fps liefern.
Zinner et al. entwickelten mit S^3E eine Implementierung des Blockmatchings mit Census Transformation, die mit Hilfe von SIMD Operationen (Single Instruction Multiple Data) optimiert wurde [3]. Dabei konnten sie Laufzeiten von 68ms pro Bild (bei 435 $\times$ 50 Pixeln) auf einem Intel Core 2 Duo mit 2GHz erreichen.
Bodkin [4] entwickelte ebenfalls eine optimierte Implementierung des Blockmatchigs für gängige CPUs, die SIMD Instruktionen, einen Sliding Window Ansatz und Parallelisierung verwendet. Damit erreichte er auf einem 384 $\times$ 288 Bild mit 16 Disparitäten eine durchschnittliche Laufzeit von 14,3ms pro Bild auf einem Intel Core I7-720Q.

Abb. 2: Stereo Kamerasystem an der Windschutzscheibe von MadeInGermany

Es wurde aber auch versucht Semi Global Matching auf einer CPU mit möglichst hoher Bildrate zu realisieren. Humenberger et al. implementierten eine Variante das SGM, die nicht das gesamte Bild, sondern nur horizontale Streifen verwendet, womit sie den benötigten Speicher reduzieren konnten, was das Verfahren auch für Eingebettete Systeme attraktiv macht. Allerdings verwendeten sie keine weiteren Optimierungen und erreichten somit nur Zeiten weit über 10fps [1].
Dagegen entwickelten Gehrig und Rabe eine SGM Implementierung, die es mit Hilfe von Parallelisierung, SIMD Operationen und Image Subsampling (für nahe Objekte wurde eine geringere Auflösung verwendet) auf mehr als 14fps bei 640×320 Pixeln schafft. Dabei wird in mehreren Stufen für große Disparitäten das Bild nur mit halber Auflösung betrachtet, um die Laufzeit zu reduzieren. Dabei zeigten sie, dass die Genauigkeit für die Tiefenkomponente damit über das gesamte Bild einen Grenzwert nicht unterschreitet [2].

2 Dichtes Stereomatching

2.1 Stereo Vision

Stereo Vision ist ein wichtiger Forschungsbereich in der digitalen Bildverarbeitung, bei dem Tiefeninformationen durch die Verwendung von zwei oder mehreren Kameras gewonnen werden soll. Werden genau zwei Kameras verwendet, so handelt es sich um *Binoculare Stereo Vision*, die dem menschlichen Sehapparat sehr ähnlich ist. Dieser Fall wird in dieser Arbeit betrachtet. Es werden also zwei Kameras verwendet um Bildpaare derselben Szene zu bekommen. Eines der beiden Bilder eines solchen Paares wird dann als Ziel- und das andere als Referenzbild bezeichnet.
Beim *dichten Stereomatching* gilt es für jeden Pixel im Zielbild den entsprechenden Pixel des Referenzbildes zu finden. Anschließend ist es möglich für jeden Pixel auf Grund von Triangulierung die Tiefe zu bestimmen.
Abbildung 3 (siehe Abb. 3) zeigt dieses Prinzip anhand zweier Punkte P und Q, die sich für das eine Bild auf einer Sichtlinie (rote Linie) befinden. Beide Punkte befinden sich dann aufgrund der epipolaren Bedingung ebenfalls auf einer Linie in dem anderen Bild.

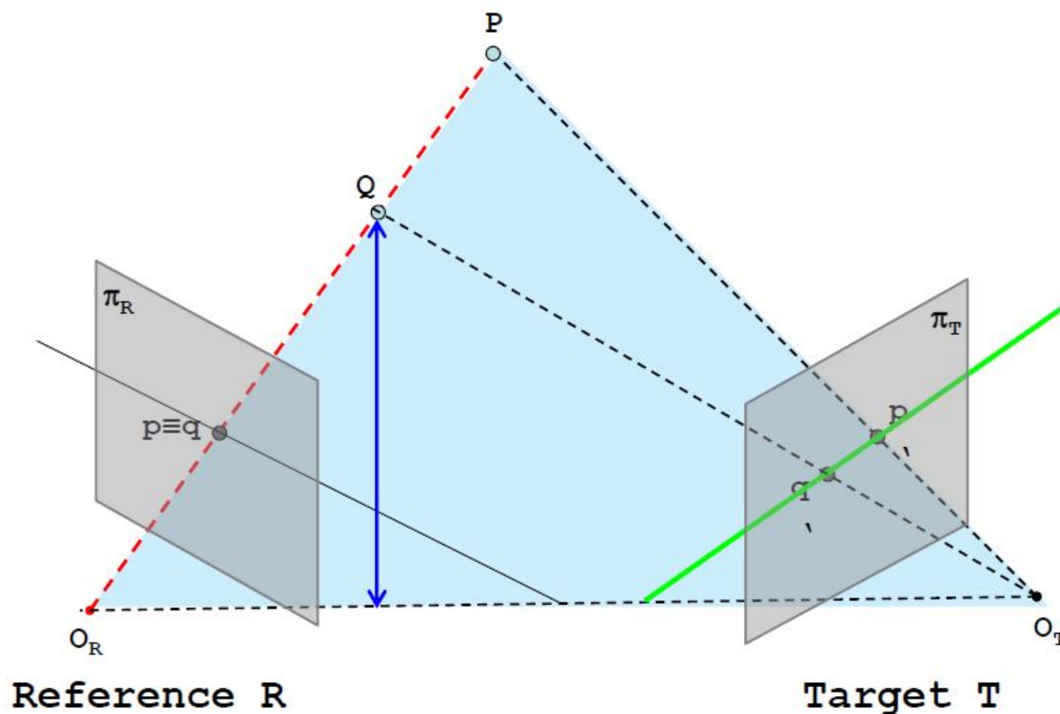


Abb. 3: Triangulierung von zwei Punkten P und Q ausgehend von zwei verschiedenen Betrachtungswinkeln [5]

Daraus ergibt sich nun, dass ein Pixel des Zielbildes nicht im kompletten 2D Referenzbild gesucht werden muss, sondern sich dort auf einem 1D Strahl befindet.
Die Standardform der Bildpaare erhält man, wenn die Kameras parallel angeordnet werden. In diesem Fall befinden sich die epipolaren Linien horizontal im Bild, was besonders günstig für die Bildverarbeitung ist. Im Normalfall erhält man diese Form durch eine Transformation der Bilder, die sogenannte *Rektifizierung*.
Beim dichten Stereomatching wird somit für jeden Pixel des Zielbildes der entsprechende Pixel im Referenzbild in horizontaler Richtung gesucht und der Abstand ermittelt, den man als Disparität bezeichnet. Dabei ist Disparität 0 die Stelle im Referenzbild, wo sich der Pixel befindet, wenn er (unendlich) weit weg ist, also im Referenzbild nicht verschoben ist. Der Abstand wird von dort aus in Pixeln angegeben und für jeden Pixel des Zielbildes abgespeichert. Das Ergebnis mit jeder Disparität für jeden Pixel wird als *Disparitätsbild* (Disparity Map) bezeichnet. Aus ihr kann dann mit Hilfe von Triangulation der Abstand von jedem Pixel zum Beobachter berechnet werden.

2.2 Census Transformation

Damit die Pixel des Zielbildes im Referenzbild gefunden werden können, werden bei einer Transformation lokale Informationen aus der Umgebung des Pixels verwendet, um daraus einen möglichst charakteristischen Wert zu bestimmen, den man dann auf der Epipolarlinie suchen kann.
Die *Census Transformation* wurde von Zabih und Woodfill [6] eingeführt. Dabei handelt es sich um eine lokale Transformation, die die Helligkeitsunterschiede eines Pixels in seiner Umgebung verwendet, um eine Darstellung für diesen Pixel zu ermitteln (siehe Abb. 4). Anschließend wird der Hammingabstand zwischen zwei Census-transformierten Pixeln bestimmt, um ihre Ähnlichkeit zu ermitteln. Der Hamming Abstand ist für zwei Bitstrings u und v der Länge n wie folgt definiert:

$$Hamming(u,v) := |\{i \in \{1...n\} | u_i \neq v_i\}| \tag{1}$$

Diese Methode hat in der Anwendung sehr gute Ergebnisse auch im Vergleich mit anderen lokalen Kostenfunktionen wie *Sum of Absolute Differences* (SAD) oder *Sum of Squared Differences* (SSD) erzielt. Sie ist vor allem bei real belichteten Szenen wesentlich stabiler als die beiden anderen gängigen Verfahren [7]. Ein anderes stabiles Verfahren, Mutual Information, vorgestellt von Hirschmüller [8] ist dagegen anfällig gegenüber Vignettierungsartefakten [9].

Konkret verwendet die Census Transformation ein $(n \times m)$ Feld um einen Pixel, um ihm wie folgt einen Bitstring zuzuweisen:

$$T(u,v) := \bigotimes_{i=-n'}^{n'} \bigotimes_{j=-m'}^{m'} \xi(I(u,v), I(u+i, v+j)), \tag{2}$$

wobei $I(u,v)$ die Intensität des Pixels (u,v), $n' := \lfloor \frac{n}{2} \rfloor$, $m' := \lfloor \frac{m}{2} \rfloor$ und $\bigotimes$ die bitweise

Konkatenation bezeichnen. Die Funktion ξ ist definiert als:

$$\xi(x, y) := \begin{cases} 0 & \text{für } x \leq y \\ 1 & \text{für } x > y \end{cases} \tag{3}$$

Die Kosten C sind dann definiert als

$$C(u, v, d) := Hamming(T_r(u, v), T_l(u + d, v)) \tag{4}$$

Dabei bezeichnen $T_l(u, v)$ und $T_r(u, v)$ den linken bzw. rechten Bitstring des Pixels (u, v) und d die Disparität.

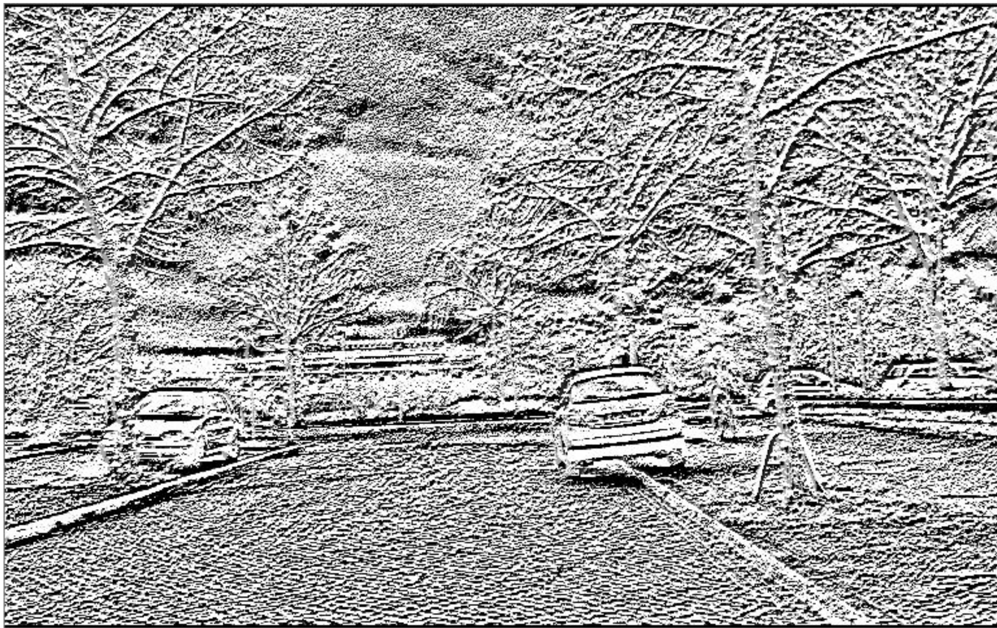


Abb. 4: Census-transformiertes Bild in Graustufen dargestellt, aus Video 1 (siehe Abschnitt 4.1)

2.3 Blockmatching

Das *Blockmatching* verwendet eine Kostenfunktion und führt mit dieser eine *lokale Aggregation* zur Bestimmung der Kosten durch. Dabei wird in dieser Arbeit der Hamming-Abstand zwischen allen Pixeln, die sich an der gleichen Stelle in einem $(n \times m)$ Block um die beiden Pixel im Ziel- und Referenzbild befinden, berechnet. Die Gesamtkosten ergeben sich dann aus der Summe:

$$C'(u, v, d) := \sum_{i=-n'}^{n'} \sum_{j=-m'}^{m'} Hamming(T_r(u + i, v + j), T_l(u + d + i, v + j)), \tag{5}$$

mit $n' := \lfloor \frac{n}{2} \rfloor$, $m' := \lfloor \frac{m}{2} \rfloor$ und $T_r(u,v)$ und $T_l(u,v)$ als Bitstrings des Pixels (u,v) im rechten bzw. linken Bild. Die lokale Aggregation ist jedoch unabhängig von den verwendeten Kosten und könnte auch mit SAD, SSD oder anderen Kosten verwendet werden.
Durch diese Summierung der Kosten aus der Umgebung der Pixel lassen sich bessere Ergebnisse erzielen als unter ausschließlicher Verwendung der Pixel selbst, da sich dadurch in Bereichen mit wenig Textur eindeutigere Treffer erzielen lassen. In solchen Bereichen werden die Ergebnisse besser, je größer der $(n \times m)$ Block ist.
Allerdings führen zu große Blöcke vor allem an Kanten zu Problemen, da diese unscharf und weniger exakt verlaufen. Denn bei Pixeln, die sich auf einer Seite einer Kante befinden, liegt immer noch ein großer Teil des Blocks auf der anderen Seite der Kante und fließt somit auch in die Kosten ein. Dies lässt sich in Abbildung 5 (siehe Abb. 5) beobachten. Im ersten Fall enthält der Block nur Pixel, die sich im gleichen Abstand zum Beobachter befinden, nimmt man aber nun einen größeren Block, so wird das Ergebnis nicht unbedingt besser, da nun auch ein großer Teil des Blocks ein Objekt enthält, das einen kleineren Abstand zum Beobachter hat. Aus diesem Grund verwendet man eine Blockgröße, die nicht zu groß aber auch nicht zu klein ist (meist etwa 5×5 oder 5×7).

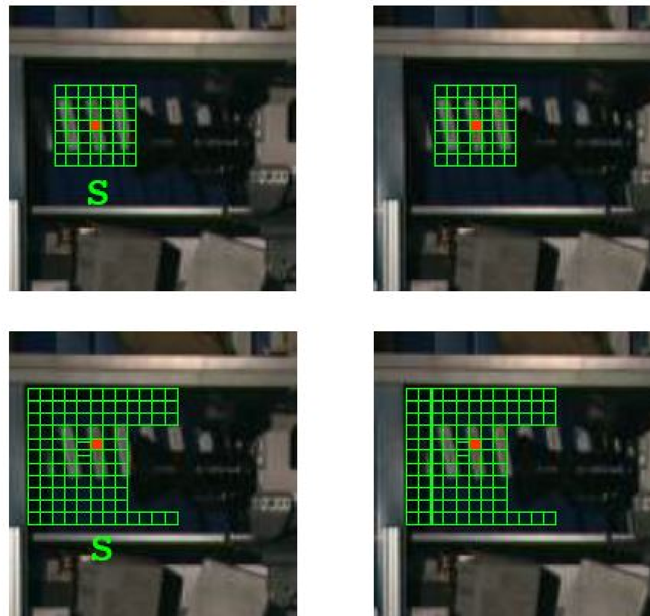


Abb. 5: Probleme mit zu großen Blöcken: Sind sie zu groß überdecken sie oft Bereiche mit verschiedenen Abständen zum Beobachter [5]

2.4 Semi Global Matching

Wenn man sehr nahe an das optimale Ergebnis (sog. *Ground Truth*) beim dichten Stereo Matching herankommen möchte, so reicht es nicht aus sich auf lokale Informationen zu beschränken. Man benötigt also zusätzlich globale Informationen, wenn man die Kosten

zwischen zwei Pixeln berechnet. Das Problem bei solchen globalen Verfahren (z.B. Belief Propagation [10] oder Graph-Cuts [11]) ist meist ein noch größerer Rechenaufwand, mit denen das Ziel von 10 fps nur schwer erreichbar wäre.
Semi Global Matching wurde als erstes von Hirschmueller vorgestellt [8]. Es versucht eine 2D Glättebedingung durch mehrere 1D Bedingungen anzunähern. Dabei zeichnet sich das Verfahren dadurch aus, dass es ein besonders gutes Verhältnis zwischen Laufzeit und Qualität des Ergebnisses erreicht. Außerdem konnte es bereits seine Robustheit und Stabilität für Fahrassistenzsysteme zeigen [12].
Die Technik minimiert die globalen Kosten in horizontaler, vertikaler und in den diagonalen Richtungen. Wobei 8 oder 16 Pfade verwendet werden können und jeder Kostenpfad $L_r(p,d)$ für einen Pixel $p := (u,v)$ mit Disparität d in Richtung r rekursiv berechnet wird:

$$\begin{aligned} L_r(p,d) := C(p,d) + min(&L_r(p-r,d), L_r(p-r,d-1)+P_1, \\ &L_r(p-r,d+1)+P_1, \min_{k\in D} L_r(p-r,k)+P_2), \end{aligned} \tag{6}$$

wobei P_1 und P_2 Strafen sind, die addiert werden, wenn sich die Disparitäten um 1 bzw. mehr als 1 ändern ($P_1 < P_2$). D ist die Menge aller möglichen Disparitäten. Siehe auch Abbildung 7 auf Seite 14. Dort sind die Richtungen für 16 Pfade für einen Pixel dargestellt.
Die Gesamtkosten S für einen Pixel p ergeben sich dann als Summe über alle Pfade in alle Richtungen r:

$$S(p,d) := \sum_r L_r(p,d) \tag{7}$$

2.5 Verwendung der Kostenfunktion

In allen drei vorgestellten Varianten wird eine Kostenfunktion definiert. Diese wird verwendet, um in der epipolaren Linie im Referenzbild für einen Pixel des Zielbildes den entsprechenden Pixel zu finden. Dabei wird der Pixel mit den minimalen Kosten als zugehöriger Pixel angenommen (*Winner Takes All*, WTA). Zusätzlich wird der beste noch mit dem zweitbesten Pixel verglichen. Dabei wird der beste Pixel als ungültig erklärt (ihm wird als Wert *undefiniert* zugewiesen), wenn sein Minimum nicht eindeutig genug ist (sog. *Consistency Check*). Somit werden Teile des Bildes invalidiert um Fehler zu vermeiden, da dadurch Werte entfernt werden, die mit einer großen Wahrscheinlichkeit nicht korrekt sind.

2.6 Rechts-Links-Check

Ein Problem beim Stereo Vision sind Bereiche, die für die eine Kamera verdeckt, für die andere aber sichtbar sind. Denn dort kann die Disparität nicht korrekt bestimmt werden. Diese Bereiche treten meist in der Nähe von Objekten auf, da sie für die beiden Kameras unterschiedliche Bereiche verdecken, sofern sie sich nicht sehr weit entfernt befinden. Aus diesem Grund sollten solche Bereiche erkannt und alle enthaltenen Pixel als

ungültig erklärt werden. Ein Verfahren dafür ist der Rechts-Links-Check. Dabei werden die Bilder beider Kameras jeweils einmal als Referenz- und einmal als Zielbild verwendet und für sie die Disparitäten bestimmt. Anschließend werden die Werte für jeden Pixel verglichen. Stimmen sie nicht (mit einer gewissen Toleranz) überein, so wird der Pixel als ungültig erklärt.

2.7 Üblicher Gesamtablauf mit Optimierungsschritten

1. Kosten Berechnen (hier Census)
2. Aggregation (hier Blockmatching oder SGM)
3. WTA links und rechts
4. Rechts-Links-Check
5. Subpixelinterpolation
6. Speckle Check / Interpolation

Davon wurden in dieser Arbeit die Punkte 1 bis 3 beschleunigt, da sie den Hauptteil der Rechenzeit verbrauchen. Bei den Punkten 5 und 6 handelt es sich um zusätzliche Filtertechniken, die das Ergebnis noch etwas verbessern können.
Bei der *Subpixelinterpolation* wird der Disparitätswert dadurch approximiert, dass durch die Kosten der beiden Pixel links und rechts neben dem Minimum eine Parabel gelegt wird. Der Scheitelpunkt dieser Parabel wird dann als Disparitätswert angenommen. Dies ist dann in der Regel ein Subpixelwert.
Der *Speckle Check* führt eine Segmentierung des Disparitätsbildes in Bereiche mit ähnlicher Disparität (z.B. ± 1 als Schwellenwert) durch. Danach werden Segmente mit wenig Pixeln, z. B. mit weniger als 100, invalidiert.

3 Implementierung

3.1 Kerntechnologie

Die Implementierung der beiden Algorithmen erfolgte in C++, da es eine gute Mischung aus Entwicklungszeit und Performance bot. Für die Entwicklung wurde Microsoft Visual Studio 2010, sowie das Cassandra Framework verwendet. Außerdem bot C++ insbesondere den Vorteil, dass mit den Intel Intrinsics Makros zur Verfügung stehen, die es ermöglichen die SIMD-Befehle (Single Instruction Multiple Data) der Intel CPU komfortabel zu verwenden ohne direkten Assembler-Code einzubinden. Die Übersetzung erfolgt als 32Bit Programm.

3.2 Cassandra

Cassandra [13] ist ein C++-Framework für die Erstellung von Bildverarbeitungsapplikationen der Hella Aglaia Mobile Vision GmbH. Dieses ist speziell auf den Automotive-Bereich ausgerichtet. Dabei ist es streng Modular aufgebaut, so dass man es um seine eigenen Module und Komponenten erweitern kann. Für eine Applikation werden dann die Komponenten in einem Datenflussnetz verbunden, wobei Komponenten für das Einlesen des Videomaterials und zur Darstellung der Ausgabebilder vorhanden sind. Außerdem ermöglicht das Framework die Anpassung von Parametern der Module zur Laufzeit und liefert dem Entwickler verschiedene Informationen wie Prozessor- und Hauptspeicherauslastung des Systems oder Berechnungszeiten einzelner Module pro Bild.
Für die beiden Algorithmen wurden in C++ eigene Module entwickelt, die die entsprechende Bildverarbeitung durchführen. Diese wurden dann mit den vorhandenen Modulen des Frameworks in einem Datenflussnetz kombiniert.

3.3 Intel SSE

Die Implementierung der beiden Algorithmen benutzt die Vektorrecheneinheiten der CPU. Für diesen Zweck existieren mit Intel SSE verschiedene SIMD-Befehlssätze (Single Instruction Multiple Data), die für die gleichzeitige Berechnung mehrerer Werte verwendet werden können. Dafür werden in C++ die Intel Intrinsics bereitgestellt, bei denen es sich um Makros handelt, die dann direkt in die Assembler-Befehle der CPU übersetzt werden.
Da die Perfomance der Algorithmen sehr wichtig ist, kann dadurch die Bildrate erhöht werden, dass besonders rechenintensive Bereiche hierdurch parallelisiert werden. Unter Verwendung von SSE im 32Bit Modus können 128 Bit breite Register berechnet werden, wobei diese aus 64, 32, 16 oder 8 Bit Werten bestehen können. Somit lassen sich für die Implementierungen dieser Arbeit immer 8 Werte gleichzeitig berechnen, da nach der Berechnung der Kosten der Census-Transformation die Werte für jeden Pixel nur 16 Bit groß sind.

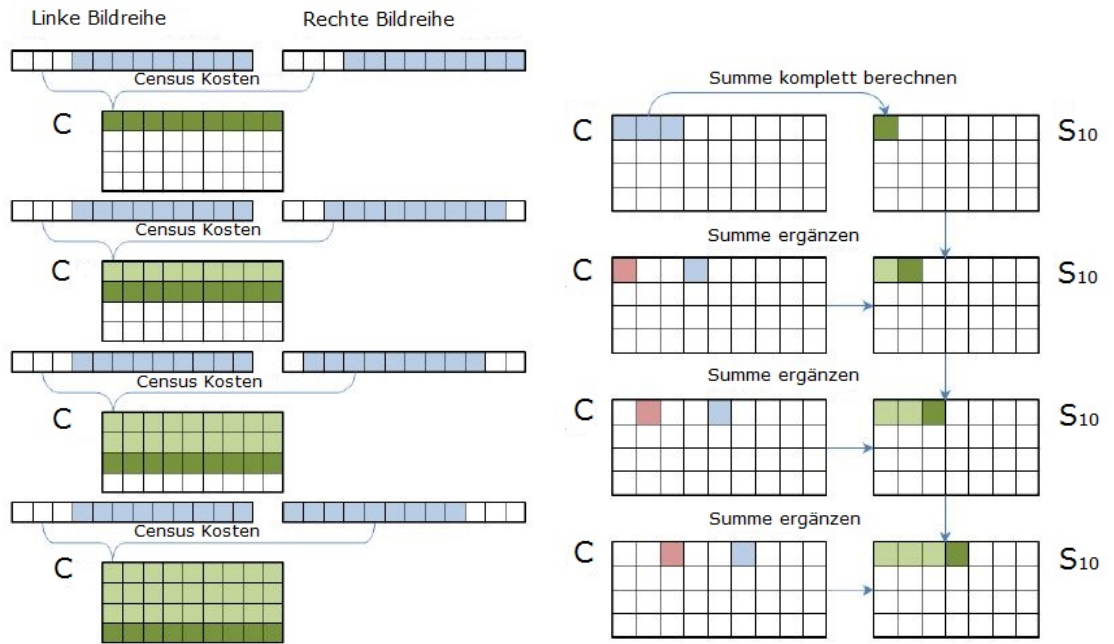
3.4 Multithreading

Zur weiteren Beschleunigung auf Systemen mit mehreren Prozessorkernen wurden Teile des Algorithmus zusätzlich dadurch parallelisiert, dass sie in mehreren Threads parallel berechnet werden.
Zu diesem Zweck wurde das Open MP Framework verwendet, dass die Erzeugung, Aufgabenzuweisung und Synchronisierung der Threads für den Programmierer besonders einfach macht. Es ist ausreichend Blöcke zu definieren, die dann automatisch durch den verfügbaren Threadpool abgearbeitet werden. Auch für die Synchronisierung müssen nur einfache Statements für eine Barriere oder als kritisch markierte Blöcke verwendet werden. Dies ist für das Framework ausreichend, um den entsprechenden Code zu generieren.

3.5 Blockmatching

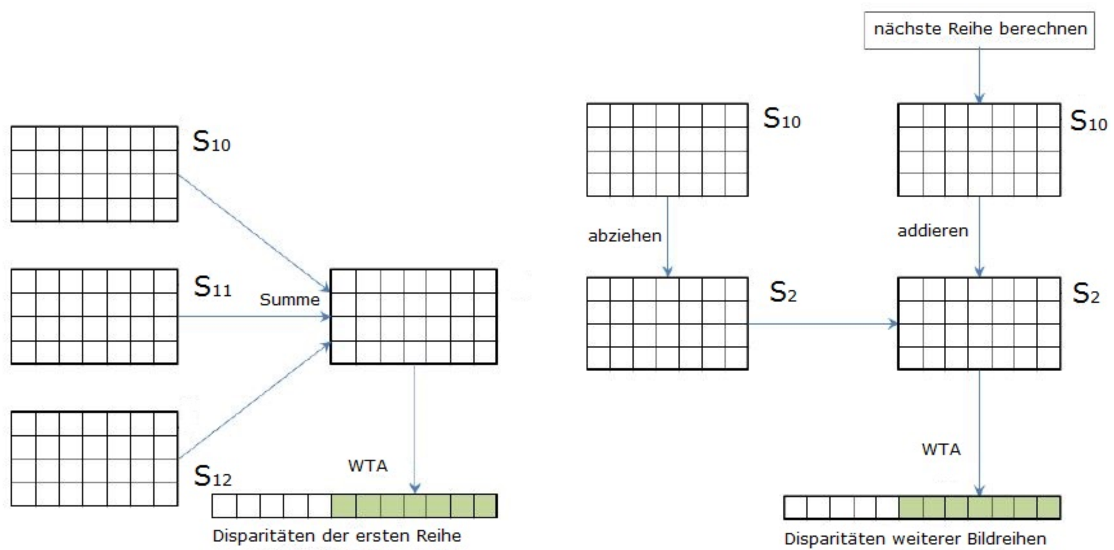
Das Blockmatching verwendet einen Sliding Window Ansatz, damit möglichst wenig Berechnungen mehrfach durchgeführt werden müssen und stattdessen viele Zwischenergebnisse wiederverwendet werden können. Dafür werden Datenstrukturen zum Zwischenspeichern von Ergebnissen verwendet (siehe Abb. 6).
Dabei werden zunächst für alle Pixel einer Reihe alle Census-Kosten für alle Disparitäten berechnet und in C gespeichert (ein *Bildbreite* $\times$ *maximale Disparität* Feld). Anschließend können jeweils in einer Zeile von C die Census-Kosten mit Hilfe des Sliding Window Verfahrens aufsummiert werden, die sich in der Breite des Aggregationsfensters um den jeweiligen Pixel befinden. Dabei werden dann die Summen in ein S_1 Feld für jeden Pixel und jede Disparität gespeichert.
Sliding Window bezeichnet dabei das Vorgehen, dass man für den ersten Pixel alle Werte im Aggregationsfenster aufsummiert und bei allen weiteren Pixeln lediglich einen Wert addieren und einen subtrahieren muss, da sich das Fenster nur um eine Stelle verschiebt [14].
Von den S_1 Feldern werden so viele benötigt wie das Aggregationsfenster hoch ist, da diese dann stellenweise aufsummiert und in S_2 gespeichert werden. Sind also alle S_1 Felder einmal initialisiert muss für jede weitere Reihe nur jeweils ein S_1 Feld abgezogen und addiert werden.
Der Vorteil dieser Vorgehensweise beim Berechnen der Blockmatching Kosten ist, dass für jeden Pixel und jede Disparität nur einmal die Census-Kosten berechnet werden müssen, und nicht mehrfach wie beim trivialen Vorgehen. Außerdem wird die Anzahl an Additionen durch das Sliding Window Verfahren stark reduziert [4].

Mit Hilfe der SIMD Befehle können zusätzlich beim Befüllen von C, beim Aggregieren der Spalten und beim Aggregieren innerhalb einer Zeile immer mehrere Spalten parallel berechnet werden. Dafür werden bei letzterem zunächst zwei mal acht Werte gelesen und diese anschließend jeweils nach einem Rechts-Shift auf die Summe addiert, so dass die ersten acht Werte dann die korrekten Summen enthalten. Auch beim WTA können mehrere Spalten gleichzeitig berechnet werden, wenn man die aktuell besten Werte in



(a) Census Kosten in einer Reihe für alle Pixel und Disparitäten berechnen

(b) Kosten innerhalb der Reihe summieren (Sliding Window)



(c) Kosten der Reihen summieren (Initialisierung)

(d) Bei weiteren Reihen Summen aktualisieren (Sliding Window)

Abb. 6: Datenstrukturen beim Blockmatching (Beispiel, abgeändert aus [4])

einem Array für die Zeile zwischenspeichert.

Zur weiteren Parallelisierung wird das gesamte Ausgabebild in horizontale Streifen aufgeteilt, von denen jeder Streifen von einem eigenen Thread berechnet wird. Die Anzahl der Threads (und damit auch Streifen) kann dabei variiert werden und entspricht standardmäßig der Anzahl an Prozessorkernen. Somit kann auf jedem Kern echt parallel gerechnet werden. Dadurch, dass das gesamte Bild aufgeteilt wird und es keine globalen Optimierungstechniken gibt, gibt es auch keine Speicherbereiche, die von mehreren Threads gleichzeitig beschrieben werden können. Daher sind auch keine Synchronisierungen notwendig.

3.6 Semi Global Matching

Die Implementierung des Semi Global Matchings verwendet Dynamische Programmierung zur Berechnung der Disparitäten. Dabei wird die Rekursionsgleichung (6) aufgelöst, indem in umgekehrter Weise vom Startwert aus alle Zwischenergebnisse berechnet werden. Dazu werden in einem Feld jeweils die Kosten der Disparitätswerte der letzten Bildreihe zwischengespeichert, damit sie bei der Berechnung der aktuellen Reihe effizient zur Verfügung stehen. Außerdem wird das Minimum der letzten Zeile gespeichert, damit es für die Berechnung der Kosten mit Bestrafung P_2 $(\min_{k \in D} L_r(p-r,k)+P_2)$ verwendet werden kann. Dadurch ist es nicht notwendig jedes Mal das Minimum zu berechnen. Damit für die berechneten Kosten 16bit Zahlen ausreichen, werden zusätzlich in jeder Reihe die Kosten normalisiert, indem von ihnen die minimalen Kosten (von allen Disparitäten) des letzten Pixels des Pfades abgezogen werden. Dies hat keine Auswirkung auf das Ergebnis, verhindert aber, dass die Summen zu groß werden.

Die Implementierung berechnet die Kosten entlang von 8 Pfaden, wobei mit den ersten vier Pfaden in der linken oberen Ecke des Bildes begonnen wird (siehe Abb. 7). Diese vier Pfade (von links nach rechts, oben nach unten, links oben nach rechts unten und rechts oben nach links unten) werden für die Pixel auf dem Weg in die rechte untere Ecke zeilenweise berechnet (erste Phase), da die Ergebnisse der Pixel aus diesen Pfaden dann bereits berechnet sind. Dort angekommen werden nun die restlichen vier Pfade auf dem Rückweg in die linke obere Ecke berechnet (zweite Phase). Die berechneten Kosten werden in einem Kubus gespeichert (Kosten für alle Pixel und alle Disparitäten), der dann anschließend mit Hilfe des WTA-Verfahrens ausgewertet wird.
Dabei reicht dieser eine Kubus aus, um auch das WTA-Verfahren für die umgekehrte Matching-Richtung (Referenz- und Zielbild werden getauscht und ausgewertet) anzuwenden, da der Kostenkubus bereits alle nötigen Ergebnisse enthält [14]. Danach kann dann der Links-Rechts-Check angewendet werden.
Die Verwendung von 16 Pfaden zeigte wie bei Humenberger et al. [1] keine spürbare Verbesserung und wurde wegen der höheren Laufzeit und dem erhöhten Aufwand nicht weiter optimiert.

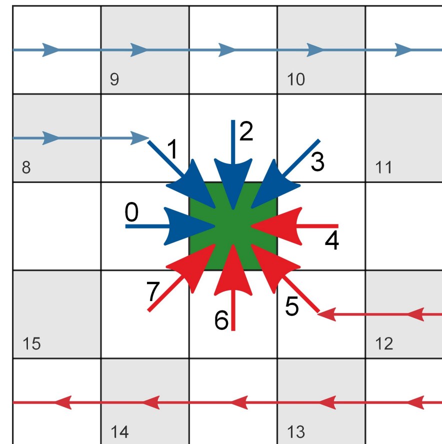


Abb. 7: Dargestellt sind die 8 Pfade (Ziffern 0-7) beim SGM, deren Pixel für die Kosten des grünen Pixels in der Mitte berücksichtigt werden. Die vier blauen Pfade werden in der ersten (hellblaue Linie) und die roten in der zweiten Phase (hellrote Linie) berechnet. Die grauen Pixel mit den Ziffern 8 bis 15 würden bei 16 Pfaden zusätzlich berechnet werden.

Mit Hilfe der SIMD Befehle konnten diese Berechnungen nun noch beschleunigt werden, indem bei der Berechnung der Pfade immer acht Ergebnisse von nebeneinander liegenden Disparitäten gleichzeitig berechnet werden. Dafür werden die acht Census-Kosten, sowie zwei mal acht Ergebnisse des letzten Pixels entlang des Pfades geladen. Mit Hilfe von Shifts können dann die Kosten des letzten Pixels ohne Strafe, sowie die beiden möglichen Summen mit den P_1 Kosten bestimmt werden. Die Summe mit den P_2 Kosten ist bei allen acht Werten gleich und muss nur einmal berechnet werden. Mit Hilfe von Vergleichen können dann die minimalen Kosten bestimmt und anschließend gespeichert werden.

Für die Parallelisierung auf mehreren Prozessorkernen wurden zwei verschiedene Varianten mit Hilfe von OpenMP implementiert. Diese parallelisieren auf verschiedenen Ebenen des Algorithmus.
Bei der ersten Variante werden mit einem einzigen Ladevorgang der Kosten der Census-Transformation die Kosten aller Pfade parallel berechnet, bevor diese dann anschließend aufsummiert werden. Der Vorteil liegt darin, dass für die Berechnung aller Pfade nur ein einziger Ladevorgang für die Census-Kosten notwendig ist und dass nur ein einziger Speichervorgang für die Gesamtkosten aus allen Pfaden benötigt wird. Dafür sind allerdings die Programmteile sehr kurz, die parallel laufen. Somit sind häufige Kontextwechsel notwendig.
Die zweite Variante ist, an jeder Stelle (x und y Wert fest) parallel jeweils für alle Pfade alle Disparitäten zu berechnen. Bei dieser Lösung sind die Programmteile etwas länger als bei der ersten, die parallel berechnet werden, und die Gesamtkosten können trotzdem noch zusammen weggespeichert werden. Das Laden muss hier allerdings mehrfach erfolgen.

Eine weitere Variante des Semi-Global-Matchings wurde implementiert, die das Ausgangsbild in horizontale Streifen teilt, die dann noch eine gewisse Überlappung haben. Diese wurde von Hirschmüller vorgestellt [15]. Humenberger et al. [1] setzten dieses Verfahren für eine höhere Performance um.
Der Vorteil dieses Verfahrens ist, dass der benötigte Speicher reduziert wird, da nur ein kleinerer Kostenkubus zum Speichern der Zwischenergebnisse notwendig ist. Außerdem lassen sich diese Streifen dann sehr einfach parallel berechnen, da die Threads dann auf disjunktem Speicher operieren.
Der Nachteil ist allerdings, dass die Bereiche, über die die Pfade gehen, immer kleiner werden und sich damit die Qualität des Disparitäsbildes mit steigender Anzahl an Streifen verringert. Dies kann durch eine große Überlappung der Streifen verringert werden. Bei dieser Implementierung kann eine beliebige Anzahl vorgegeben werden, nach der dann das Bild aufgeteilt wird. Außerdem ist die Anzahl der Threads, die die einzelnen Streifen berechnen, beliebig einstellbar.

4 Ergebnisse

4.1 Testkonfiguration

Die Laufzeit beider Verfahren wurde anhand zweier Videoaufnahmen getestet. Dabei beinhaltet Video 1 (siehe Abb. 8) eine Szene, wo sich das Auto auf einigen Seitenstraßen in der Stadt befindet. Es hat eine Auflösung von 768×480 Pixeln. Das zweite Video (siehe Abb. 9)zeigt eine Szene, bei dem sich das Auto auf der Autobahn befindet. Die Auflösung beträgt dabei 656×541 Pixel. Es stammt aus einer Arbeit von Meister et al. [16]. Für beide Videos wurden die Disparitäten im Bereich von 0 bis 63 berechnet.
Das Testsystem besitzt eine Intel Core i5-2520M CPU (2 Kerne, 2,5 GHz) und 4 GB RAM (siehe Tab. 2, Tab. 3). Außerdem wurde das Semi Global Matching noch auf einer Intel Core i7-2600 CPU (4 Kerne, 3,4 GHz, System mit 8 GB RAM) getestet und mit dem Semi Global Blockmatching aus OpenCV verglichen (siehe Tab. 4).
Für die Darstellung der Disparitätsbilder werden zwei verschiedene Arten verwendet. In Abbildung 8 (siehe Abb. 8) wird die Disparität farbcodiert, sodass von warmen zu kalten Farben die Disparität abnimmt. Diese Farben werden dann über das ursprüngliche Bild gelegt. Dagegen werden in Abbildung 9 (siehe Abb. 9) die Disparitäten direkt als Graustufen angezeigt. Je dichter der Punkt an der Kamera ist, desto heller ist er. Neben diesem Bild wird noch einmal ein Ursprungsbild zum Vergleich gezeigt.

4.2 Blockmatching

Das Blockmatching liefert wie zu erwarten war ein schlechteres Ergebnis als das Semi Global Matching, da es keine globale Optimierungsstrategie verwendet und sich somit nur auf lokale Informationen bezieht. Vor allem in texturarmen Bereichen wie dem Himmel in Abbildung 8 (siehe Abb. 8) oder der Straße in Abbildung 9 (siehe Abb. 9) enthält das Disparitätsbild viele falsche oder undefinierte Werte. Insgesamt werden aber die meisten Objekte gut erkannt. Außerdem schneidet der Algorithmus im Vergleich zum Semi Global Blockmatching (SGBM) aus OpenCV sehr gut ab (siehe Abb. 9). Dort enthält das Disparitätenbild weniger undefinierte Werte in den texturarmen Bereichen und berechnet außerdem noch die Teile des linken Randes für Disparitäten, die klein genug sind. Allerdings mussten die Eingabewerte für das SGBM nach 8Bit konvertiert werden (ansonsten werden 16Bit verwendet), was das schlechtere Bild z.T. erklären kann.
Dass das Blockmatching keine globale Strategie verwendet, macht sich dafür aber auch in der Laufzeit bemerkbar. Die Rechenzeit liegt bei etwa 50ms pro Bild, womit sich also etwa 20 Bilder pro Sekunde berechnen lassen. Dies lässt sich zu einem Großteil darauf zurückführen, dass es ausreichend ist, nur die Disparitäten für die letzten Reihen zu speichern, die sich noch im Aggregationsfenster befinden. Es müssen also nicht für alle Pixel die Disparitäten gespeichert werden, was den Zwischenspeicher vergleichsweise sehr klein hält.
Wie in Tabelle 1 (siehe Tab. 1) zu sehen ist, haben die Optimierungen alle eine Beschleunigung der Berechnung erzielen können. Allerdings ist die Parallelisierung nur sinnvoll, wenn mehrere CPUs oder Kerne zur Verfügung stehen. Bei dem vorhandenen Testsystem

Tab. 1: Durchschnittliche Berechnungszeiten beim Blockmatching pro Bild in ms auf einem 2 Kern Core i5-2520M mit 2,5 GHz

Optimierungsgrad	Video 1	Video 2	Speedup
trivial	685	655	-
mit Datenstrukturen	186	175	3,7
mit SSE	66	62	10,5
parallel in 2 Threads	54	50	12,9
parallel in 4 Threads	56	53	12,3

ist die Laufzeit optimal, wenn die Anzahl der Threads der Anzahl an Kernen entspricht auf denen sie laufen.

4.3 Semi Global Matching

Das Semi Global Matching liefert sehr gute Ergebnisse bei beiden Testvideos ab. Seine Stärken zeigen sich vor allem in den texturarmen Bereichen wie Himmel oder Straße, aber auch generell auf den Oberflächen größerer Objekte wie den Autos in Abbildung 8 (siehe Abb. 8) oder den Verkehrsschildern in Abbildung 9 (siehe Abb. 9). Generell ergibt sich bei allen Bildern ein sehr vollständiges Ergebnis mit nur wenigen Fehlstellen oder Fehlern. Dabei erzielt das SGM auch ein wesentlich besseres Disparitätenbild als das Semi Global Blockmatching aus OpenCV, welches in den texturarmen Bereichen große undefinierte Bereiche enthält und am linken Rand keine Werte für Disparitäten berechnet, die klein genug sind (siehe Abb. 9).

Die Performance des Algorithmus ließ sich vor allem mit Hilfe der SIMD-Befehle sehr stark beschleunigen. Dadurch konnte sie von 3,2s auf etwa 264ms pro Bild im zweiten Video beschleunigt werden (auf einem Core i5), was 3,8 Bildern pro Sekunde entspricht (siehe Tab. 2). Auf dem Core i7 konnte ebenfalls eine deutliche Verbesserung von 2,7s auf 207ms (4,8 fps) erzielt werden. Damit ist sie auf diesem Prozessor ebenfalls etwas schneller als das Semi Global Blockmatching von OpenCV, obwohl dieses nur 8 Bit Eingabebilder verwendet und ein wesentlich schlechteres Ergebnis liefert. Betrachtet man in Tabelle 5 die berechneten Disparitäten pro Sekunde, so ist der Algorithmus sogar etwas schneller als die Implementierung von Gehrig und Rabe und etwa doppelt so schnell wie die von ihnen referenzierte GPU Implementierung. Lediglich die FPGA Variante ist noch einmal wesentlich schneller [2].
Die beiden Varianten zur Parallelisierung (ohne das Gesamtbild in Streifen zu teilen) konnten allerdings keinerlei Beschleunigung erzielen. Im Gegenteil verlangsamten sie die Rechenzeit auf 9 bzw. 1,6s (siehe Tab. 2). Ursache dafür ist vermutlich, dass das Open MP Framework für diese kurzen und häufigen parallelen Passagen zu schwergewichtig ist, wodurch der Overhead bei der Synchronisierung zu groß wird und mehr Zeit benötigt als die eigentlichen Berechnungen.

Tab. 2: Durchschnittliche Berechnungszeiten beim Semi Global Matching pro Bild auf einem 2 Kern Core i5-2520M mit 2,5 GHz

Version	Video 1	Video 2	Speedup
unoptimiert	3,3s	3,2s	-
mit SSE	280ms	264ms	11,9
parallel Variante 1	9s	9s	0,4
parallel Variante 2	1,6s	1,6s	2,0

Bei der Auswertung der SGM-Variante, die das Ausgangsbild in Streifen teilt, zeigt sich, dass die Ergebnisse auch bei einer geringen Streifenbreite noch sehr gut sind. Für viele Anwendungen könnten diese Ergebnisse ausreichend sein. Bei einer Streifenbreite von mehr als 120 Pixeln (Bild in vier Streifen geteilt) zeigt sich außerdem, dass sich in der Realen Szene (siehe Abb. 10 *b)*) kaum Unterschiede zum Ergebnis unter Verwendung des gesamten Bildes für die Pfade erkennen lassen. Erst bei geringeren Breiten kommt es verstärkt vor, dass in den texturarmen Bereichen die Anzahl an invalidierten und falschen Disparitätswerten zunimmt. Dies sieht man in der Bildstrecke vor allem im Himmel und auf den Autooberflächen.

Insgesamt erhält man auf diese Weise also eine einfache Variante die Berechnung zu parallelisieren, ohne dass das Ergebnis spürbar verschlechtert wird, da dabei keine Synchronisierungen notwendig sind. Die Laufzeit verbessert sich dadurch auf die Spitzenwerte von 185ms (5,4fps) auf dem Core i5 (2 Threads, 2 Streifen, siehe Tab. 3) und erreicht auf dem Core i7 sogar fast die gewünschten 10fps mit 101ms (4 Threads, 4 Streifen, siehe Tab. 4). Diese Variante ist außerdem im Gegensatz zur sequentiellen noch etwas schneller als die SGM Implementierung auf einem FPGA von Gehrig et al. [18] (siehe Tab. 5).
Zu einem ähnlichen qualitativen Ergebnis kommen auch Humenberger et al. [1]. Allerdings kann die Implementierung aus dieser Arbeit nicht von dem verringerten Zwischenspeicher profitieren. In Tabelle 3 lässt sich anhand der durchschnittlichen Laufzeiten pro Bild erkennen, dass der Geschwindigkeitszuwachs lediglich auf die parallele Berechnung der Streifen im Vergleich zur nicht parallelen Variante zurückzuführen ist. Mit steigender Anzahl an Streifen nimmt dann die Laufzeit wieder weiter zu. Auch beim Core i7 liegt der beste Wert dort wo die Anzahl an Kernen, Threads und Steifen übereinstimmen (siehe Tab. 4). Der Overhead beim Aufruf zur Berechnung eines Streifens scheint demnach größer zu sein als der Vorteil nur kleineren Zwischenspeicher zu benötigen.
Bei einem Cache von 8MB (8388608Bit) beim Core i7 dürften bei den 16Bit großen Kosten, 64 Disparitäten und den Bildbreiten von 768 bzw. 656 Pixeln die Streifen höchstens 10 bzw. 12 Pixel hoch sein, damit alleine der Kostenkubus in den Cache passt. Beim Core i5 mit 3MB Cache sind es sogar nur maximal 3 bzw. 4 Pixel. Damit sind die Caches für die hier verwendeten Datenstrukturen nicht groß genug.

Tab. 3: Berechnungszeiten pro Bild in ms beim Semi Global Matching mit Aufteilung des Bildes in Streifen unter Verwendung von zwei Threads auf einem 2 Kern Core i5-2520M mit 2,5 GHz

Anzahl Streifen	Video 1	Video 2
1 (nur 1 Thread)	280	264
2	190	185
4	195	191
8	204	198
16	223	208
32	248	229
120	408	347
240	551	466

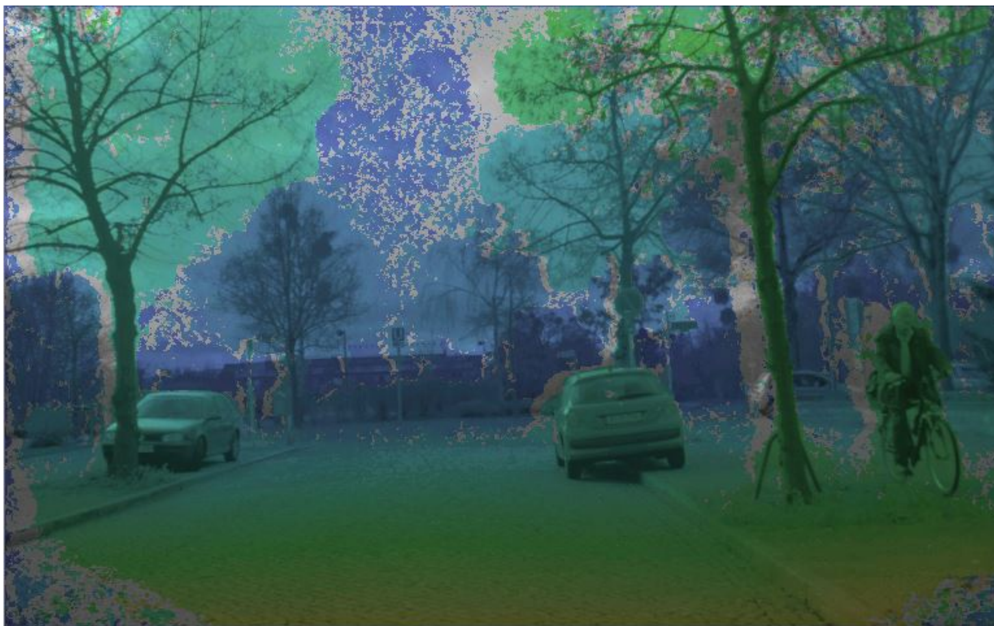
Tab. 4: Berechnungszeiten pro Bild in ms auf einem 4 Kern Core i7-2600 mit 3,4 GHz für Video 2. Dabei werden verschiedene Algorithmen bzw. Anzahlen an Streifen beim SGM abgedeckt. Auch die Größe des Threadpools variiert.

Algorithmus	Streifen	1 Thread	2 Threads	4 Threads	8 Threads
SGM (unoptimiert)	1	2740	-	-	-
SGM	1	207	-	-	-
SGM	2	-	131	139	154
SGM	4	-	130	101	122
SGM	6	-	132	114	105
SGM	8	-	133	102	108
SGM	12	-	137	106	105
SGM	16	-	137	105	110
OpenCV SGBM	1	213	-	-	-

Tab. 5: Vergleich zwischen den Anzahlen an berechneten Disparitäten pro Sekunde bei verschiedenen SGM Implementierungen auf verschiedenen Systemen. Die oberen drei Zeilen geben die Werte der Implementierungen aus dieser Arbeit auf dem 4 Kern Core i7-2600 mit 3,4 GHz für Video 2 an. Die unteren drei geben Referenzwerte von Gehrig und Rabe, sowie von ihnen referenzierte Ergebnisse an [2] [17] [18].

Recheneinheit	Algorithmus	Disp/s [10^6/s]
CPU	unoptimiert	10
CPU	SSE	130
CPU	parallele Streifen	266
CPU	Referenz [2]	96
GPU	Referenz [17]	64
FPGA	Referenz [18]	218

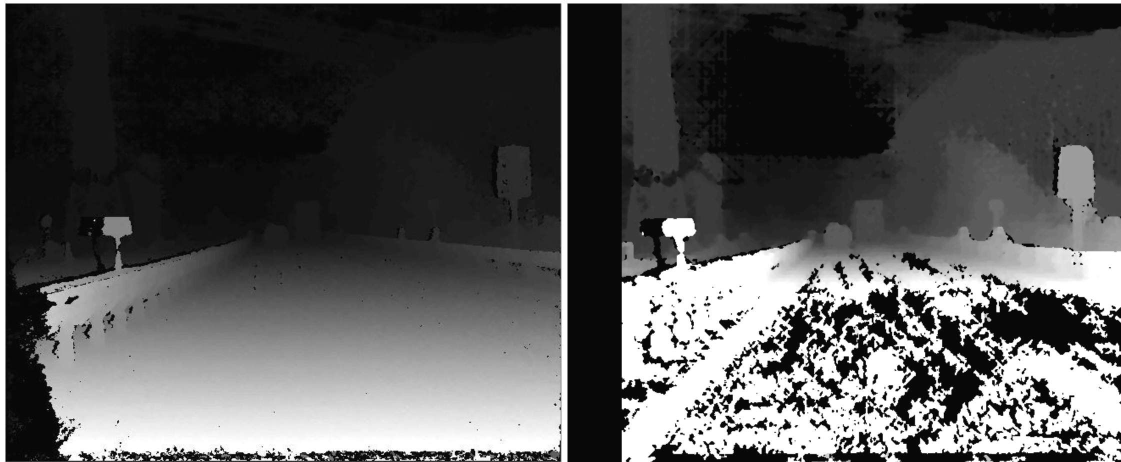
(a) Ergebnis des Blockmatchings



(b) Ergebnis des Semi Global Matchings

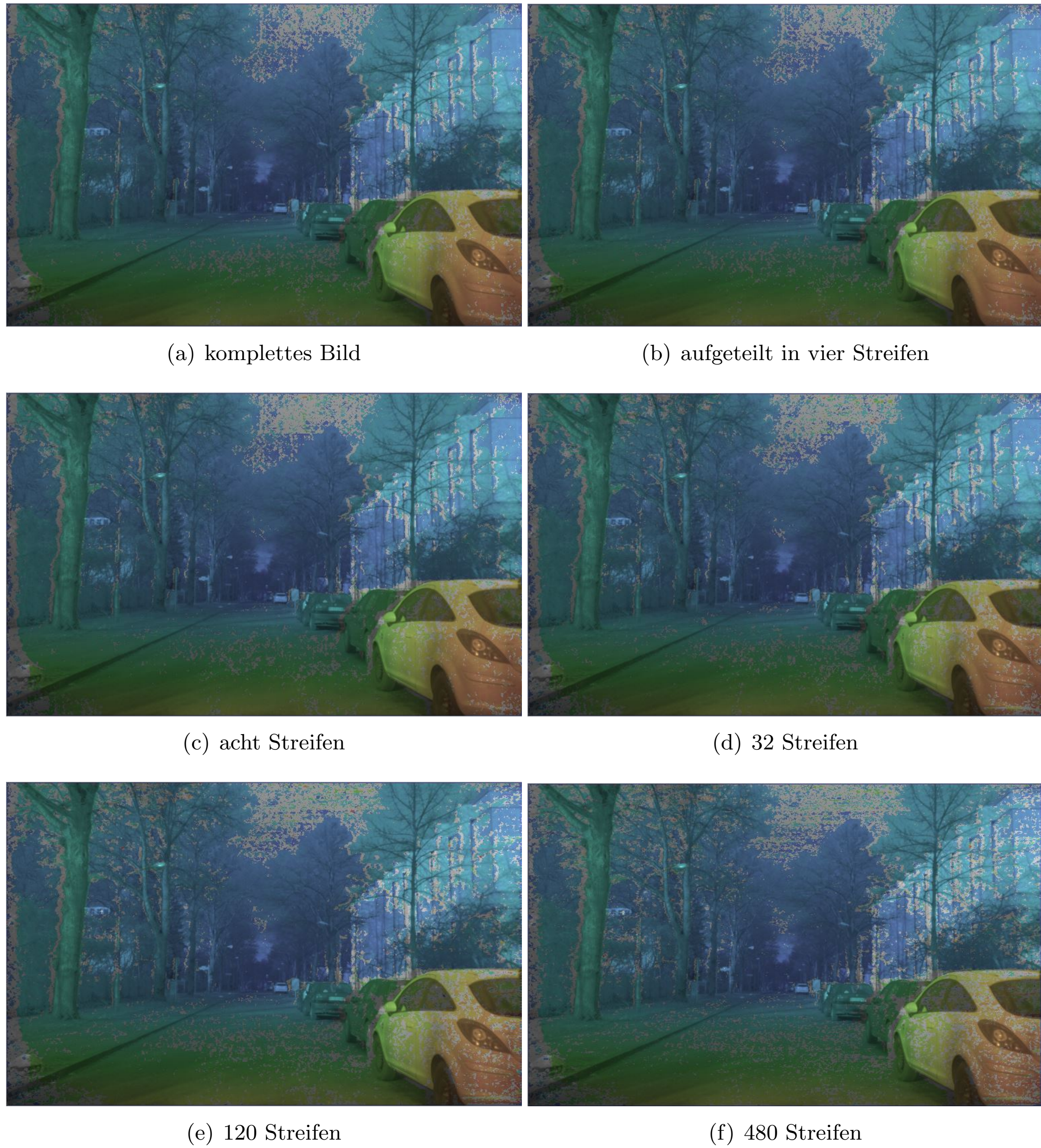
Abb. 8: Vergleich der resultierenden Disparitätsbilder aus den beiden Verfahren, Video 1

(a) Ursprungsbild (b) aus Blockmatching



(c) aus SGM (d) aus OpenCV SGBM

Abb. 9: Resultierende Disparitätsbilder der beiden Verfahren neben einem Ausgangsbild sowie dem Ergebnis des OpenCV Semi Global Blockmatchings, Video 2



(a) komplettes Bild
(b) aufgeteilt in vier Streifen
(c) acht Streifen
(d) 32 Streifen
(e) 120 Streifen
(f) 480 Streifen

Abb. 10: Gegenüberstellung der Dispariätsbilder bei verschiedener Anzahl an horizontalen Streifen beim Semi Global Matching (Quelle: Video 1)

5 Diskussion und Ausblick

Die Bestzeiten von beiden Algorithmen mit etwa 50ms für das Blockmatching und etwa 200ms für das Semi Global Matching bei 768×480 Pixeln auf einem 2 Jahre alten Intel Core i5 können sich gut mit den Ergebnissen aus anderen Arbeiten messen. Diese lagen bei Zinner et al. [3] mit 68ms pro Bild auf 435×50 Pixeln mit einem Core 2 Duo für das Blockmatching noch etwas darüber, was aber dem älteren Prozessor zuzuschreiben sein dürfte.
Auch im Vergleich zu Bodkin [4], der eine Zeit von 53.22 ms pro Bild (384×288 Pixel mit 16 Disparitäten) auf einem Core I7-720Q ohne parallele Threads bzw. 14,3ms parallel auf 4 Kernen erreichen konnte, kann die Implementierung mithalten. Sie erreichte mit 62ms ohne Parallelisierung bzw. 50ms auf zwei Kernen parallel auf einem etwa dreimal so großen Bild mit vier mal mehr Disparitäten ein etwas besseres Ergebnis. Außerdem ist die Zeit für das Blockmatching damit auch weit über den anvisierten 10 fps und lässt damit noch genug Zeit für weitere Weiterverarbeitungsschritte.
Für das Semi Global Matching konnte bei Gehrig und Rabe [2] eine Rechenzeit von 50ms für 320×240 Pixel auf einem Core i7 erreicht werden. Damit liegt sie trotz ihres verwendeten Image Subsamplings (das hier nicht implementiert wurde) in einer ähnlichen Größenordnung wie der Algorithmus aus dieser Arbeit. Diese Implementierung des Global Matchings konnte auf einem Intel Core i7 mit 4 Kernen knapp 100ms erreichen. Damit kommt die Implementierung auch sehr nahe an das Ziel von 10 Bildern pro Sekunde heran. Aus dem Vergleich in Tabelle 5 geht außerdem hervor, dass damit mehr Disparitäten pro Sekunde berechnet werden als bei Gehrig und Rabe [2], sowie bei Ernst und Hirschmüller auf einer GPU [17] und Gehrig et al. auf einem FPGA [18]. Dabei ist allerdings noch zu beachten, dass diese Arbeiten zwischen 2008 und 2010 veröffentlicht wurden. Auf aktuelleren System wären wohl auch noch schnellere Ergebnisse möglich.

Der Einsatz von SSE zur Beschleunigung hat sich als sehr effektiv erwiesen. So konnte bei beiden Algorithmen dadurch eine etwa 10 mal höhere Geschwindigkeit im Vergleich zur ursprünglichen Version erzielt werden. Es ist also zu diesem Zweck allgemein zu empfehlen.
Der Einsatz von OpenMP hat sich auch als gute Maßnahme herausgestellt, allerdings muss die Parallelisierung dafür auf einer hohen Ebene geschehen, da ansonsten ein gegenteiliger Effekt entstehen kann. Dieser tritt ein, wenn der parallele Programmcode zu kurz ist. Dann erzeugen die Kontextwechsel und Synchronisierungen zu viel Overhead und können sogar mehr Zeit verbrauchen als der eigentliche Programmcode selbst. In diesem Fall sollte man es mit einer leichtgewichtigeren Implementierung als durch OpenMP versuchen.

In Bezug auf die Qualität der Ergebnisse leisten die Algorithmen, was von ihnen erwartet wurde. Das Blockmatching kann ein gutes Ergebnis mit großen Schwächen in texturarmen Bereichen erzielen. Das Semi Global Matching erreicht dagegen sehr gute Ergebnisse, die nur sehr wenige Fehlstellen enthalten und fast so gut sind wie die Ergebnisse aus aufwändigeren globalen Verfahren [8].

Mit kommenden Prozessorgenerationen werden die Laufzeiten für die vorgestellten Algorithmen noch weiter sinken, so dass diese dann noch besser für Echtzeitanwendungen auf einer gängigen CPU laufen können. Außerdem könnten mit Hilfe von Intel AVX2, das bei der aktuellen Prozessorgeneration hinzugekommen ist, noch höhere Bildraten erzielt werden, da damit im Gegensatz zu SSE die Berechnung mit 256Bit breiten Registern möglich ist. Es können somit doppelt so viele Elemente im Vergleich zu SSE mit einer Instruktion berechnet werden. Für die Algorithmen aus dieser Arbeit könnten also 16 statt 8 Werte gleichzeitig berechnet werden.
Darüber hinaus gibt es noch weiteres Potential das Semi Global Matching zu verbessern. So könnte man das Image Subsampling von Gehrig und Rabe [2] noch implementieren. Außerdem könnte es in Zukunft möglich sein die noch besseren globalen Verfahren (Belief Propagation, Graph Cuts) auf CPUs in Echtzeit zu realisieren. Dort gibt es bereits erste Entwicklungen. Für Belief Propagation konnte auf einer GPU für kleine Bilder gezeigt werden, dass das Verfahren echtzeitfähig ist [19].

Danksagung

Ich danke Robert Spangenberg für seinen Rat und die ausgiebige Unterstützung bei dieser Arbeit.
Außerdem bedanke ich mich, dass ich in der Arbeitsgruppe Künstliche Intelligenz und Robotik und dem Projekt AUTONOMOS unter Leitung von Prof. Raúl Rojas bzw. Tinosch Ganjineh die Arbeit anfertigen durfte.

Abbildungsverzeichnis

1 Autonomes Fahrzeug MadeInGermany des Projekts AUTONOMOS der FU Berlin . . . 2
2 Stereo Kamerasystem an der Windschutzscheibe von MadeInGermany . . 3
3 Triangulierung von zwei Punkten P und Q ausgehend von zwei verschiedenen Betrachtungswinkeln [5] . . . 4
4 Census-transformiertes Bild in Graustufen dargestellt, aus Video 1 (siehe Abschnitt 4.1) . . . 6
5 Probleme mit zu großen Blöcken: Sind sie zu groß überdecken sie oft Bereiche mit verschiedenen Abständen zum Beobachter [5] . . . 7
6 Datenstrukturen beim Blockmatching (Beispiel, abgeändert aus [4]) . . . 12
7 Dargestellt sind die 8 Pfade (Ziffern 0-7) beim SGM, deren Pixel für die Kosten des grünen Pixels in der Mitte berücksichtigt werden. Die vier blauen Pfade werden in der ersten (hellblaue Linie) und die roten in der zweiten Phase (hellrote Linie) berechnet. Die grauen Pixel mit den Ziffern 8 bis 15 würden bei 16 Pfaden zusätzlich berechnet werden. . . . 14
8 Vergleich der resultierenden Disparitätsbilder aus den beiden Verfahren, Video 1 . . . 20
9 Resultierende Disparitätsbilder der beiden Verfahren neben einem Ausgangsbild sowie dem Ergebnis des OpenCV Semi Global Blockmatchings, Video 2 . . . 21
10 Gegenüberstellung der Dispariätsbilder bei verschiedener Anzahl an horizontalen Streifen beim Semi Global Matching (Quelle: Video 1) . . . 22

Literatur

[1] M. Humenberger, T. Engelke, and W. Kubinger. A census-based stereo vision algorithm using modified semi-global matching and plane fitting to improve matching quality. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer Society Conference on*, pages 77–84, 2010.

[2] S.K. Gehrig and C. Rabe. Real-time semi-global matching on the cpu. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer Society Conference on*, pages 85–92, 2010.

[3] Christian Zinner, Martin Humenberger, Kristian Ambrosch, and Wilfried Kubinger. An optimized software-based implementation of a census-based stereo matching algorithm. In George Bebis, Richard Boyle, Bahram Parvin, Darko Koracin, Paolo Remagnino, Fatih Porikli, Jörg Peters, James Klosowski, Laura Arns, YuKa Chun, Theresa-Marie Rhyne, and Laura Monroe, editors, *Advances in Visual Computing*, volume 5358 of *Lecture Notes in Computer Science*, pages 216–227. Springer Berlin Heidelberg, 2008.

[4] B. H. Bodkin. Real-Time Mobile Stereo Vision. Master's thesis, University of Tennessee, 2012.

[5] Stefano Mattoccia. *Stereo Vision: Algorithms and Application.* University of Bologna, 2012. Online verfügbar bei http://vision.deis.unibo.it/ smatt/Seminars/StereoVision.pdf; besucht am 16.10.2013 19:20.

[6] Ramin Zabih and John Woodfill. Non-parametric local transforms for computing visual correspondence. In Jan-Olof Eklundh, editor, *Computer Vision — ECCV '94*, volume 801 of *Lecture Notes in Computer Science*, pages 151–158. Springer Berlin Heidelberg, 1994.

[7] Bogusław Cyganek. Comparison of nonparametric transformations and bit vector matching for stereo correlation. In Reinhard Klette and Joviša Žunić, editors, *Combinatorial Image Analysis*, volume 3322 of *Lecture Notes in Computer Science*, pages 534–547. Springer Berlin Heidelberg, 2005.

[8] H. Hirschmüller. Accurate and efficient stereo processing by semi-global matching and mutual information. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 2, pages 807–814 vol. 2, 2005.

[9] H. Hirschmüller and D. Scharstein. Evaluation of cost functions for stereo matching. In *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on*, pages 1–8, 2007.

[10] Jonathan S. Yedidia, William T. Freeman, and Yair Weiss. Exploring artificial intelligence in the new millennium. chapter Understanding belief propagation and its

generalizations, pages 239–269. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[11] V. Kolmogorov and R. Zabih. Computing visual correspondence with occlusions using graph cuts. In *Computer Vision, 2001. ICCV 2001. Proceedings. Eighth IEEE International Conference on*, volume 2, pages 508–515 vol.2, 2001.

[12] Pascal Steingrube, Stefan K. Gehrig, and Uwe Franke. Performance evaluation of stereo algorithms for automotive applications. In Mario Fritz, Bernt Schiele, and JustusH. Piater, editors, *Computer Vision Systems*, volume 5815 of *Lecture Notes in Computer Science*, pages 285–294. Springer Berlin Heidelberg, 2009.

[13] Hella Aglaia Mobile Vision GmbH. Cassandra homepage: http://www.cassandra-vision.com/.

[14] K. Muhlmann, D. Maier, J. Hesser, and R. Maenner. Calculating dense disparity maps from color stereo images, an efficient implementation. In *Stereo and Multi-Baseline Vision, 2001. (SMBV 2001). Proceedings. IEEE Workshop on*, pages 30–36, 2001.

[15] H. Hirschmüller. Stereo processing by semiglobal matching and mutual information. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(2):328–341, 2008.

[16] S. Meister, B. Jähne, and D. Kondermann. Outdoor stereo camera system for the generation of real-world benchmark data sets. *Optical Engineering*, 51(02):021107, 2012.

[17] Ines Ernst and Heiko Hirschmüller. Mutual information based semi-global stereo matching on the gpu. In George Bebis, Richard Boyle, Bahram Parvin, Darko Koracin, Paolo Remagnino, Fatih Porikli, Jörg Peters, James Klosowski, Laura Arns, YuKa Chun, Theresa-Marie Rhyne, and Laura Monroe, editors, *Advances in Visual Computing*, volume 5358 of *Lecture Notes in Computer Science*, pages 228–239. Springer Berlin Heidelberg, 2008.

[18] StefanK. Gehrig, Felix Eberli, and Thomas Meyer. A real-time low-power stereo vision engine using semi-global matching. In Mario Fritz, Bernt Schiele, and JustusH. Piater, editors, *Computer Vision Systems*, volume 5815 of *Lecture Notes in Computer Science*, pages 134–143. Springer Berlin Heidelberg, 2009.

[19] Qingxiong Yang, Liang Wang 0002, Ruigang Yang, Shengnan Wang, Miao Liao, and David Nistér. Real-time global stereo matching using hierarchical belief propagation. In Mike J. Chantler, Robert B. Fisher, and Emanuele Trucco, editors, *BMVC*, pages 989–998. British Machine Vision Association, 2006.