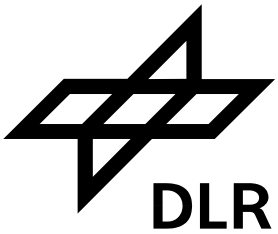


Implementierung des Feature-Matching-Verfahrens ORB auf der GPU unter Nutzung von Stereoinformationen

Bachelorarbeit

für die Prüfung zum
Bachelor of Engineering

des Studienganges Informationstechnik
an der Dualen Hochschule Baden-Württemberg Mannheim

von
Florian Lehmann

11.09.2017

Bearbeitungszeitraum	12 Wochen
Matrikelnummer, Kurs	9205404,TINF14ITIN
Ausbildungsfirma	Deutsches Zentrum für Luft- und Raumfahrt, Berlin-Adlershof
Betreuer der Ausbildungsfirma	Dr. Sergey Zuev
Gutachter der Dualen Hochschule	Prof. Dr. Rainer Colgen

Eigenständigkeitserklärung

Ich versichere durch meine Unterschrift, dass ich die hier vorgelegte Arbeit selbstständig verfasst habe. Ich habe mich dazu keiner anderen als der im Anhang verzeichneten Quellen und Hilfsmittel, insbesondere keiner nicht genannten Onlinequellen, bedient. Alle aus den benutzten Quellen wörtlich oder sinngemäß übernommenen Teile (gleich ob Textstellen, bildliche Darstellungen usw.) sind als solche einzeln kenntlich gemacht.

Die vorliegende Arbeit ist bislang keiner anderen Prüfungsbehörde vorgelegt worden. Sie war weder in gleicher noch in ähnlicher Weise Bestandteil einer Prüfungsleistung im bisherigen Studienverlauf und ist auch noch nicht publiziert. Die als Druckschrift eingereichte Fassung der Arbeit ist in allen Teilen identisch mit der elektronisch eingereichten Fassung.

Berlin-Adlershof, den 11.09.2017

Kurzfassung

Das Institut für optische Sensorsysteme stellt mit dem *Integrated Positioning System (IPS)* einen Multisensoransatz zur Lagebestimmung im Raum vor. Das IPS nutzt unter anderem die Informationen aus einem Stereokamerasystem um seine aktuelle Lage zu schätzen. Die Lagebestimmung kann durch die Wiedererkennung von bereits betrachteten Merkmalen in einer Szene verbessert werden. Für die Erkennung und Beschreibung von Merkmalen wurde das Verfahren Oriented FAST and Rotated BRIEF (ORB) im Rahmen einer Evaluierung ausgewählt.

Die aktuelle Prozessierung der Informationen des IPS verarbeitet 10 Bilder pro Sekunde auf einem Prozessrechner mit einer CPU mit integrierter GPU. Für die Verarbeitung wird bisher fast ausschließlich die CPU genutzt. Für eine bessere Lastverteilung auf dem System werden wesentliche Teile der Prozessierung von ORB mit Hilfe von OpenCL auf die GPU ausgelagert. Die vorliegende Arbeit beschreibt die Anpassung der Verarbeitungskette von ORB an die Anforderungen zur Prozessierung auf der GPU und die dazu angefertigte Implementierung.

Zusätzlich ist ein Konzept ausgearbeitet worden, um die Informationen aus der Aufnahmegeometrie des IPS zu nutzen. Zum einen ermöglicht der feste Aufbau des Stereokamerasystems eine Einschränkung des Suchbereichs bei der Zuordnung von Merkmalen in den Stereobildpaaren. Zum anderen können bereits gefundenen Merkmale anhand ihrer Positionsinformation gezielt in Bildern aus anderen Perspektiven gesucht werden. Auf diesem Weg wird die Zuordnung von Features vereinfacht und falsche Zuordnungen vermieden. Die vorgestellten Konzepte können unabhängig von ORB für andere Verfahren zur Erkennung und Beschreibung von Merkmalen verwendet werden.

Die initiale Implementierung erlaubt die Erkennung und Beschreibung von Merkmalen in 30 Bildern pro Sekunde. Dabei wird die CPU zu 5 % ausgelastet, während die GPU auf eine Auslastung von 50 % kommt. Die anschließende Gegenüberstellung mit der CPU-Implementierung von ORB aus der Bildverarbeitungsbibliothek OpenCV hat gezeigt, dass eine Verarbeitung von 100 Bildern in der Sekunde möglich ist.

Abstract

The Institute for Optical Sensor Systems provides a multi-sensor approach for position determination in 3D space, which is called Integrated Positioning System. Among other sensors, IPS uses the information of a stereo camera system to estimate its current position. The estimated position can be improved by recognizing previously observed features in a scene. The method Oriented FAST and Rotated BRIEF was evaluated and selected as the best option to recognize and describe those features.

The current processing of the information from IPS handling 10 images pers second runs on a process computer with a CPU with an integrated GPU. Currently, the processing runs almost exclusively on the CPU. In order to improve the load balance on the system, the processing of ORB is outsourced to the GPU using OpenCL. This paper describes the adaption of the processing chain of ORB to the requirements for processing on the GPU and its implementation.

In addition, a concept has been developed to use the information of the camera geometry of IPS. On the one hand, the fixed structure of the stereo camera system makes it possible to restrict the search area when features are matched in the stereo image pairs. On the other hand, features that have already been found can be recognized in images from other perspectives on the basis of their position information. The presented concepts can be used independently from ORB for other methods for the recognition and description of features.

The initial implementation accomplishes the recognition and description of features in 30 images per second. During this process, the CPU is utilized to only 5 % capacity, while the GPU is well-utilized at 50 % capacity. The subsequent comparison with the CPU implementation of ORB from the image processing library OpenCL showed that it is possible to process even 100 images per second.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

AGAST	Accelerated Segment Test
BRIEF	Binary Robust Independent Elementary Feature
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DLR	Deutsches Zentrum für Luft- und Raumfahrt
DoG	Difference of Gaussian
FAST	Features from Accelerated Segment Test
FLANN	Fast Library for Approximate Nearest Neighbors
FPGA	Field Programmable Gate Array
GPGPU	General Purpose Computation on Graphics Processing Units
GPS	Global Positioning System
GPU	Graphics Processing Unit
HCRM	Harris Corner Response Measurement
IMU	Inertialsensor
IPS	Integrated Positioning System
NMS	Non-Maximum Suppression
OpenGL	Open Graphics Library
OpenCL	Open Computing Language
OpenCV	Open Source Computer Vision Library
OpenVX	OpenVX cross-plattform Vision Acceleration

ORB Oriented FAST and Rotated BRIEF

SDK Software Development Kit

SIFT Scale Invariant Feature Transform

SIMD Single Instruction Multiple Data

1 Einleitung

In vielen Anwendungsbereichen ist eine genaue Lokalisierung des Nutzers innerhalb seiner Umgebung von entscheidender Bedeutung. Das Global Positioning System (GPS) ermöglicht seinen Nutzern weltweit die eigene Position auf wenige Meter genau zu bestimmen, ist allerdings auf die Lokalisierung in Außenbereichen beschränkt [1].

Im Projekt Integrated Positioning System entwickelt das Institut für Optische Sensorsysteme am Deutschen Zentrum für Luft- und Raumfahrt (DLR) einen Multisensoransatz zur Lagebestimmung [2] insbesondere in Innenräumen. Das IPS vereint ein Stereokamerasystem mit einem Inertialsensor (IMU) und lässt sich durch weitere Sensoren wie dem GPS erweitern. Die Vorteile beider Sensoren können so genutzt werden. Eine IMU ist ein Sensor der die Beschleunigung und Rotationsgeschwindigkeit misst und durch Integration der Messwerte eine Schätzung der Bewegung ermöglicht. Messfehler, die unter anderem durch eine Temperaturabhängigkeit des Sensors entstehen, können das Ergebnis verfälschen. Das Stereokamersystem ist ein Messinstrument dessen Informationen die Messungen der IMU berichtigen können. Abbildung 1.1 zeigt einen beispielhaften Aufbau des IPS.

Abbildung 1.1: Beispielhafter Aufbau des IPS.

Das dargestellte System ist mit einer Hand nutzbar und hat an der Frontseite zwei Kameras und zwei Infrarot-Leuchtdioden zur Ausleuchtung der Umgebung. Im Inneren ist eine IMU verbaut. Die Prozessierung der Messdaten und Bilder wird durch einen ange-

schlossenen Rechner durchgeführt. Das Konzept des IPS ist skalierbar und unterstützt unterschiedliche Kameraanordnungen und Inertialsensoren.

Die Anpassbarkeit des Systems erlaubt die Verwendung in verschiedenen Bereichen, in denen eine Selbstlokalisierung notwendig ist. Beispielsweise kann die Bewegung des Systems über die Zeit in einer sogenannten Trajektorie aufgezeichnet werden. Die Abbildung 1.2 zeigt eine solche vom IPS aufgenommene Trajektorie in einem Gebäudekomplex. Die dargestellte Genauigkeit kann mit GPS beispielsweise nicht erreicht werden.

Die genaue Lokalisierung ermöglicht neuartige Anwendungen in zivilen wie auch in industriellen Bereichen. Die Fußgängernavigation benötigt eine genaue Positionierung, um Navigation in Gebäuden oder auf Gehwegen zu ermöglichen. Abstandsmessungen auf Basis der Stereokamera gewinnen zum Beispiel bei autonom fahrende Kraftfahrzeugen oder auch Schienenfahrzeugen an Bedeutung.

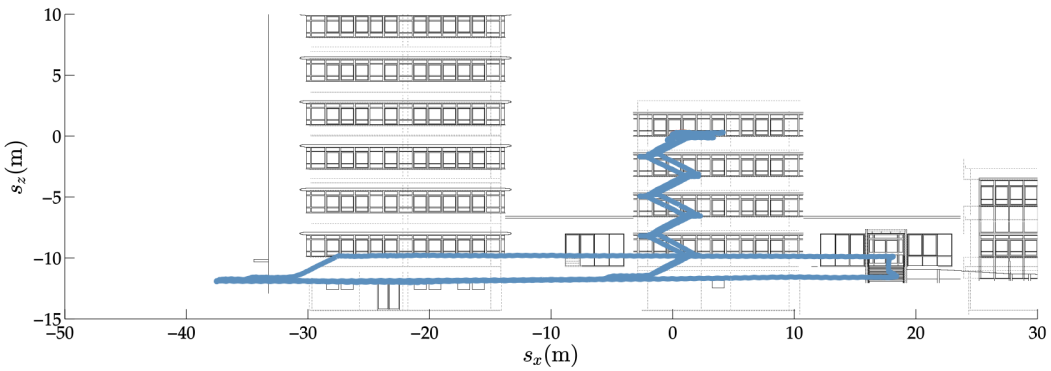


Abbildung 1.2: Aufgenommene Trajektorie des IPS [2, S. 68].

In einigen Szenarien bewegt sich das System in einer bekannten Umgebung oder passiert verschiedene Szenen mehrfach. Durch Erkennung und Beschreibung relevanter Szenen können diese voneinander unterschieden und wiedererkannt werden. Die Relokalisierung anhand dieser Szenen wird als Loop-Closing bezeichnet.

Durch das Loop-Closing können Messungenauigkeiten, die während der Aufnahme der Trajektorie auftreten, erkannt und ausgeglichen werden. Die Genauigkeit der Trajektorie wird dadurch erhöht. Zur Erkennung relevanter Szenen müssen markante Merkmale, im Folgenden Features genannt, im Bild erkannt und beschrieben werden. Diese Arbeit beschreibt einen Ansatz um Features in Bildern zu erkennen und zu beschreiben.

1.1 Motivation

Für die Umsetzung eines effektiven Loop-Closings ist es nötig, möglichst robuste Features in Bildern zu erkennen. Robuste Features zeichnen sich durch Invarianz gegenüber Änderungen der Aufnahmegeometrie sowie der Aufnahmebedingungen aus. Auf Basis einer ausführlichen Evaluierung ist ORB als schnelles und effizientes Verfahren ausgewählt worden, um Features in Bildern zu detektieren und zu beschreiben. ORB liefert skalierungs- und rotationsinvariante Features und beschreibt diese anhand ihrer Umgebung. Eine genaue Beschreibung ist in Kapitel 3 zu finden.

Die aktuelle Verarbeitungskette des IPS nutzt vorrangig die Central Processing Unit (CPU) für Berechnungen. Neuere Entwicklungen lagern Teile der Prozessierung auf die Graphics Processing Unit (GPU) aus. So kann die auf parallele Prozesse ausgelegte Architektur der GPU für die Bildverarbeitung genutzt werden und die Prozessierung beschleunigen. Zusätzlich ermöglicht die Auslagerung auf die GPU eine bessere Lastverteilung auf dem gesamten System. Eine Umsetzung von ORB unter Nutzung der Grafikkarte bedient beide Vorteile. Daraus ergibt sich die folgende Aufgabenstellung.

1.2 Aufgabe

Für das Projekt IPS wird ORB als skalierungs- und rotationsinvariantes Feature-Matching-Verfahren implementiert. Das Verfahren wird mithilfe von Open Computing Language (OpenCL), Open Graphics Library (OpenGL) oder OpenVX cross-plattform Vision Acceleration (OpenVX) durch Nutzung der Grafikkarte beschleunigt. Für die Entwicklung von GPU-beschleunigten Algorithmen wird ein Framework entwickelt. Die ORB-Implementierung nutzt diese.

Der ORB Detector und Descriptor nutzen sowohl CPU als auch GPU, um eine gute Lastverteilung auf dem System zu ermöglichen. Eine geeignete Architektur bildet die erkannten Features und die dazugehörigen Feature Vectors sowie die Schnittstellen zwischen Detector, Descriptor und Matcher ab.

Aus der Bildaufnahmegeometrie des IPS stehen Stereo- und damit Tiefeninformationen zur Verfügung. Diese werden genutzt, um ORB zu erweitern und dadurch eine Verbesserung bei der Zuordnung von Features zu erreichen. Die Features im linken Stereokamerabild werden im Bild der rechten Kamera erkannt. Durch Vorwissen aus der Stereogeometrie können mögliche Zuordnungen vorgefiltert werden. Das entwickelte Ver-

fahren wird als Stereo-Matcher bezeichnet.

Ein ähnlicher Ansatz wird genutzt, um die Featureerkennung aus anderen Perspektiven von bis zu 30° Blickwinkeländerung zu ermöglichen. Dabei liefert das IPS Informationen über die Aufnahmegeometrie. Dies ermöglicht auch hier eine Filterung möglicher Zuordnungen. Das Verfahren wird als Feature-Tracker bezeichnet.

Die Bibliothek Open Source Computer Vision Library (OpenCV) beinhaltet eine optimierte Implementierung von ORB. Gegen diese werden Stereo-Matcher und Feature-Tracker getestet. Die Implementierung soll eine vergleichbare Laufzeit, auf einem System wie im folgenden Abschnitt 1.3 beschrieben, vorweisen. Als Minimalanforderung sollen die Features aus zwei Bildern mit einer Größe von 680×512 Pixeln in unter 100 ms erkannt und beschrieben werden.

1.3 Ausgangssituation

Die Verarbeitungskette des IPS ermöglicht eine vollständige Lokalisierung des Systems anhand der Messdaten aus IMU und Stereokamerasystem. Das IPS erkennt in den Bildern der linken und rechten Kamera Features. Diese Features werden über die Zeit hinweg in beiden Bildern verfolgt. Die Verschiebung der Features zwischen den einzelnen Bildern lässt Rückschlüsse über die Bewegung des Systems zu. Der Vergleich der Bewegung basierend auf den Kameradaten und der Bewegung basierend auf den IMU-Daten erlaubt die genaue Schätzung der tatsächlichen Bewegung. Die Featureerkennung basiert auf dem Accelerated Segment Test (AGAST) Feature Detector und wird auf fortlaufende Bildsequenzen angewendet [3]. Das angewendete Verfahren erlaubt eine stabile Verfolgung von Features, ist allerdings nur bedingt skalierungs- und rotationsinvariant.

Der Prozessrechner zur Verarbeitung der Bild- und Messdaten ist in den meisten Fällen ein Laptop mit einer Spezifikation, die mit der folgenden Hardware vergleichbar ist:

- Prozessor: Intel Core i7 4@2,5 GHz
- Arbeitsspeicher: 8 GB 1,333 MHz
- Grafikkarte: Intel HD 4000

Auf diesem System läuft die beschriebene Prozessierung in Echtzeit mit einer Aufnahme- und Prozessierungsgeschwindigkeit der Bilder von 10 Hz.

1.4 Aufbau der Arbeit

Diese Arbeit beschreibt die GPU-beschleunigte Umsetzung von ORB und dessen Erweiterung. Die erarbeiteten Konzepte und die zum Verständnis benötigten Grundlagen sind beschrieben. Zur Strukturierung ist die Arbeit in die folgenden Kapitel untergliedert.

Kapitel 2 beschreibt die verwendeten Technologien und Frameworks. Dabei wird besonders das Potenzial von Grafikkarten für die parallele Prozessierung herausgestellt.

Im Kapitel 3 sind die mathematischen Grundlagen und Algorithmen der Bildverarbeitung, die in dieser Arbeit Anwendung finden, beschrieben. Für das Verständnis des Konzepts ist grundlegendes Wissen über Stereosysteme und deren Aufnahmegeometrie erforderlich. Die benötigten Grundlagen sind in diesem Kapitel zu finden.

Der konzeptuelle Teil dieser Arbeit ist in Kapitel 4 dargelegt. Sowohl die nötigen Schritte für ORB auf der GPU, als auch die Konzepte des Stereo-Matcher und Feature-Trackers sind ausgearbeitet.

Das 5. Kapitel beschreibt die Umsetzung des in Kapitel 4 beschriebenen Konzepts. Die Architektur und die Anpassung des Verfahrens an eine GPU-gestützte Implementierung stehen im Mittelpunkt. Auf eine Beschreibung der genauen Umsetzung im Code wird verzichtet. In diesem Kapitel befindet sich die Gegenüberstellung der aktuellen Implementierung und der CPU-Implementierung durch OpenCV.

Kapitel 6 liefert eine abschließende Betrachtung der Arbeit. Außerdem beinhaltet es einen Ausblick, der Richtungen für weiterführende Arbeiten aufzeigt.

2 Verwendete Technologien

Dieses Kapitel beschreibt die verwendeten Werkzeuge und Frameworks dieser Arbeit. Neben Grundlagen zur GPU-Programmierung und dazugehöriger Frameworks ist die freie Bibliothek OpenCV erklärt. Die Beschreibung beschränkt sich auf relevante Informationen für das Verständnis dieser Arbeit und erhebt keinen Anspruch auf Vollständigkeit. Als Grundlage dienen vorrangig die Veröffentlichungen [4], [5] und [6].

2.1 Parallelisierung mithilfe von Grafikkarten

Moderne Grafikkarten sind das Ergebnis einer mittlerweile 40 Jahre andauernden Entwicklung einer Hardware für Grafikprozessierung [4]. Die ersten Grafikkarten waren einfache Co-Prozessoren, um Texte und Bilder auf dem Bildschirm darzustellen, ohne den Hauptprozessor mit dem Rendering zu belasten. Schnell entwickelten sich diese weiter und ermöglichten die echtzeitfähige Darstellung von 3D-Modellen. Die dafür ausgelegte Hardware der Grafikkarten unterstützt neben schnellen Fließkommaberechnungen die effiziente Ausführung von Matrix- und Vektoroperationen. Das fest definierte Anwendungsgebiet erlaubte den Aufbau und die Optimierung eines in Hardware umgesetzten Prozessierungsablaufs, der sogenannten Fixed-Function-Pipeline. Diese setzt sich aus dem *Vertex Processing* für 3D-Punkte, der *Rasterization* zur Dreiecksberechnung und dem *Fragment Processing* für Pixelberechnungen zusammen. Sogenannte Shader setzen diese Verarbeitungsschritte um. Diese sind in der *Fixed-Function-Pipeline* fest programmiert und nur durch Parameter anpassbar. 1999 brachte NVidia mit der GeForce 256 eine der ersten Grafikkarten auf den Markt, deren Vertex Shader frei programmierbar ist [5]. Hier entstanden die ersten Ansätze, die Grafikkarte für parallele Prozessierung zu verwenden.

Die Prozessoren der Grafikkarte, auch Shaderkerne genannt, arbeiten nach dem Single Instruction Multiple Data (SIMD)-Ansatz [7]. Die einmal programmierten Shaderprogramme werden auf die Shaderkerne verteilt, die dann dieses Programm auf verschiedenen Eingabedaten ausführen. Dies eröffnet eine vergleichsweise einfache und günstige Möglichkeit zur parallelen Berechnung. Das Bedürfnis und Kapital der Spielindustrie nach weiteren programmierbaren Shadern für realistischeres Rendering treibt die Ent-

wicklung weiter voran. Moderne Grafikkarten bestehen teilweise ausschließlich aus frei programmierbaren Prozessoren, die beim Rendering die jeweiligen Shader übernehmen.

Grafikkarten stellen heutzutage eine effiziente und kostengünstige Hardware zur Beschleunigung verschiedenster Algorithmen dar. Vergleiche haben gezeigt, dass die Implementierungen einfacher umzusetzen sind als Lösungen auf einem Field Programmable Gate Array (FPGA) und dabei eine gleichwertige Optimierung ermöglichen [8]. Zudem ist die Implementierung durch verschiedene Schnittstellen (siehe Abschnitt 2.1.3) hardwareunabhängig.

Der folgende Abschnitt befasst sich mit dem grundlegenden Aufbau der GPU und den Anforderungen um diese für parallelisierte Berechnungen zu nutzen. Zudem sind verschiedene Frameworks vorgestellt, die für die Umsetzung dieser Arbeit in Frage kommen.

2.1.1 Aufbau einer Grafikkarte

Grafikkarten sind ein fester Bestandteil moderner Computersysteme. In den meisten Fällen handelt es sich um Steckkarten, die als Erweiterungen wie zum Beispiel über die *PCIe*-Schnittstelle an das Motherboard des Computers angeschlossen sind (siehe Abbildung 2.2b). Sie übernehmen die Darstellung von Inhalten auf dem Bildschirm und entlasten vermehrt den Hauptprozessor, indem sie Berechnungen übernehmen.

Ein Businterface ermöglicht die Kommunikation mit dem Prozessor, diverse Grafikausgänge die Kommunikation mit angeschlossenen Grafikgeräten. Für die Berechnungen sind ein eigener schneller Speicher mit dazugehörigem Controller und eine Recheneinheit vorgesehen. Die Recheneinheit besteht aus zahlreichen Prozessoren, die bereits als Shaderkerne vorgestellt wurden. Diese grundlegende Architektur wird von den Hardwareherstellern umgesetzt. Der interne Aufbau, insbesondere die Umsetzung der Recheneinheit, ist herstellerspezifisch.

Die folgende Abbildung 2.1 zeigt den grundlegenden internen Aufbau einer Grafikkarte von NVidia mit Fermi-Architektur und verdeutlicht die Optimierung hinsichtlich paralleler Prozessierung. Jeweils 32 Shaderkerne sind in einem Streaming-Multiprocessor zusammengefasst. Jeder Multiprocessor verfügt über einen eigenen Cache für Eingabedaten (L1-Cache) und einen *Texture Cache* für Bilddaten. Der Unterschied zwischen den beiden Cachetypen liegt in den Zugriffen während der Ausführung eines Shadingprogramms. Der L1 Cache ist langsamer, unterstützt dafür aber sowohl Schreib- als

auch Leseoperationen während der Ausführung. Der *Texture Cache* erlaubt hingegen nur jeweils lesende oder schreibende Zugriffe, ist dafür aber schneller. Diese Einschränkung bringt in verschiedenen Anwendungsfällen einen Geschwindigkeitsvorteil, bedarf allerdings einer erneuten Konfiguration der verwendeten Textur, was als „Ping-Pong-Rendering“ bezeichnet wird [9].

Die 16 Streaming Multiprocessors greifen auf einen gemeinsamen L2-Cache zu. Im Vergleich zu herkömmlichen DDR3-RAM ist in aktuellen Grafikkarten GDDR5-RAM verbaut, wodurch diese interne Übertragungsgeschwindigkeiten von über 100 GB/s erreichen können (siehe Abbildung 2.5).

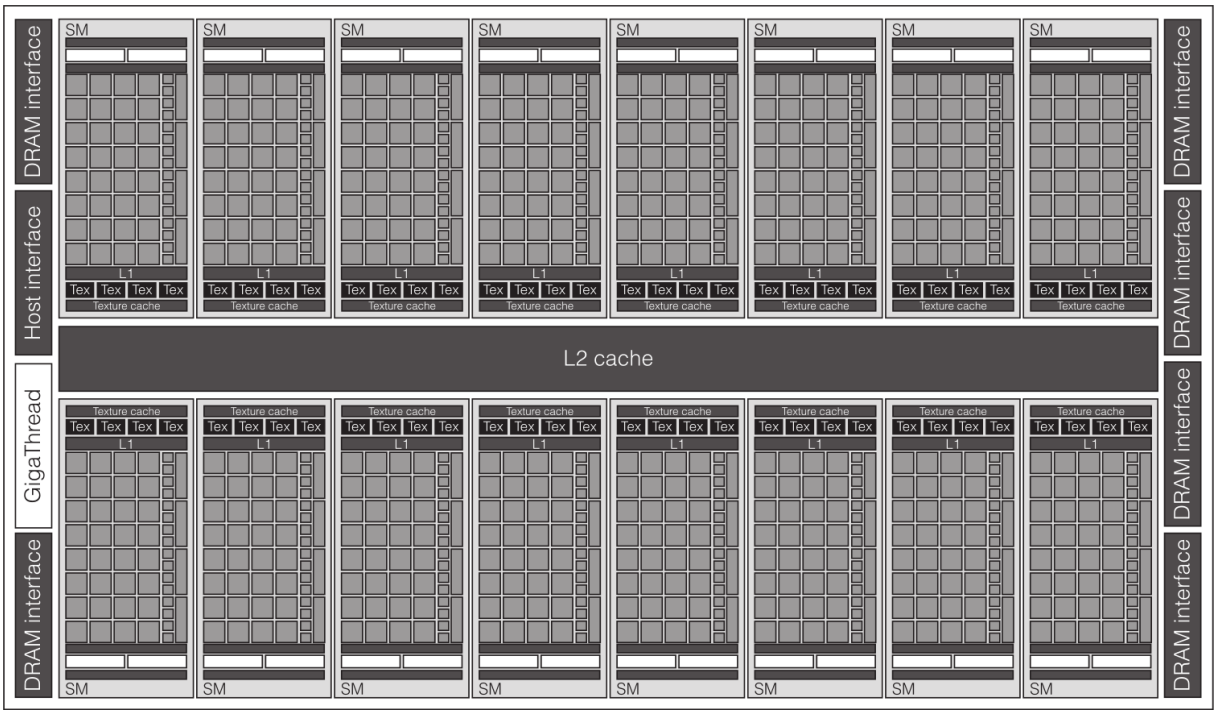
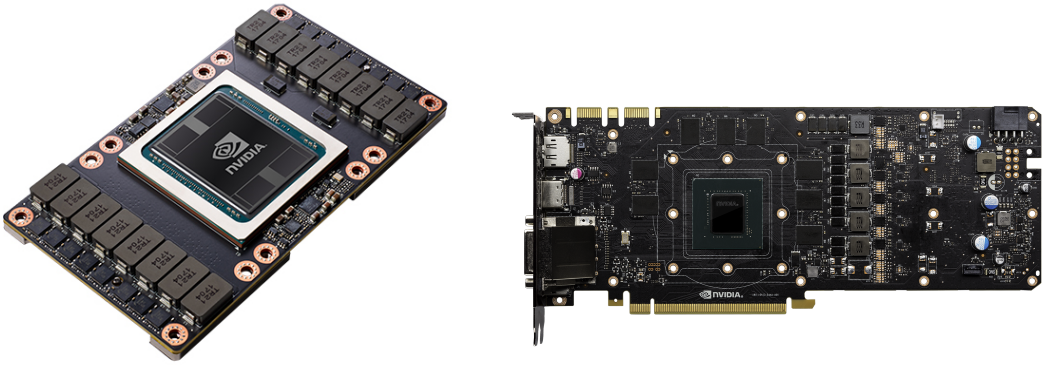


Abbildung 2.1: Interner Aufbau einer GPU mit acht Prozessoreinheiten und sechs Speicherschnittstellen [5, S. 61].

Die Fermi-Architektur besteht ausschließlich aus frei programmierbaren Shaderkernen und erlaubt damit die volle Nutzung der GPU für allgemeine Berechnungen.

Die Abbildung 2.2a zeigt eine NVidia Tesla V100 als Beispiel für eine dedizierte Grafikkarte, die ausschließlich für den General Purpose Computation on Graphics Processing Units (GPGPU)-Bereich konzipiert ist. Sie besitzt keinen Grafikausgang für den Bildschirm, sondern setzt sich nur aus Speicherbausteinen, Prozessoren und einer NVLINK®-

Schnittstelle zur Kommunikation mit der CPU zusammen. Derartige Grafikkarten werden in Rechenzentren genutzt



(a) NVidia Tesla V100 für AI- und Visionbeschleunigung [10].

(b) NVidia GTX 1080 für Grafikrendering [11].

Abbildung 2.2: Beispiele für Grafikkarten.

Das Zielsystem IPS besitzt keine dedizierte Grafikkarte. Der verwendete Prozessor Intel Core i7 vereint als *System on a Chip* die Prozessorkerne und einen integrierten Grafikchip Intel HD 4000. Dieser besteht aus einem Interface mit dem Hauptprozessor, einem Thread Dispatcher zur Threadverwaltung und zahlreichen Shaderkernen, bei Intel Execution Units genannt [12]. Abbildung 2.3 veranschaulicht die Architektur eines Intel Core Prozessors.

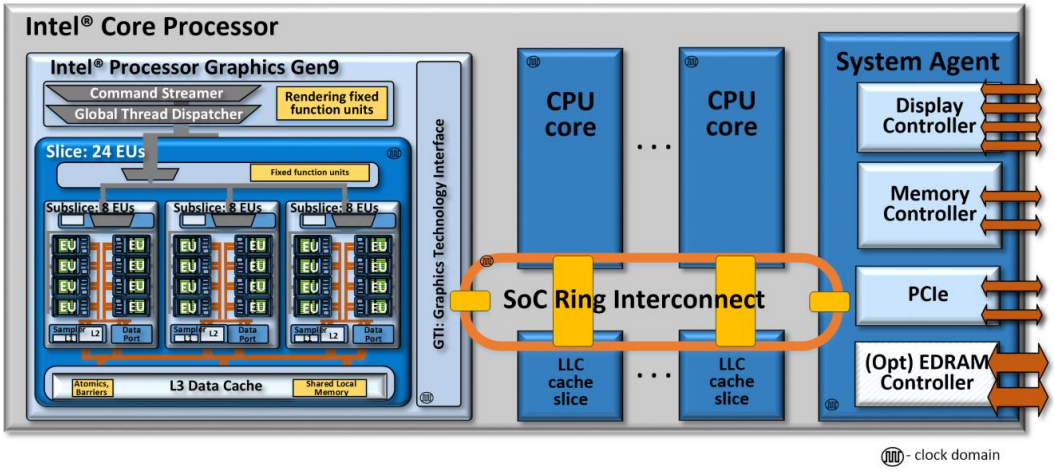


Abbildung 2.3: Core Processor Architektur von Intel [12, S. 3].

Die Intel GPU teilt sich mit der CPU den Arbeitsspeicher. Ein eigener dedizierter

Grafikspeicher ist somit nicht vorhanden. Die neuesten Prozessoren teilen sich zudem mit der Grafikeinheit den Cache, um noch schneller zu kommunizieren. Die Transferrate zwischen GPU und dem Arbeitsspeicher ist langsamer als bei dedizierten Grafikkarten. Integrierte GPUs eignen sich insbesondere aufgrund ihrer geringen Leistungsaufnahme für mobile Systeme. Im Vergleich mit dedizierten Grafikkarten sind sie allerdings weniger leistungsfähig.

Der folgende Abschnitt 2.1.2 zeigt wie die vorgestellte Hardware für Berechnungen genutzt werden kann.

2.1.2 General Purpose Computation on Graphics Processing Units

Der vorangehende Abschnitt stellt den Aufbau von Grafikkarten als effiziente Hardware für parallele Berechnungen zur Darstellung von Bilder auf dem Bildschirm dar. Der folgende Abschnitt befasst sich nun mit den Prinzipien der Nutzung dieser Hardware für allgemeine Berechnungen, der sogenannten General Purpose Computation on Graphics Processing Units (GPGPU).

Der Prozessor und die Grafikkarte sind als getrennte Komponenten beim Aufbau von GPU-beschleunigten Algorithmen zu verstehen. Die CPU wird als Host bezeichnet, die GPU als Device [13].

Die Speicherbereiche von Host und Device sind getrennt und unterliegen jeweils einer eigenen Speicherverwaltung. Die Synchronisation der Speicherbereiche ist Aufgabe des Programmierers - ebenso das Verteilen und Ausführen der GPU-Programme. Ein solches Programm wird bei OpenGL Shader oder auch Shading Program genannt [4], bei OpenCL und OpenVX Kernel.

Ein Kernel ist als eigenständiges Programm zu sehen. Er wird in einer Programmiersprache wie beispielsweise Compute Unified Device Architecture (CUDA), OpenCL oder GLSL geschrieben und der Quellcode im ausführenden Programm als String hinterlegt. Das ausführende Programm übergibt den Quellcode dem jeweiligen Grafikkartentreiber des Systems. Dieser parst den String und kompiliert den Kernel für die entsprechende Hardware. Dieser Schritt ist nötig, da die zugrunde liegende Hardware sich je nach System stark unterscheidet und die Hersteller die Programme unterschiedlich im Maschinencode umsetzen. Die Zeit zum Laden und Kompilieren des Kernels ist bei der Programmierung zu beachten und sollte in zeitunkritischen Momenten geschehen [6].

Ein Kernel lässt sich wie in Abbildung 2.4 abstrakt darstellen.

Für den ausgeführten Kernel müssen Speicherbereiche für die Eingabedaten und Speicherbereiche für die Ausgabedaten definiert werden. Die Eingabedaten müssen vor Ausführung des Kernels auf die GPU übertragen werden. Die Größe des benötigten Ausgabespeichers auf der Grafikkarte wird ebenso vor Ausführung des Kernels definiert.

Der Kernel wird sooft parallel ausgeführt, wie durch den Programmierer vorgegeben, die Speicherbereiche sind dann für jede Instanz verfügbar. Gleichzeitig schreibenden Zugriffen auf gleiche Speicherbereiche sind dabei zu vermeiden.

Beim Start des Kernels werden die Adressen der Eingabe- und Ausgaberegister übergeben. Zusätzliche Daten aus Texturen im Texture Cache stehen zur Verfügung. Texturen ermöglichen die Ablage von großen Datenblöcken, wie zum Beispiel Arrays, im Grafikspeicher. Zusätzlich können Konstanten definiert und eigene Speicher reserviert werden. Der Speicher wird beim Beenden des Kernels wieder freigegeben. Nach der Ausführung des Kernels müssen die Ergebnisse manuell vom Speicher der GPU in den RAM übertragen werden.

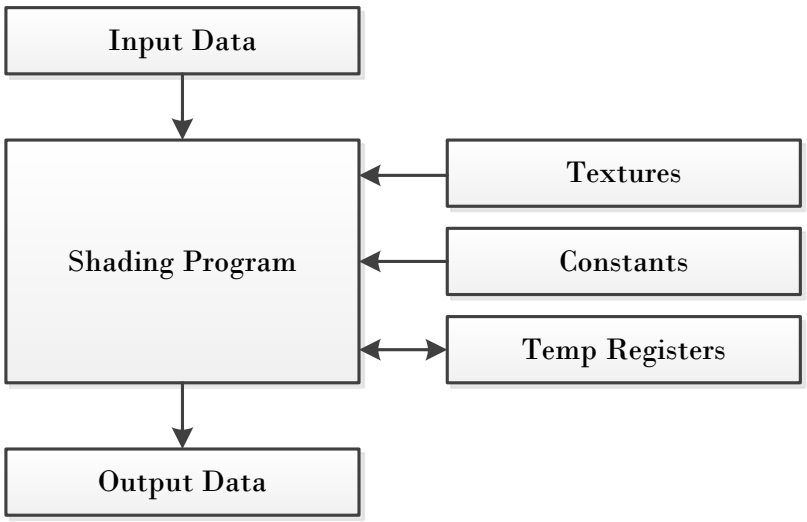


Abbildung 2.4: Abstraktes Shading Program.

Bei GPGPU müssen die Übertragungszeiten zwischen den Hardwarekomponenten beachtet werden. Der Transfer eines Bildes vom Host zum Device und zurück kostet Zeit. In Abbildung 2.5 sind beispielhaft Übertragungsgeschwindigkeiten dargestellt.

Die Übertragungsgeschwindigkeit zwischen dem Arbeitsspeicher und der CPU ergibt sich aus der Taktrate des Speichers multipliziert mit der Adressbreite von 64 Bit (8 Byte).

Herkömmlicher DDR-RAM mit 800 MHz hat durch die doppelte Taktrate dementsprechend eine maximale Übertragungsrate von 12.8 GB/s. In der Grafikkarte können Übertragungsraten von über 100 GB/s erreicht werden. Die Übertragung der Daten zwischen CPU und GPU über beispielsweise den *PCIe*-Bus stellt oftmals einen sogenannten Flaschenhals dar. Neben der Übertragung auf dem Bus kosten die Schreib- und Leseoperationen Zeit. Die Übertragung von einem Bild mit einer Auflösung von 1280×720 Pixeln benötigt ca. 3 ms [13]. Dieselbe Zeit gilt für die Rückübertragung der Ergebnisse. Die Kommunikation zwischen Host und Device sollte minimiert werden, um mögliche Verzögerungen zu vermeiden.

Diese Zeiten müssen beachtet und die Parallelisierung auf der GPU dahingehend geprüft werden. Die Aufhellung eines Bildes wird zum Beispiel durch eine einfache Addition eines Wertes auf den Farbwert eines Pixels umgesetzt. Diese Operation ist auf der CPU schnell ausgeführt. Eine Übertragung auf die Grafikkarte ist nicht sinnvoll.

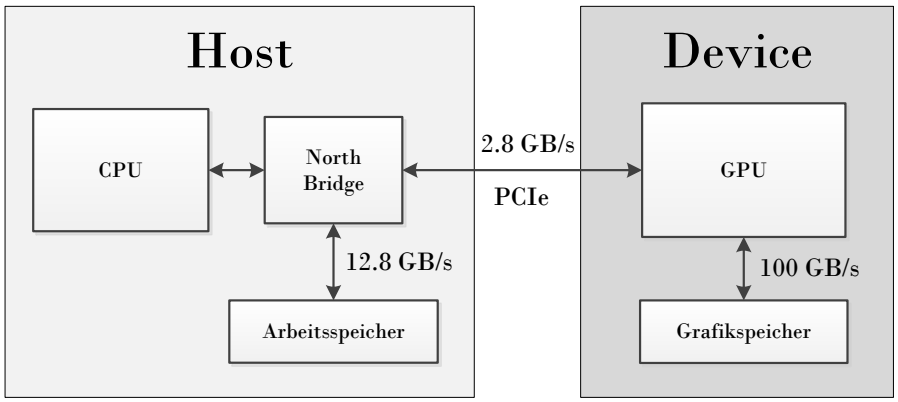


Abbildung 2.5: Übertragungsgeschwindigkeiten zwischen den Hardwarekomponenten [4].

Zur Umsetzung von Befehlen auf der GPU sind zusätzliche Frameworks nötig. Im folgenden Abschnitt 2.1.3 sind einige Frameworks beschrieben.

2.1.3 Frameworks zur GPU-Programmierung

Für die Grafikkarten-gestützte Implementierung stehen die von der Khronos Group definierten Spezifikationen OpenCL, OpenGL und OpenVX zur Auswahl. Die Khronos Group ist ein Konsortium aus verschiedenen Unternehmen der Technologiebranche, mit dem Ziel, offene Standards über verschiedene Plattformen hinweg zu etablieren und zu entwickeln[1]. Zu den Mitgliedern gehören unter anderem die Grafikkarten- und Prozessorhersteller AMD, Intel und NVidia sowie Softwarehersteller wie Apple, Google und Microsoft.

OpenCL, OpenGL und OpenVX sind jeweils eigene Standards im Computing-Bereich. Dies umfasst auch eine Schnittstelle zur Kommunikation mit der Grafikkarte und enthält beispielsweise Funktionen, um Daten im Speicher der Grafikkarte abzulegen, Programme auf den GPU-Kernen auszuführen und Daten wieder in den Arbeitsspeicher zurück zu überführen. Ein Teil der Spezifikation sind Header-Files, die die Schnittstelle in C/C++ definieren. Verschiedene Implementierungen werden durch die Hardwarehersteller bereitgestellt und sind zum Teil in den jeweiligen Treiberpaketen enthalten[2].

Die drei Spezifikationen OpenCL, OpenGL und OpenVX sind jeweils für unterschiedliche Anwendungsgebiete ausgelegt und bieten verschiedene Ansätze zur GPGPU. Die folgenden Abschnitte beschreiben diese Ansätze und erklären die Entscheidung für OpenCL als Schnittstelle für die Implementierung. Die Reihenfolge ist ergibt sich aus dem Jahr der Veröffentlichung der jeweiligen Schnittstelle.

OpenGL

OpenGL ist eine plattformunabhängige Programmierschnittstelle für das Rendering von 2D- und 3D-Grafiken[3]. Die Spezifikation wurde 1992 vom *OpenGL Architecture Review Board* veröffentlicht. Seit 2006 ist die Khronos Group für die Weiterentwicklung der Spezifikation verantwortlich.

OpenGL stellt eine Abstraktionsschicht zwischen der Anwendung und der darunterliegenden Hardware bereit. Es ermöglicht die Darstellung von 3D-Punkten (Vertices) auf einer 2D-Fläche (Fragments) wie dem Bildschirm. Das Rendering wird durch die sogenannte Renderpipeline umgesetzt. Diese besteht aus einem *Vertex-* und einem *Frag-*

[1] `https://www.khronos.org/about/`
[2] `https://developer.nvidia.com/opengl-driver/`
[3] `https://www.khronos.org/opengl/`

mentshader. Neuere Entwicklungen erweitern die Pipeline um *Geometry-*, *Tesselation-* und *Computeshader*.

Der *Vertexshader* nimmt die 3D-Punkte einer Geometrie entgegen, transformiert diese und projiziert sie auf die 2D-Ebene. Der *Fragmentshader* nimmt die Texturen der Geometrie und den Ausgang des *Vertexshaders* entgegen und berechnet auf Basis dieser Daten die Farbe von jedem Pixel im ausgegebenen Bild [14].

Diesen Prozess bildeten die Grafikkarten vor dem Jahr 2000 in Hardware ab. Eine Anpassung dieser *Fixed-Function-Pipeline* war nur begrenzt möglich. Das Aufkommen von programmierbaren Shaderkernen ermöglichte ab 2004 erste Ansätze zum GPGPU [5]. Die Shader dienen zur Parallelisierung der Prozesse. Die *Vertex-* und *Fragmentshader* enthalten den parallelisiert ausgeführten Code. Die Einschränkung auf die Renderpipeline bedingt eine Anpassungen der zu beschleunigenden Algorithmen, die häufig sehr aufwändig ist.

Die Grafikkartenhersteller implementieren den OpenGL-Standard in eigenen Bibliotheken und setzen die Funktionen in den jeweiligen Treibern um. OpenGL ist weit verbreitet und mit wenig Aufwand integrierbar.

OpenCL

OpenCL ist ein plattform- und hardwareunabhängiges Framework zur Programmierung auf heterogener Hardware (Grafikkarten, Mehrkernprozessoren, etc.)[4]. Die Khronos Group veröffentlicht den OpenCL-Standard 2008 und liefert damit eine einheitliche Schnittstelle für die Beschleunigung von Algorithmen [15].

Die Spezifikation legt unter anderem das *memory model*, das *platform model* und das *execution model* fest [16]. Darin sind Schnittstellen zur Auswahl der prozessierenden Hardware, zum Speichermanagement und zur Ausführung von Funktionen definiert. Für den Nutzer ist die Art der Hardware, auf der die Funktionen ausgeführt werden, irrelevant. Die jeweilige Implementierung setzt die Funktionen um.

Die Hardwarehersteller implementieren die Schnittstelle in eigenen Bibliotheken. Die benötigten Bibliotheken sind für die marktführenden Grafikkarten im jeweiligen Standardtreiber enthalten. Die OpenCL-Implementierung führt die in der Spezifikation definierten Funktionen auf der Grafikkarte aus. Eine Anpassung an die Grafikpipeline wie bei OpenGL ist nicht nötig.

[4] `https://www.khronos.org/opencl/`

Khronos bietet zudem eine eigene Bibliothek an, um zur Laufzeit den benötigten Treiber anzusprechen. Diese ist nicht an ein Framework der Hersteller gebunden und ist damit in eigene Bibliotheken integrierbar.

OpenVX

OpenVX ist ein 2014 von der Khronos Group entwickelter Standard, um Prozesse der Computer Vision abzubilden und zu beschleunigen. Es stellt eine plattform- und hardwareunabhängige Schnittstelle zwischen Anwendung und heterogener Hardware dar[5]. Diese heterogene Hardware kann zum Beispiel aus verschiedenen GPUs, CPUs und FPGAs bestehen. OpenVX setzt ein Graphenmodell um und ermöglicht den Aufbau von verarbeitenden Pipelines sowie deren Optimierung. Dabei unterstützt es grundlegende Operationen und Formate und zahlreiche bekannte Computer Vision-Algorithmen, wie Bildpyramiden, Filter und Bildspeicher [17].

Wie bei den vorhergehenden Standards liegt die Implementierung bei den Hardwareherstellern. OpenVX als neue Entwicklung ist durch zahlreiche Frameworks wie NVidias VisionWorks® und Intels Computer Vision SDK umgesetzt. Es existiert allerdings noch keine generische Lösung, die sich wie OpenCL oder OpenGL in eigene Bibliotheken integrieren lässt und dabei auf ein externes Software Development Kit (SDK) verzichtet.

Entscheidung für Framework

Für die Umsetzung von ORB auf der GPU sind grundsätzlich alle drei vorgestellte Frameworks geeignet. OpenGL zeichnet sich durch seine breit gefächerte Verwendung aus. Zahlreiche Veröffentlichungen zeigen, dass Feature Detectors und Descriptors mithilfe von OpenGL beschleunigt werden können. Das Vorgehen durch Nutzung von Shadern ist allerdings veraltet und durch mächtigere Schnittstellen für GPGPU wie CUDA oder OpenCL erreichbar geworden.

OpenVX verspricht ein Konzept, dass sehr gut auf die Verarbeitungskette des IPS passt. Das beschriebene Pipeline-Modell lässt sich in der Theorie gut integrieren. Aufgrund des frühen Entwicklungsstands ist OpenVX allerdings noch nicht für einem Produktivbetrieb im IPS geeignet. Die Plattformunabhängigkeit auf Entwicklerseite ist nicht gegeben.

[5] `https://www.khronos.org/openvx/`

OpenCL bietet die benötigte Plattformunabhängigkeit und ermöglicht eine gute Integration in die bestehenden Bibliotheken des IPS. Aus diesem Grund ist die Implementierung von ORB mit OpenCL umgesetzt.

2.2 OpenCV

OpenCV ist eine plattformunabhängige C/C++-Bibliothek im Bereich Computer Vision[6]. Sie stellt eine Framework für verschiedenste Prozesse der Computer Vision bereit. Dazu gehören unter anderem Bildanalyse und Objekterkennung, Kamerakalibrierung und Stereo-Vision sowie Ansätze für Machine Learning und Robotics [18].

Die quelloffene Bibliothek wurde 1999 veröffentlicht und wird weltweit durch Entwickler in Forschungseinrichtungen und Unternehmen weiterentwickelt. Eine kommerzielle Nutzung ist aufgrund der BSD-Lizenz möglich. Die Implementierungen in OpenCV sind für echtzeitfähige Anwendungen konzipiert und dementsprechen optimiert. Teilweise existieren zusätzliche Implementierungen die mit CUDA oder OpenCL unter Nutzung der GPU beschleunigt werden.

Durch sein breites Anwendungsspektrum ist OpenCV weitverbreitet und repräsentiert für viele Bereiche den Stand der Technik. OpenCV beinhaltet den ORB Feature Detector und Descriptor sowie eine Sammlung von Algorithmen zur Zuordnung von Features.

[6] `http://opencv.org/`

3 Wissenschaftlicher Hintergrund

Diese Arbeit ist im Bereich der Photogrammetrie angesiedelt und basiert auf dem Zusammenspiel verschiedenster Verfahren aus der Computer Vision. In diesem Kapitel sind die wissenschaftlichen Grundlagen für das Verständnis dieser Arbeit beschrieben. Es führt grundlegende Fachbegriffe ein, beschreibt die abstrakte Betrachtung von optischen Messinstrumenten und deren Bildaufnahmegeometrie sowie grundlegende Verfahren im Umgang mit Features.

Der erste Teil betrachtet Features in Bildern und Verfahren zu deren Erkennung und Beschreibung im Allgemeinen. ORB stellt ein solches Verfahren zur Erkennung und Beschreibung um. Dafür sind grundlegende Herangehensweisen im Umgang mit Features nötig, diese sind beschrieben. Auf Basis dieser Beschreibung wird ORB im Anschluss vorgestellt.

Der Abschnitt 3.4 stellt grundlegende Betrachtungen im Bereich der Bildaufnahmegeometrie vor. Im Anschluss stellt der Abschnitt 3.5 weitere Arbeiten vor, die ähnliche Ansätze verfolgen wie die vorliegende Arbeit.

3.1 Grundlegende Begriffe

Für die weitere Verwendung in der Arbeit erläutert dieser Abschnitt grundlegend Features und deren Erkennung, Beschreibung und Zuordnung. Die Erklärungen und Darstellungen orientieren sich an [19, S. 13-39] und [20].

Ein Feature ist ein lokales Merkmal in einem Bild, dass sich von seiner direkten Umgebung unterscheidet. Verfahren zur Erkennung von Features werden als Detector bezeichnet, Verfahren zur Beschreibung als Descriptor. Die Zuordnung von Features aus zwei Bildern auf Basis der Beschreibung des Descriptors geschieht im Matcher. Die korrespondierenden Features zweier Bilder werden Matches genannt. Die folgende Abbildung 3.1 zeigt das Zusammenspiel von Detector, Descriptor und Matcher.

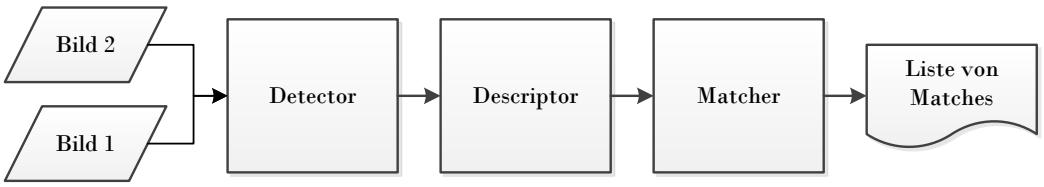


Abbildung 3.1: Verallgemeinerter Ablauf beim Feature-Matching.

3.1.1 Feature

Für die Verarbeitung von Bildern in der Computer Vision ist es nötig, Bilder und deren Inhalt möglichst genau und unterscheidbar zu repräsentieren. Eine mögliche Repräsentation ist die Beschreibung von Features beziehungsweise Features, die das Bild ausmachen. Dabei ist grundlegend zwischen globalen und lokalen Features zu unterscheiden.

Globale Features beschreiben ein Bild als Ganzes. Das kann zum Beispiel mithilfe eines Histogramms oder durch die Informationen aus Filterfunktionen über das Bild geschehen. Die Features werden in einem eindimensionalen Feature Vector zusammengefasst. Diese sind oft gut zu vergleichen und schnell zu verarbeiten. Auf diese Weise können Bilder über große Datenmengen hinweg eindeutig identifiziert und indiziert werden.

Eine genauere Betrachtung ermöglichen lokale Features. Die Repräsentation eines Bildes setzt sich hier aus zahlreichen Beschreibungen von Strukturen im Bild zusammen. Nach Tuytelaars [20, S. 2] sind lokale Features Bildmuster, die sich von ihrer Nachbarschaft unterscheiden. Die Unterscheidung von der Nachbarschaft kann dabei auf unterschiedliche Weise gelingen. In jedem Fall sollten die gefundenen Features die folgenden Eigenschaften besitzen:

- *Wiederholbarkeit und Robustheit*: Ein Feature wird auch unter verschiedenen Aufnahmebedingungen erkannt. Änderungen in der Belichtung genauso wie die Veränderung des Betrachtungswinkels und der -position wirken sich kaum auf die Erkennung aus.
- *Unterscheidbarkeit*: Das Feature besitzt Eigenschaften, die es von anderen Features abgrenzt.
- *Anzahl*: Oft ist eine große Anzahl gefundener Features von Vorteil.
- *Genauigkeit*: Die Position des Features ist im Bezug auf Skalierung und Form des Features genau.
- *Effizienz*: Die Erkennung des Features ist in angemessener Zeit möglich.

Für die weiteren Betrachtungen in dieser Arbeit sind ausschließlich lokale Features relevant.

3.1.2 Detector

Der Feature Detector erkennt im Bild Features, mit den oben beschriebenen Eigenschaften. Die Herausforderung besteht darin, Features auch unter stark veränderlichen Aufnahmebedingungen zu erkennen. Kontrast- und Belichtungsänderungen, erhöhtes Rauschen sowie Änderungen in der Aufnahmegeometrie wie zum Beispiel unterschiedliche Aufnahmewinkel oder Positionswechsel bei der Bildaufnahme erschweren die Erkennung.

Ein Detector ist entweder single-scale oder multi-scale. Ein single-scale Detector erkennt dieselben Features in einem Bild auch nach Rotation und Translation des Bildes. Eine Skalierung des Bildes führt allerdings dazu, dass die meisten Features nicht mehr erkannt werden. Eine Skalierung im Bild kann zum Beispiel durch Herangehen an das aufgenommene Objekt stattfinden. Ein multi-scale Detector wird verwendet um dieses Problem zu lösen. Dafür ist eine Betrachtung der Features anhand ihrer Unterscheidbarkeit in unterschiedlichen Skalierungsstufen notwendig.

3.1.3 Descriptor

Nach der Detektion der Features beschreibt der Descriptor jedes Feature anhand seiner lokalen Umgebung. Das Ergebnis ist ein sogenannter Feature Vector, der die Beschreibung von einem Feature zusammenfasst. Dieser kann beliebig lang sein. Anhand seiner Länge n spricht man von einem n-dimensionalen Feature Vector. Je länger der Vector, desto genauer können Features beschrieben werden. Allerdings braucht der Vergleich zweiter Feature Vectors auch entsprechend mehr Zeit, was die Performance des Matchers beeinflusst.

Es wird zwischen reellen und binären Feature Vectors unterschieden. Ein reeller Feature Vector setzt sich aus n reellen Werten zusammen. Diese können beispielsweise lokale Bildgradienten abbilden. Binäre Feature Vectors bestehen hingegen nur aus zwei Werten. Durch die Einschränkung der Dimensionen auf jeweils zwei Elemente sind die Feature Vectors schneller zu vergleichen, aber schwerer voneinander zu unterscheiden.

3.1.4 Matching

Der Feature Matcher empfängt Listen von Feature Vectors und sucht darin passende Zuordnungen. Das Ziel liegt darin möglichst viele sogenannte Matches zu finden und falsche Matches zu vermeiden.

Ein einfacher Ansatz für das Matching ist die Brute-Force-Methode, bei der jeder Feature Vector aus dem ersten Bild mit allen Feature Vectors aus dem zweiten Bild verglichen wird. Eine Distanzfunktion liefert den Abstand der Feature Vectors im n-dimensionalen Raum. Der geringste Abstand in n Dimensionen entspricht dann dem Match. Sehr lange reelle Feature Vectors benötigen für diesen Schritt viel Zeit. Aus diesem Grund sind vorhergehende Filter sinnvoll. Matcher für binäre Feature Vectors nutzen häufig die Hamming-Distanz als Maß für die Ähnlichkeit zweier Feature Vectors. Die eingeschränkte Größe der Dimensionen beschleunigt die Ermittlung der Distanz.

Das sogenannte *Locality Sensitive Hashing*, ein Nearest Neighbor Ansatz, beschleunigt das Matching von binären Feature Vectors. Bei diesem Ansatz werden die Vectors auf Hash Tables verteilt, wobei ähnliche Vectors zusammengefasst werden. Von den Hash Tables werden die Hashes gebildet und abgespeichert. Bei binären Feature Vectors entspricht die Hashfunktion einer Untermenge der einzelnen Bits im Feature Vector. Die Generierung ist dementsprechend einfach.

Beim Matching werden die Hashes der Feature Vectors beider Bilder verglichen, somit ist nur noch der Vergleich einer Teilmenge der gesamten Feature Vectors nötig. Die OpenCV-Bibliothek Fast Library for Approximate Nearest Neighbors (FLANN) setzt ein solches Matching um.

3.2 Verfahren im Umgang mit Features

Der ORB Detector und Descriptor nutzen verschiedene gängige Verfahren der Bildverarbeitung, um möglichst robuste Features zu erkennen und zu beschreiben. Die Verfahren sind im folgenden Abschnitt beschrieben. Die Reihenfolge richtet sich nach dem Vorkommen bei ORB (siehe Abschnitt 3.3).

3.2.1 Non-Maximum Suppression

Non-Maximum Suppression (NMS) ist ein Verfahren zur Suche von lokalen Extrema [21]. Es findet in vielen verschiedenen Computer-Vision Algorithmen Anwendung - unter anderem in verschiedenen Verfahren zur Erkennung von Kanten und Features.

Das Ziel eines Detectors ist eine Bildkoordinate zu ermitteln, die die Position des Features im Bild am besten beschreibt. Dazu untersuchen der Detector häufig jeden Pixel in der betrachteten Umgebung. Nach der Erkennung stehen teilweise mehrere Pixel als mögliche Koordinaten zur Auswahl. Die Abbildung 3.2a zeigt das Ergebnis einer Kantenerkennung. Die Kanten werden teilweise als mehrere Pixel breite Streifen detektiert.

NMS untersucht diese Auswahl und ermittelt anhand der umliegenden Kandidaten eine alleinstehende Bildkoordinate. Dazu bewertet NMS jeden Pixel anhand einer Bewertungsfunktion. Diese richtet sich nach dem genutzten Detector. Durch den Vergleich dieser Bewertung sucht NMS das lokale Maximum. Die Abbildung 3.2b zeigt das Ergebnis nach angewendeter NMS. Die Kanten sind als einen Pixel breite Linien erkannt.

Die Bewertungsfunktion kann für jeden Pixel unabhängig ausgeführt werden. Aus diesem Grund eignet sich NMS für eine Implementierung auf der Grafikkarte [21].

(a) Kantenerkennung. (b) Non-Maximum Suppression.

Abbildung 3.2: Beispiel für Non-Maximum Suppression anhand einer Liniendetektion [22].

3.2.2 Harris Corner Response Measurement

Harris Corner Response Measurement (HCRM) entstammt dem Harris Corner Detector und bewertet einen Pixel anhand seiner Umgebung. Jeder Pixel erhält basierend auf der Umgebung eine Bewertung wie sehr er einer Ecke entspricht.

Die Betrachtung der Umgebung eines Pixels anhand der Änderung zur umliegenden Region liefert Rückschlüsse über die abgebildete Form im Bild [23]. Flächen haben nur geringe Änderungen, Kanten Änderungen in eine Richtung, Ecken haben Änderungen in mehrere Richtungen. Der Grad der Änderung lässt sich durch Ableitung des Bildes in x- und y-Richtung berechnen. Aus dieser Betrachtung heraus verfasst Harris folgende Aussage:

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \tag{3.1}$$

Die Summe der Produkte der Ableitungen ergeben die Matrix M. Die Eigenwertzerlegung der Matrix M liefert die Eigenwerte λ_1 und λ_2. Die Ableitung betrachtet dazu den Verlauf der Intensität I in x- und in y-Richtung des Bildes an der Stelle des betrachteten Pixels. Die Intensitäten in der Umgebung dieses Pixels werden betrachtet.

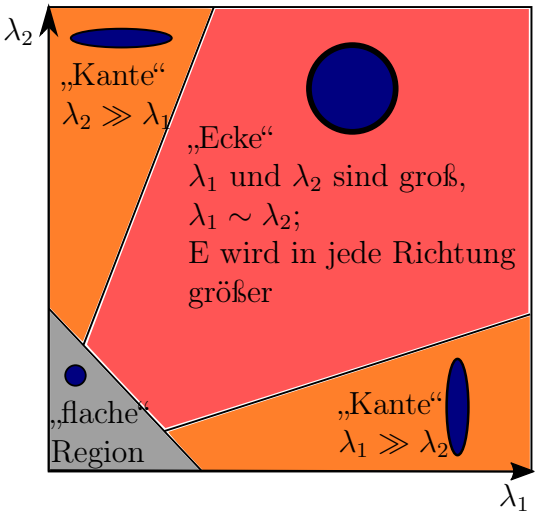


Abbildung 3.3: Harrisregionen anhand der Werte von λ_1 und λ_2.

Werden die Eigenwerte zu einer Ellipse aufgespannt, ermöglicht die Betrachtung der Form und Größe dieser Ellipse eine Einordnung des Pixels. Eine schmale, langgestreckte Ellipse ergibt sich aus einem kleinen und einem großen λ, was einer Kante entspricht.

Sind beide Werte groß, so ist der Punkt eine Ecke. Die Abbildung 3.3 veranschaulicht diese Einordnung.

Die Matrix M lässt zusätzlich Rückschlüsse auf den Grad der Unterscheidbarkeit der Ecke zu. Die Harris Corner Response R ergibt sich aus der Determinante von M, abzüglich des um den Faktor k gestauchten Quadrats der Spur von M. k ist eine empirisch festgelegte Konstante mit einem Wert zwischen 0.04 und 0.06.

$$R = \det M - k(\operatorname{trace} M)^2 \tag{3.2}$$

Die beiden Eigenwerte können zur Berechnung der Determinante und der Spur genutzt werden.

$$\det M = \lambda_1 \lambda_2$$

$$\operatorname{trace} M = \lambda_1 + \lambda_2$$

3.2.3 Bildpyramide

Ein in der Nähe gefundenes Feature ist aufgrund des geringen Abstands zur Kamera hoch aufgelöst. Dasselbe Feature aus weiterer Entfernung betrachtet ist dagegen geringer aufgelöst. Um dennoch in beiden Fällen das Feature zu erkennen ist Wissen über die Skalierung des Features nötig. Eine Unterabtastung mit kleinerer Auflösung liefert diese Informationen. Der Grundgedanke dahinter ist die konsequente Ausnutzung des Abtasttheorems [24].

Das unterabgetastete Bild enthält Informationen über ein Objekt als wäre es aus größerer Entfernung aufgenommen. Eine Bildpyramide enthält mehrere unterschiedlich unterabgetastete Bilder. Die Bildpyramide ist die Repräsentation eines Bildes in verschiedenen Auflösungen. Die unterste Stufe entspricht dem voll aufgelösten Bild. Die jeweils darüber liegende Ebene hat die halbe Auflösung. Zur Generierung der jeweils nächsten Stufe wird die Ausgangsstufe mit einem Gaußschen Weichzeichner mit festgelegtem Sigma weichgezeichnet und auf die Hälfte skaliert. Die Abbildung 3.4 stellt die Bildpyramide graphisch dar.

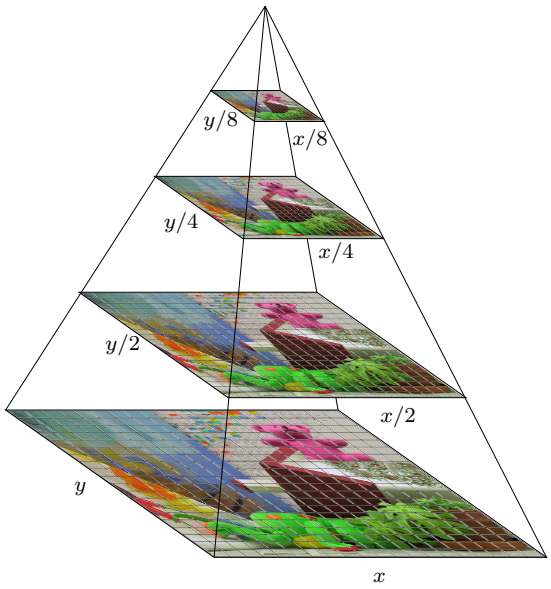


Abbildung 3.4: Darstellung einer Bildpyramide.

3.2.4 Intensity Centroid

Der Intensity Centroid erlaubt es die Orientierung einer Ecke anhand der Intensitäten der umliegenden Pixel zu bestimmen [25]. Der Mittelpunkt der Intensitäten um einen Punkt wird als Centroid bezeichnet. Das Verfahren basiert auf der Annahme, dass der Centroid einen Offset zum Zentrum der Ecke aufweisen. Zwischen dem Zentrum der Ecke und dem Centroid kann ein Vektor aufgespannt werden, dessen Betrag und Winkel eine Klassifizierung ermöglicht. Die folgende Abbildung 3.5 veranschaulicht den Intensity Centroid.

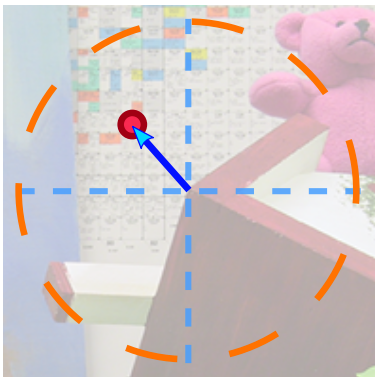


Abbildung 3.5: Intensity Centroid am Beispiel.

Die Orientierung ergibt sich aus der Betrachtung der Position des Moments der Intensitäten um die Ecke relativ zur Lage der Ecke und lässt sich wie folgt berechnen.

$$m_{pq} = \sum_{x,y} x^p y^q I(x,y) \tag{3.3}$$

Die Betrachtung der umliegenden Pixel um das Feature liefert das Moment. Dazu wird die Intensität I der umliegenden Pixel aufsummiert. Die Position des Centroid C ergibt sich direkt aus dem Moment.

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right)$$

Die Orientierung ergibt sich nun aus dem aufgespannten Vector.

$$\theta = \text{atan2}\left(m_{01}, m_{10}\right)$$

Die Orientierung von jedem Feature wird aufgenommen und ermöglicht die korrekte Positionierung des Descriptors.

3.3 Oriented FAST and Rotated BRIEF

ORB beschreibt ein Verfahren, um Features zu erkennen und zu beschreiben. Es wurde im Jahr 2011 von Rublee et al. im Incubator WillowGarage entwickelt. Das Verfahren basiert auf dem Detector Features from Accelerated Segment Test (FAST) und dem Descriptor Binary Robust Independent Elementary Feature (BRIEF) [26] und zeichnet sich durch seine Skalierungs- und Perspektivinvarianz aus. ORB benötigt mit 150 ms im Vergleich zu gängigen Verfahren wenig Rechenzeit.

Dieser Abschnitt beschreibt den ORB Feature Detector und den Feature Descriptor wie sie in der Arbeit von Rublee et al. [26] veröffentlicht sind.

3.3.1 ORB Detector

Der ORB Detector basiert auf dem FAST Feature Detector nach E. Rosten. Dieser betrachtet jeden Pixel im Bild und vergleicht ihn mit den umliegenden Pixeln, um Ecken zu extrahieren. Durch die einfachen Vergleichsoperationen ist FAST sehr schnell. Im Gegenzug ist der Detector nicht skalierungsinvariant und anfällig für Rauschen [19].

Die Abbildung 3.6 veranschaulicht das Vorgehen. Der Detector legt das dargestellte Pattern auf jeden Pixel p des Bildes und vergleicht die Intensität I des Pixels p mit 16 umliegenden Pixeln. Diese liegen auf einem Bresenham-Kreis mit dem Radius 3[27].

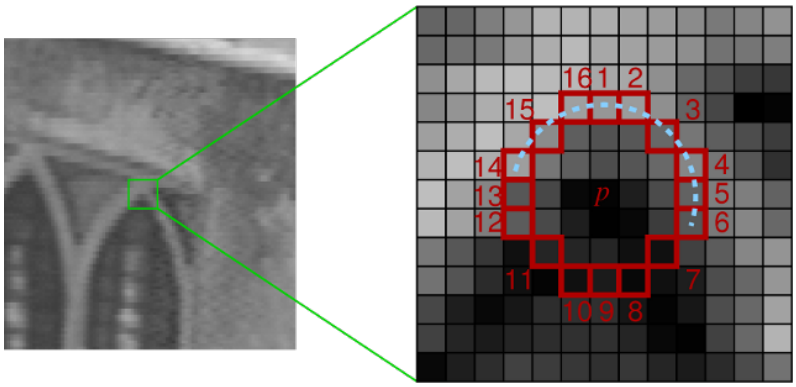


Abbildung 3.6: FAST Feature Detector [28].

Von den 16 Pixeln im Vergleich müssen N auf einem durchgehenden Kreisabschnitt liegende Pixel entweder allesamt um t heller oder dunkler sein als der Pixel p. Die Anzahl der gefundenen Pixel, in der Formel (3.4) als C bezeichnet, muss größer sein als ein Schwellwert N. Dieser ist anpassbar.

$$C(x,y) = max(\sum_{j \in S_{bright}} |I_{p \to j} - I_p| - t, \sum_{j \in S_{dark}} |I_p - I_{p \to j}| - t) \tag{3.4}$$

Sind mindestens N Pixel auf dem Kreisabschnitt heller oder dunkler als p, so gilt p als Feature. Über den Schwellwert t kann die Anzahl der gefundenen Features variiert werden. Eine Anzahl von 9 Pixeln für die Bewertung als Feature liefert die besten Ergebnisse [19]. In diesem Fall wird von FAST-9 gesprochen.

Zur Beschleunigung vergleicht FAST erst den Pixel p mit den Pixeln 1, 5, 9 und 13. Mindestens drei der Pixel müssen eine Intensität über oder unter dem Vergleichspixel inklusive des Schwellwerts besitzen. Ist dies nicht der Fall, so kann die Betrachtung abgebrochen werden, da ein kontinuierlicher Kreisbogen von mindestens 9 Pixeln mit der entsprechenden Eigenschaft nicht mehr möglich ist.

FAST produziert je nach eingestellten Parametern sehr viele als Features geltende Pixel. Diese liegen oft nah beieinander und markieren dasselbe Feature. *Non-maximum suppression* liefert aus den zusammenhängenden Pixeln eine zentrale Position für das Feature und ermöglicht eine genauere Lokalisierung.

Im nächsten Schritt sortiert ORB die Features mithilfe des HCRM und wählt die besten Features aus. Die Anzahl an Features kann durch einen Parameter festgelegt werden. Das Ergebnis ist eine Liste von repräsentativen Features im Bild, die an den Descriptor weitergegeben wird.

Problematisch ist die Skalierungsabhängigkeit von FAST. Um dieser entgegenzuwirken, wird der Detector auf eine Bildpyramide des eingegebenen Bildes angewendet (siehe Abbildung 3.2.3). Daraus resultieren erkannte Features auf verschiedenen Skalierungen des Bildes. Die Information über die Skalierung ist Teil der gefundenen Features.

3.3.2 ORB Descriptor

Der Descriptor von ORB basiert auf dem BRIEF Descriptor entwickelt von M. Calonder et al. an der technisch-naturwissenschaftlichen Universität Lausanne [29]. Bei diesem Descriptor handelt es sich um einen binären Descriptor mit einem resultierenden Feature Vector aus 256 Binärwerten. Diese 256 Werte ergeben sich aus 256 Vergleichen von jeweils 2 unterschiedlichen Pixeln rund um das Feature, deren Ergebnis jeweils 1 oder 0 ist. Die Pixel rund um das Feature werden über ein festgelegtes Aufnahmemuster ausgewählt.

Die Vergleichsfunktion, die die Werte des Feature Vectors setzt, betrachtet die Intensität der ausgewählten Pixel. Ist der erste Pixel heller als der zweite, so wird eine 1 in den Feature Vector geschrieben, ist der dunkler eine 0. Die Reihenfolge der Pixel ergibt sich aus dem Aufnahmemuster. Auf diese Weise wird die Struktur um das Feature vom Descriptor beschrieben. Die Testfunktion τ beschreibt diesen Vergleich. Um die Empfindlichkeit gegenüber Rauschen in der Bildaufnahme zu verringern wird die Region um das Feature weichgezeichnet.

$$\tau(p; x, y) := \begin{cases} 1 & \text{if}(p(x) < p(y)) \\ 0 & \text{otherwise} \end{cases} \tag{3.5}$$

Für das Aufnahmemuster stehen nach der Arbeit von Calonder et al. verschiedene Verteilungen zur Verfügung. Die Abbildung 3.7 zeigt eine statische Normalverteilung der zu vergleichenden Punkte. Im Vergleich mit gleich verteilten Punkten oder auch einem fest definierten symmetrischen Muster liefert dieses die stabilsten Features [29].

BRIEF ist nicht per se rotationsinvariant. Für jedes gefundene Feature wird mithilfe des Intensity Centroids (siehe Abschnitt 3.2.4) die Orientierung berechnet. Um diese Orientierung wird das Aufnahmemuster gedreht und dann auf das Feature gelegt.

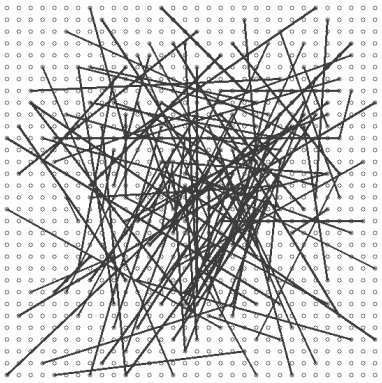


Abbildung 3.7: Normalverteiltes BRIEF Pattern [29, S. 5].

3.4 Grundlagen der Bildgeometrie

Dieser Abschnitt beschreibt die theoretische Betrachtung optischer Messinstrumente. Die Projektion von 3D-Punkten im Raum auf eine 2D-Ebene wie einem Kamerasensor lässt sich durch das Lochkameramodell betrachten. Für das Verständnis des Konzepts in Kapitel 4 ist insbesondere Wissen über die Epipolargeometrie relevant, welche auf dem Lochkameramodell und der Aufnahmegeometrie zweier Kameras beruht.

Die Abbildungen sind von O. Schreer [30] und von R. Fritzsche [31] übernommen beziehungsweise orientieren sich daran.

3.4.1 Das Lochkameramodell

Das Lochkameramodell dient als Grundlage für die theoretische Betrachtung der projektiven Geometrie, wie sie durch Kameras umgesetzt ist. Das Prinzip der *Camera Obscura*, auch Lochkamera genannt, hat diesem Modell seinen Namen gegeben. Die folgende Abbildung 3.8 veranschaulicht das Prinzip.
Ein 3D-Punkt M im Raum wird über den Brennpunkt, der das optische Zentrum darstellt, auf die Bildebene projiziert. Der resultierende Bildpunkt m ist punktsymmetrisch gespiegelt. Zur einfacheren Betrachtung kann die Bildebene auch ungespiegelt vor dem Brennpunkt dargestellt werden. Dies hat keine Konsequenzen auf die weiteren mathematischen Betrachtungen [31].

Daraus ergibt sich die folgende schematische Betrachtung einer Kamera 3.9. C ist der Brennpunkt und gilt als Ursprung des Kamerakoordinatensystems. Eine zwischen Brennpunkt und Bildebene aufgespannte Normale hat ihren Schnittpunkt mit der Bil-

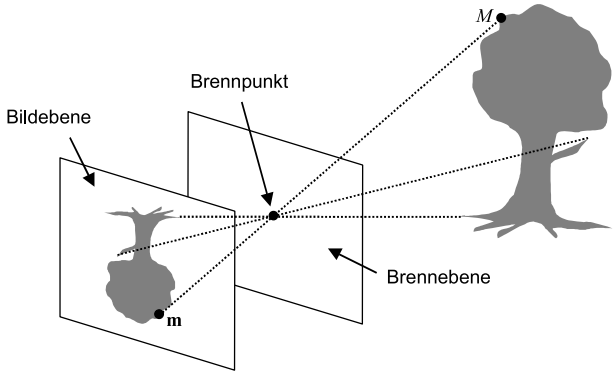


Abbildung 3.8: Projektion mit dem Lochkameramodell [30, S. 41].

debene im Punkt c, der auch als Bildhauptpunkt bezeichnet wird. Dieser Punkt dient als Koordinatenursprung für das Bildkoordinatensystem.

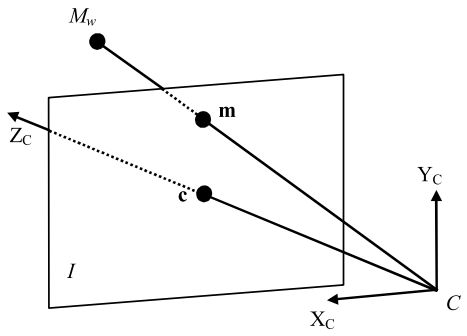


Abbildung 3.9: Grundlegender Aufbau des Lochkameramodells [30, S. 41].

Die Projektion nach dem Lochkameramodell wird von der folgenden Gleichung 3.6 umgesetzt. Es fällt auf, dass bei der Transformation die Tiefeninformation Z verloren geht. Diese kann nicht ohne zusätzliche Informationen wieder hergestellt werden.

$$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} \tag{3.6}$$

3.4.2 Stereogeometrie

Zwei auf einer Achse befestigte Kameras werden als Stereosystem bezeichnet (siehe Abbildung 3.10). Stereokameras stellen eine Möglichkeit zur Berechnung von Tiefeninforma-

tionen dar. Ähnlich wie beim menschlichen Sehen kann über den bekannten Abstand der Augen, beziehungsweise Kameras, die Entfernung vom Betrachter zu den betrachteten Objektpunkten berechnet werden.

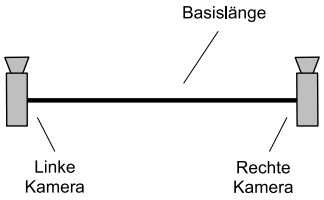


Abbildung 3.10: Grundlegender Aufbau einer Stereokamera [30, S. 66].

Im Idealfall sind die Kameras exakt achsparallel ausgerichtet, so dass die Kameraachsen parallel liegen und die Pixel gleicher Objektpunkte in beiden Kamerabildern nur eine Verschiebung in x-Richtung aufweisen. Die Abbildung 3.11 skizziert einen solchen achsparallelen Aufbau.

mithilfe von Triangulation durch die Punkte m_1 und m_2 kann durch die bekannte Basislänge B der Abstand ρ berechnet werden.

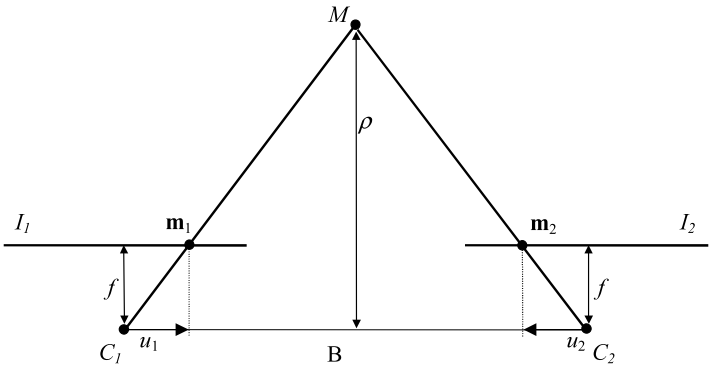


Abbildung 3.11: Draufsicht auf eine achsparallele Stereogeometrie [30, S. 67].

3.4.3 Epipolargeometrie

Die Epipolargeometrie beschreibt die grundlegende Geometrie zwischen zwei Kameras [2]. Die achsparallele Stereogeometrie ist ein Spezialfall der Epipolargeometrie.

Im Folgenden ist angenommen, dass beide Kameras denselben Objektpunkt M betrachten (siehe Abbildung 3.12). Spannt man ein Dreieck zwischen dem Objektpunkt M und den jeweiligen Kamerapositionen C_1 und C_2, so ergibt sich zwischen C_1 und C_2

jeweils ein Schnittpunkt e_1 und e_2 mit den jeweiligen Bildebenen der Kameras. Eine Gerade zwischen dem Schnittpunkt e und dem jeweiligen Bildpunkt m wird als Epipolarlinie bezeichnet. Alle Punkte, die bei C_2 auf dieser Linie liegen, sind bei C_1 im Bildpunkt m_1. Alle Linien zusammengefasst werden Epipolarbüschel genannt.

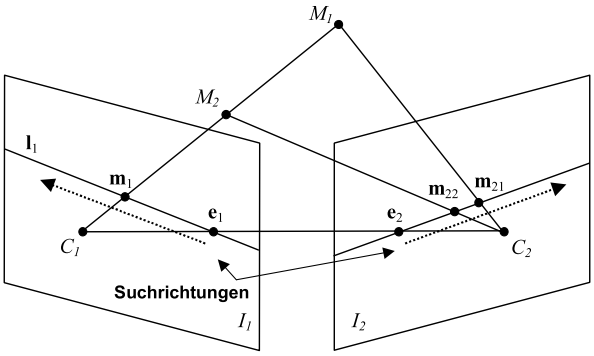


Abbildung 3.12: Prinzip der Epipolargeometrie [30, S. 130].

Korrekte Berechnungen auf Basis der Epipolargeometrie setzen ein genau kalibriertes Kamerasystem voraus. Die Brennweite, der Bildhauptpunkt und die Verzeichnungsparameter sowie die relative Orientierung der Kameras zueinander müssen bekannt sein. Sind diese Parameter mit Unsicherheiten versehen, so schlägt sich das auf die Epipolarlinien nieder. Diese bilden dann einen Korridor im Bild, in dem die bekannten Punkte liegen können.

Dies ist auch der Fall, wenn die Aufnahmen einer Szene von einer einzelnen Kamera betrachtet werden. Die Kamera nimmt zu unterschiedlichen Zeiten aus unterschiedlichen Positionen Bilder einer Szene auf. Ist die Position der Kamera zum Aufnahmezeitpunkt bekannt, so kann die Epipolargeometrie auch in diesem Szenario angewendet werden.

Die Epipolargeometrie vereinfacht sich, wenn durch Vorwissen, wie zum Beispiel durch Nutzung einer Stereokamera, Positionsinformationen von Objektpunkten zur Verfügung stehen. Die möglichen Positionen des Bildpunktes bei bekannter Kameraposition beschränkt sich dann auf einen zusammenhängenden Bereich im Bild, die sogenannte Epipolarregion.

3.5 Verwandte Arbeiten

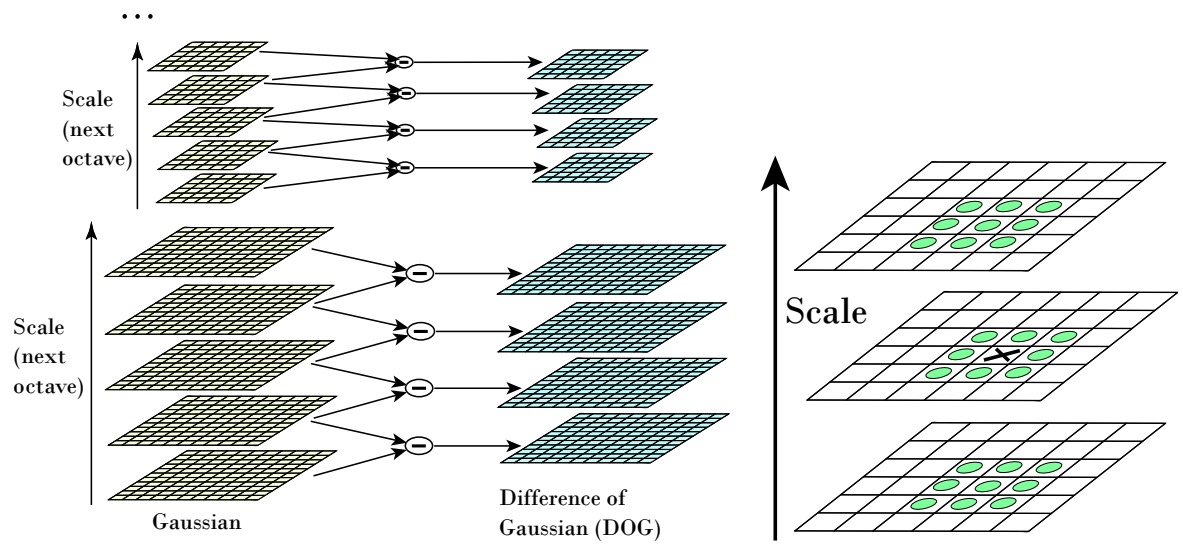
Dieses Kapitel gibt einen Einblick in aktuelle Forschungen im Bereich GPGPU sowie Algorithmen zur Featureerkennung. Neben ORB für Feature Matching existieren zahlreiche

weitere Ansätze Features aus Bildern zu extrahieren. Als Referenz für Feature Detectors und Descriptors gilt Scale Invariant Feature Transform (SIFT) von D. Lowe [32].

Für die meisten Verfahren stehen Implementierungen auf der GPU, CPU sowie für FPGA zur Verfügung. Aus diesem weiten Feld ist der Ansatz zur Beschleunigung von SIFT mithilfe der GPU von Sinha et al. [9]vorgestellt.

3.5.1 SIFT - Scale-Invariant Feature Transform

Im Jahr 2004 veröffentlicht David G. Lowe mit SIFT einen skalierungs- und perspektivinvarianten Feature Detector und Descriptor. SIFT zeichnet sich durch eine hohe Robustheit bezüglich Rotation und Skalierung aus, ist allerdings sehr rechen- und somit auch zeitintensiv. Aufgrund seiner stabilen Features wird das Verfahren oft zum Vergleich herangezogen. ORB wird in der Arbeit von Rublee et al. mit SIFT verglichen und ist zwei Größenordnungen schneller als SIFT und liefert ähnlich stabile Features [26, S. 1]. Im Folgenden ist SIFT in seinen Grundzügen beschrieben. Im Vergleich mit ORB fällt die Komplexität des Verfahrens auf.



(a) Difference of Gaussian [32, S. 6].

(b) Detektion der Features [32, S. 7].

Abbildung 3.13: SIFT Detector unter Nutzung der DoG.

Der SIFT Detector nutzt zur Erkennung von Features die Difference of Gaussian (DoG). Das Ausgangsbild wird dazu mehrfach mit einem Gauß-Filter weichgezeichnet und jede Stufe davon gespeichert. Die Sammlung der weichgezeichneten Stufen des Bildes

heißt Oktave. Die Anzahl an Stufen in einer Oktave ist nicht festgelegt. Die benachbarten Stufen der Oktave werden voneinander subtrahiert und bilden die Difference of Gaussian. Eine schematische Darstellung der DoG ist in Abbildung 3.13a zu sehen. Zur Verbesserung der Skalierungsinvarianz wird die DoG auf eine Bildpyramide des Ausgangsbildes angewendet.

Der Detector sucht in der erstellten Pyramide der DoG nach lokalen Extrema. Jeder Pixel wird mit seinen acht umliegenden Pixeln und den neun Pixeln in der Stufe sowohl darüber und als auch darunter verglichen (siehe Abbildung 3.13b). Ist der aufgenommene Wert größer als die Werte der umliegenden Pixel, so wird der Punkt als Feature erkannt.

Zur Filterung der Features wendet SIFT die HCRM an. Somit werden vorrangig Features genutzt, die an besonders markanten Ecken liegen. Von den gefundenen Features wird die Orientierung mithilfe der Bildgradienten der umliegenden Pixel berechnet. Um die berechnete Orientierung wird das Aufnahmemuster des Descriptors gedreht.

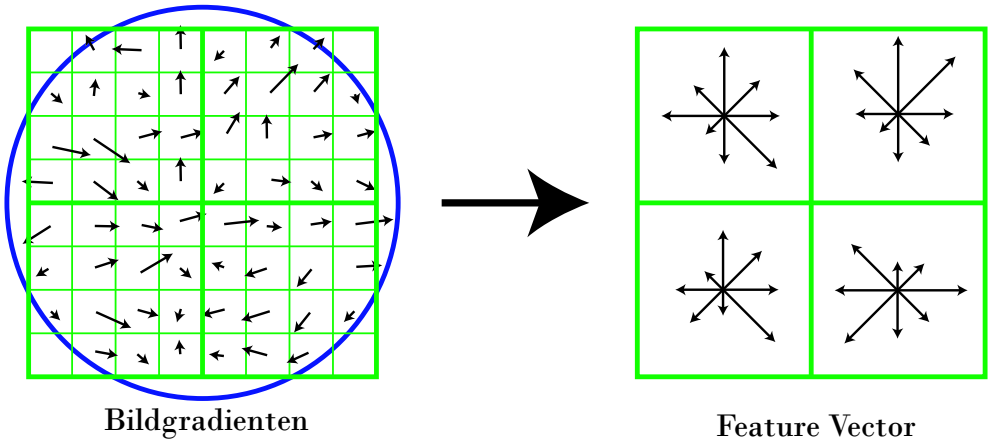


Abbildung 3.14: Descriptor von SIFT [32, S. 15].

Für die Featurebeschreibung nimmt SIFT die Bildgradienten der Pixel um das betrachtete Feature in ein Histogramm auf. Das Histogramm setzt sich aus acht möglichen Richtungen zusammen (siehe Abbildung 3.14).

Das Aufnahmemuster ist ein 16×16 Pixel großes Quadrat um das Feature. Dieses ist in 16 Unterregionen unterteilt. Jede Unterregion besteht somit aus 4×4 Pixeln. Der Bildgradient eines jeden Pixels wird berechnet und für jede Unterregion in einem Histogramm zusammengefasst. Die Histogramme der 16 Regionen bilden den Feature Vector. Durch die acht möglichen Richtungen im Histogramm ergibt sich ein 128-elementiger Vektor. Die Abbildung 3.14 zeigt ein Viertel des tatsächlichen Aufnahmemusters.

3.5.2 GPU-Beschleunigung von SIFT

Die Grafikkarte wird für verschiedene Algorithmen genutzt (siehe Abschnitt 2.1.2). Sinha et al. von der Universität von North Carolina veröffentlichten 2006 eine Arbeit in der die Nutzung der GPU zur Beschleunigung des Feature Detectors und Descriptors von SIFT beschrieben ist [9]. Zu dem Zeitpunkt übliche CPU-Implementierungen benötigen zur Detektion und Beschreibung von 1000 SIFT Features, bei einer Auflösung von ca. 0,3 Megapixeln (MP), 1,1 s (siehe Abbildung 3.15). Mit zunehmender Bildgröße steigt die Berechnungsdauer signifikant - größere Bilder brauchen bedeutend länger. Derartige Berechnungszeiten eignen sich nicht für Echtzeitanwendungen.

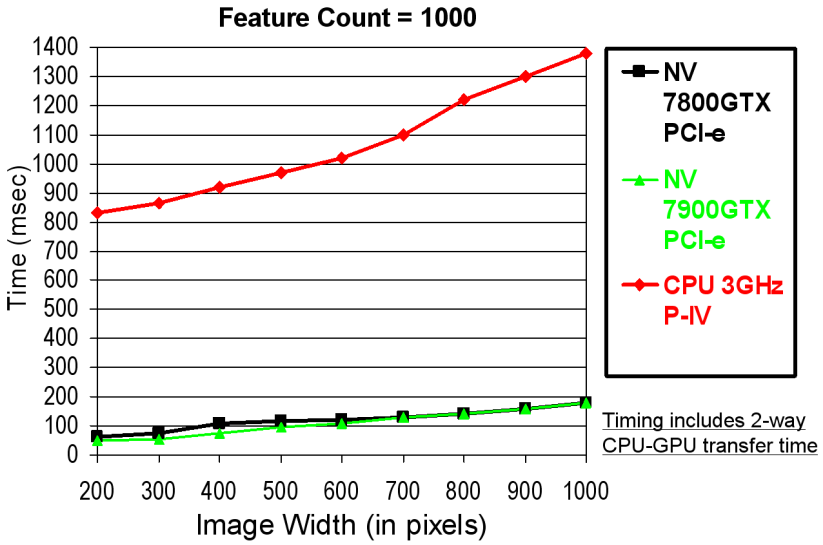


Abbildung 3.15: Ergebnisse der Beschleunigung von SIFT nach Sinha et al. [9, S. 12].

Die Implementierung von Sinha et al. mithilfe von OpenGL erlaubt die Prozessierung eines Bildes mit ähnlicher Auflösung und 1000 SIFT Features bei ca 0,1 s. Damit ist diese Implementierung zehn Mal schneller als die Referenzimplementierung auf der CPU. Bei größeren Bildern vergrößert sich dieser Faktor noch. Bei kleineren Bildern nähern sich die Berechnungszeiten an. Dies ist auf die CPU-GPU-Kommunikation zurückzuführen.

Die Abbildung 3.16 zeigt die Verarbeitungskette der GPU-Implementierung. Die Berechnungen werden auf GPU und CPU verteilt. Parallelisierbare Aufgaben wie der Aufbau der Bildpyramide und des dazugehörigen Skalenraums, die Detektion der Features und die Berechnung der Orientierung geschehen auf der GPU. Die Kommunikation wurde auf ein Minimum reduziert. Insbesondere große Datenmengen, wie die Feature-

Koordinaten auf den verschiedenen Skalenebenen, werden explizit komprimiert.

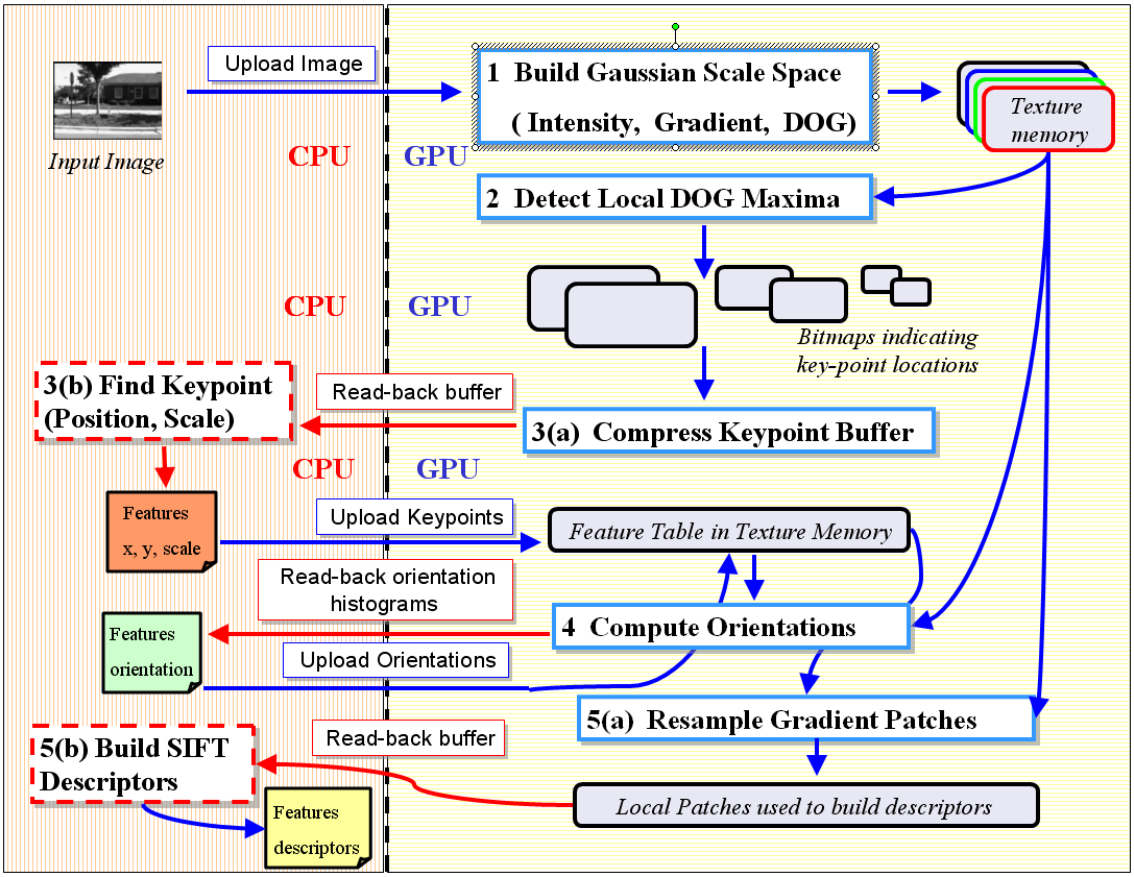


Abbildung 3.16: Konzept zur Beschleunigung mit der GPU nach Sinha et al. [9, S. 10].

Die Vorarbeit von Sinha et al. wurde von Chanchang Wu von der Universität von North Carolina zur Implementierung von SIFTGPU[1] genutzt.

[1] `https://github.com/pitzer/SiftGPU`

4 Konzept

Dieses Kapitel befasst sich mit der konzeptuellen Arbeit zur GPU-basierten Beschleunigung von ORB und dessen Erweiterung zur Verbesserung des Matching-Prozesses. Als Grundlage dienen die Abschnitte 3.3 und 3.4.3, sowie die beschriebenen Ansätze zur GPGPU aus Abschnitt 2.1.2.

Im ersten Teil ist das Konzept zur Implementierung von ORB auf der GPU beschrieben. Dabei ist insbesondere die Verteilung der Rechenlast auf CPU und GPU herausgestellt.

Der zweite Teil konzentriert sich auf die Erarbeitung des Stereo-Matchers und Feature-Trackers. Der Stereo-Matcher nutzt ORB, um in Stereobildpaaren korrespondierende Features zu finden. Die gefundenen Features können aufgrund der Zuordnung im Bildpaar im 3D-Raum lokalisiert werden (Siehe 3.4.2). Diese Positionsinformation ist zusätzlich zum Feature Vector für spätere Anwendungen verfügbar. Der Feature-Tracker nutzt diese Informationen, um in einem Bild, das dieselbe Szene wie das Stereobildpaar darstellt, Features wiederzuerkennen.

Die folgende Abbildung 4.1 stellt den abstrakten Aufbau des Stereo-Matchers und des Feature-Trackers dar. Der im Abschnitt 3.1 vorgestellte Ablauf beim Feature-Matching ist um verschiedene Filterfunktionen basierend auf der bekannten Epipolargeometrie des Stereosystems erweitert. Diese Erweiterung ist in den folgenden Abschnitten erklärt.

4.1 Konzept zur GPU-gestützten Implementierung von ORB

Die Grafikkarte eignet sich, wie im Abschnitt 2.1.2 beschrieben, für die Prozessierung von Bildern, da hier die SIMD-Architektur gut genutzt werden kann. Derselbe Algorithmus wird beispielsweise bei einem 1280×720 Pixel großen Bild, wenn auf das gesamte Bild angewendet, fast eine Million Mal ausgeführt. Es sollten nur wenige und gleichzeitig kleine Datenmengen zwischen CPU und GPU ausgetauscht werden. Auf diese Besonderheiten ist das Konzept zur GPU-gestützten Implementierung von ORB ausgelegt. Die Abbildung 4.2 dient im Folgenden der Veranschaulichung. Nach der Benennung in 2.1.2

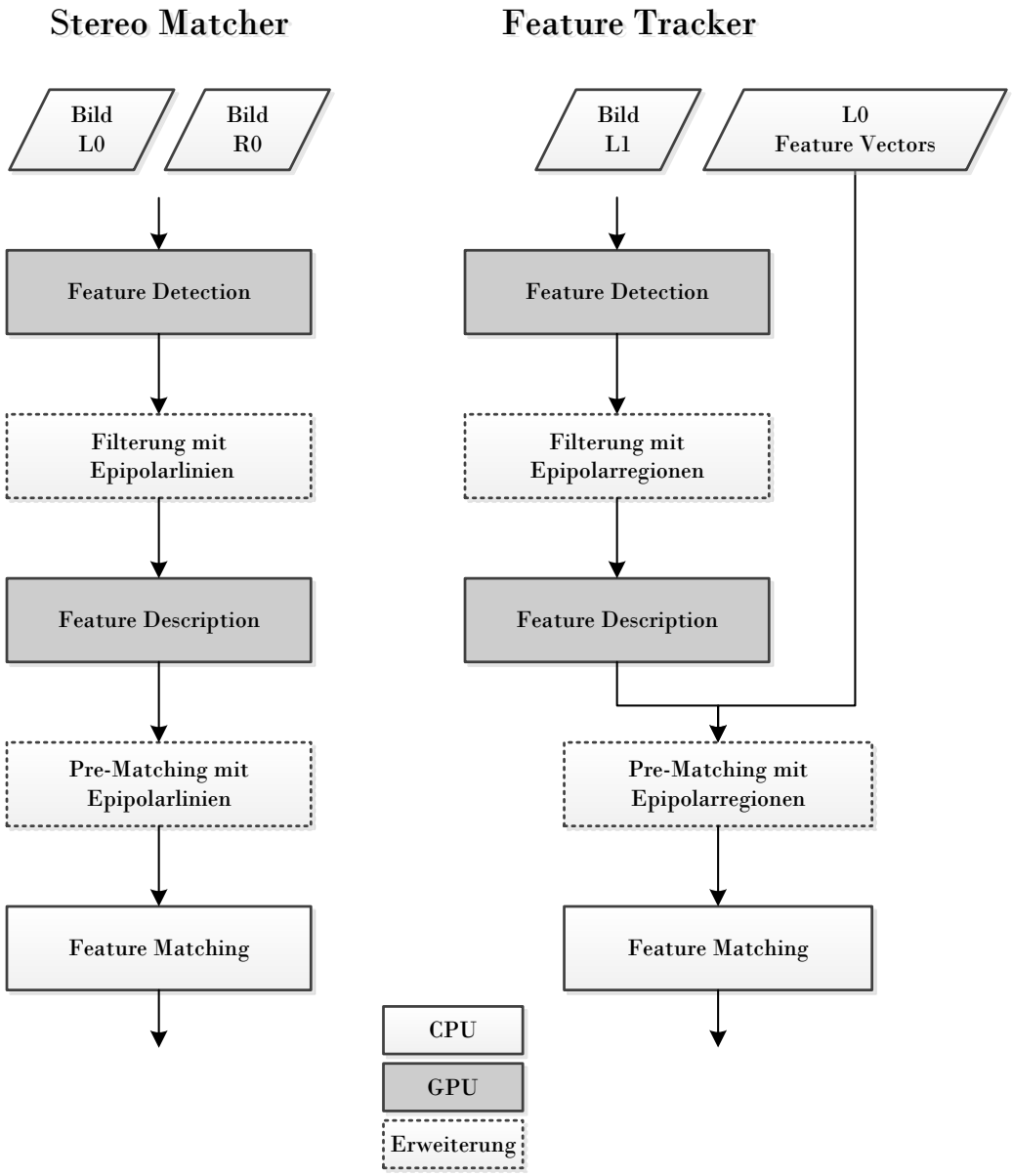


Abbildung 4.1: Übersicht über Stereo-Matcher und Feature-Tracker.

dient die CPU als Host und die GPU als Device.

Als Ausgangspunkt dient ein Bild, das von der Festplatte oder auch direkt von der Kamera kommt und das im Arbeitsspeicher liegt. Als Erstes wird das zu verarbeitende Bild in den Speicher der GPU geladen. Der Kommunikationsaufwand beschränkt sich damit auf die einfache Bildgröße.

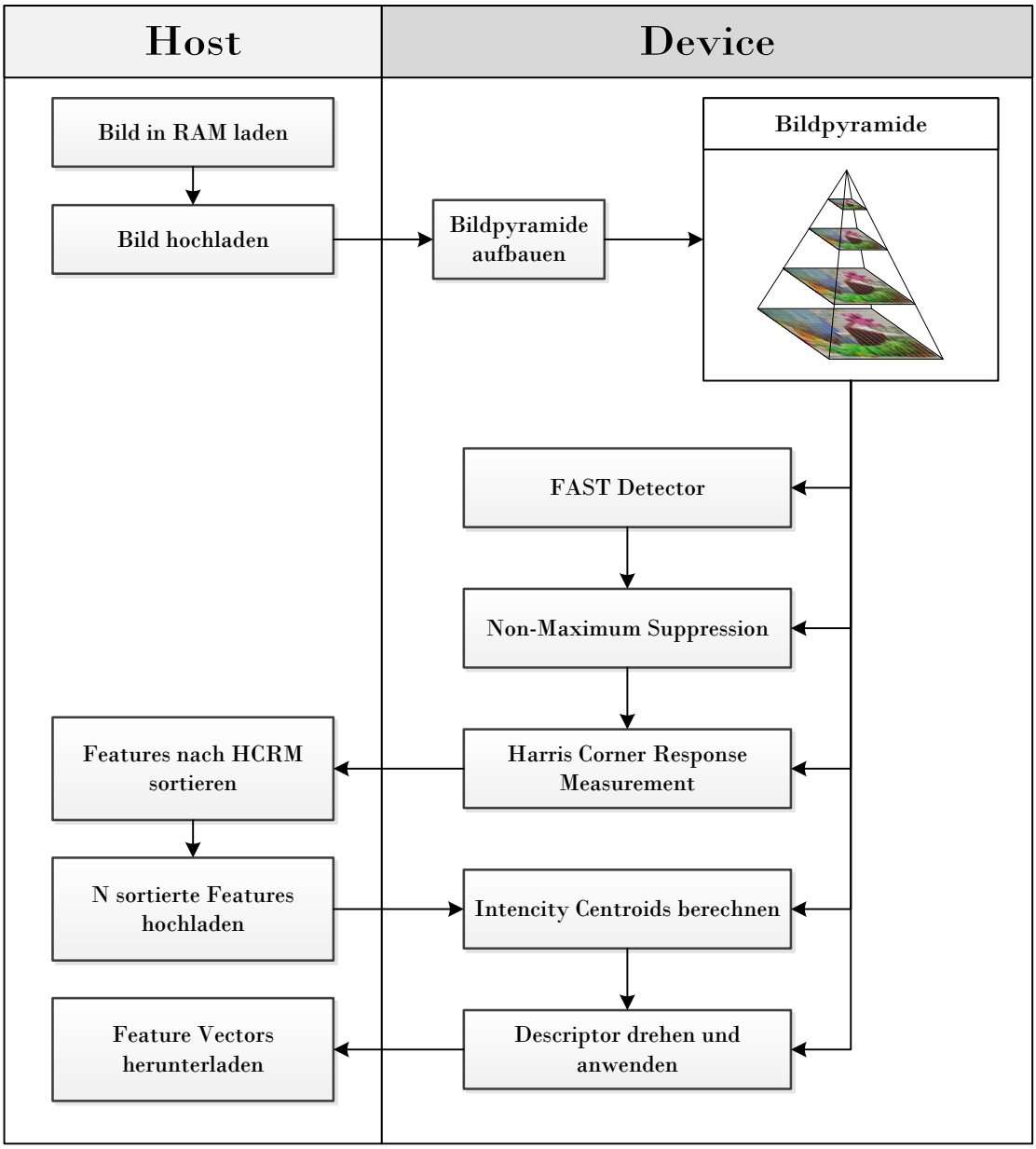


Abbildung 4.2: Verteilung der Prozessierungsschritte von ORB auf Host und Device.

Die Erstellung der Bildpyramide setzt sich aus Kopier- und Interpolationsfunktionen zusammen, was durch den schnellen Speicher der Grafikkarte gut umgesetzt werden kann. Alle weiteren Verarbeitungsschritte werden auf jedem Level der Bildpyramide ausgeführt. Von hier an sind ausschließlich lesende Zugriffe auf den Speicherbereich der Pyramide notwendig.

FAST betrachtet alle Pixel jeder Stufe der Bildpyramide und detektiert somit auf verschiedenen Skalierungsstufen Features. Der Anzahl der Ausführungen von FAST richtet sich nach der Anzahl an Pixeln in der Bildpyramide. Diese setzt sich zusammen aus der Bildbreite und Bildhöhe sowie der Anzahl der Pyramidenlevel N und der Skalierung s.

$$\text{Anzahl Iterationen/Pixel} = \sum_{i=0}^{N-1} \frac{\text{Bildbreite} \times \text{Bildhöhe}}{s^i} \tag{4.1}$$

Als Ergebnis liegen nun die Bildkoordinaten der Pixel vor, die FAST als Features betrachtet (Siehe 3.3.1). Non-Maximum Suppression filtert diese Features. Die Anzahl der Ausführungen entspricht der Anzahl an gefundenen FAST-Features. Die gefilterten Features werden weiterverwendet, die nicht benötigten Features werden verworfen.

Die HCRM liefert zu jedem Feature den Grad der Unterscheidbarkeit der Ecke. Die Sortierung der Features anhand dieser Bewertung erfolgt auf der CPU. Aufgrund des sequentiellen Zugriffs auf die Zielliste können Sortierfunktionen die parallele Prozessierung der GPU nur schlecht nutzen. Die mögliche Verwendung eines parallelen Sortierverfahrens mithilfe von OpenCL ist zu erarbeiten.

Die berechneten Bewertungen sowie die dazugehörigen Koordinaten der Features werden in den Arbeitsspeicher geladen. Anhand dieser Kennzahlen sortiert die CPU die Features und behält die besten - alle anderen werden verworfen. Die Anzahl an gesuchten Features dient hier als Parameter.

Die sortierte Liste wird wieder in den Speicher der GPU geladen. Die Grafikkarte berechnet zu jedem Feature den Intensity Centroid und extrahiert den Winkel. Um diesen Winkel gedreht beschreibt der Descriptor die Umgebung des Features. Das Descriptor Muster kann schon bei der Initialisierung im GPU-Speicher abgelegt werden.

Wie in Abbildung 4.2 zu sehen sind bei bei der Ausführung vier Datentransfers zwischen Host und Device nötig. Diese umfassen einen Bildtransfer sowie zwei Datentransfers von Koodinaten und einen Transfer der Feature Vectors.

4.2 Erweiterung von ORB um Epipolargeometrie

ORB zeichnet sich durch seine Geschwindigkeit und seine große Anzahl stabiler Features aus [26]. Sobald allerdings größere Veränderungen im Bild auftreten liefert der Matching-Prozess zahlreiche Fehlmatches [33]. Diese sind im weiteren Verlauf nicht nutzbar und

sollten aus diesem Grund vermieden werden.

Die Erweiterung von ORB um die Nutzung der Epipolargeometrie stellt einen Ansatz vor, um die Zahl der Fehlmatches zu verringern. Durch den festen Kameraaufbau und Vorwissen aus der Trajektorie stehen Informationen über die Aufnahmegeometrie zur Verfügung, die die Berechnung von Epipolarlinien ermöglichen. Die Nutzung der Epipolargeometrie ist in diesem Abschnitt beschrieben.

4.2.1 Epipolarlinien zur Filterung der zu beschreibenden Features

Ein erster Ansatz zur Nutzung der Epipolargeometrie besteht in der Filterung der zu beschreibenden Features. Der Ablauf ist in der folgenden Abbildung 4.3 dargestellt.

Der ORB Detector findet im linken wie im rechten Kamerabild Features. Zu jedem Feature im linken Kamerabild wird die dazugehörige Epipolarlinie im rechten Bild berechnet. Features im rechten Bild, die auf einer dieser Linien liegen werden vom Descriptor beschrieben. Alle anderen Features werden verworfen.

Durch diese Filterung stehen im anschließenden Matching nur Feature Vectors zur Auswahl die auf Epipolarlinien liegen. Zusätzlich verringert sich die Zahl der zu beschreibenden Features wodurch die Laufzeit reduziert wird. Die berechneten Linien können im Stereo-Matcher erneut verwendet werden.

4.2.2 Stereo-Matching mit ORB

Mithilfe des Stereo-Matchers werden Features aus einem Stereobildpaar verglichen und einander zugeordnet. Die gefundenen Paare ermöglichen die Rekonstruktion der Tiefeninformation (Siehe Abschnitt 3.4.2).

Die von ORB detektierten und beschriebenen Features können mithilfe der Epipolargeometrie für das Matching gefiltert werden. Es wird angenommen, dass die beschriebene Filterung aus Abschnitt 4.2.1 angewendet wurde. In Abbildung 4.4 ist das Konzept bildlich dargestellt. $L0$ entspricht dem linken Stereokamerabild, $R0$ dem rechten Bild.

Anhand der Features im linken Bild spannt die Erweiterung ein Epipolarbüschel im rechten Bild auf. Jedes Feature im linken Bild besitzt nun eine korrespondierende Epipolarlinie im rechten Bild, diese sind farblich hervorgehoben. Die Farbe des Features im linken Bild entspricht der Farbe der dazugehörige Epipolarlinie im rechten Bild. Alle

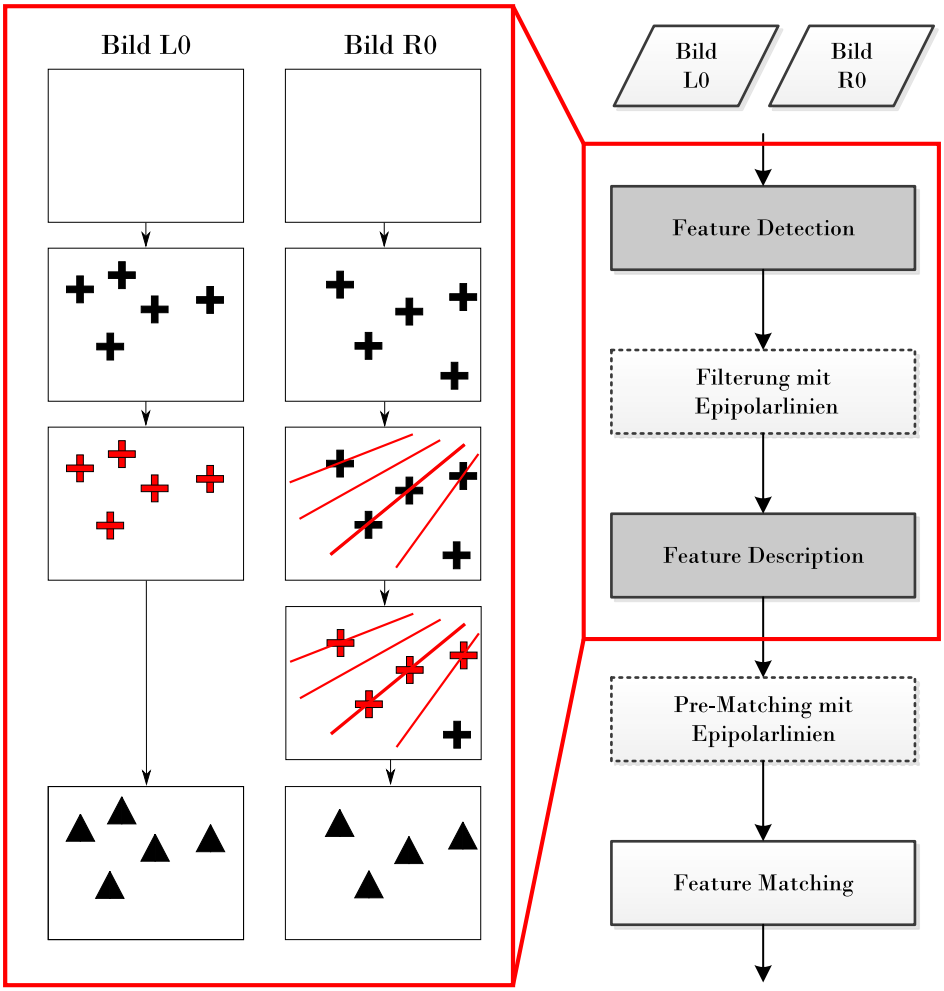


Abbildung 4.3: Filterung der zu beschreibenden Features.

beschriebenen Features im rechten Bild liegen auf mindestens einer Epipolarlinie. Jede dieser Linien stellt eine Liste von Features dar, die als mögliche Matches für das korrespondierende Feature im linken Bild in Frage kommen. Diese Liste stellt nur noch einen Bruchteil der Features dar. Features können in verschiedenen Listen vorkommen. Die Zahl möglicher Fehlmatches wird durch die eingeschränkte Auswahl für das Matching reduziert.

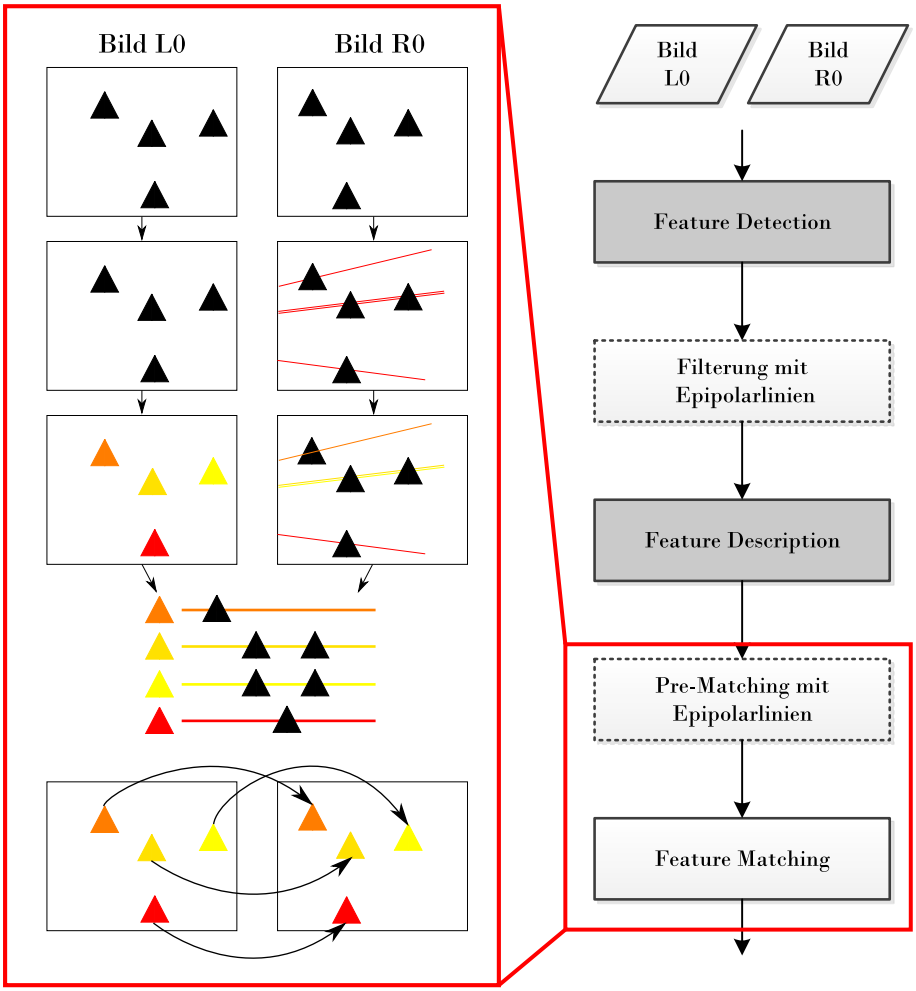


Abbildung 4.4: Konzept für den Stereo-Matcher.

4.2.3 Feature-Tracking mit ORB

Der Feature-Tracker ermöglicht die Wiedererkennung von Features in einem Bildpaar, dessen Bilder zu unterschiedlichen Zeitpunkten aufgenommen wurden. Diese sind in der Abbildung 4.5 als L0 und L1 gekennzeichnet. Der Stereo-Matcher extrahiert die Features im Bild L0 und dem dazugehörigen Bild R0 zu einem früheren Zeitpunkt und speichert die dazugehörigen Feature Vectors. Durch das Stereo-Matching stehen zusätzlich zum Feature Vector Positionsinformationen zu den einzelnen Features zur Verfügung.

Ebenso ist die Position der Kamera zu dem Zeitpunkt bekannt, an dem L1 aufgenommen wird. Aus den Positionen der Features von L0 erlaubt die Epipolargeometrie eine Schätzung, wo diese Features in L1 liegen werden. Diese werden Epipolarregionen

genannt. Die mögliche Position ist in der Abbildung farblich markiert.

Die Features innerhalb der Epipolarregionen werden zu Listen zusammengefasst. Jedes Feature aus L0, das theoretisch in L1 sichtbar ist, hat eine Liste von möglichen korrespondierenden Features. Der Matcher vergleicht jetzt das jeweilige Feature mit der dazugehörigen Liste aus L1 und sucht Zuordnungen.

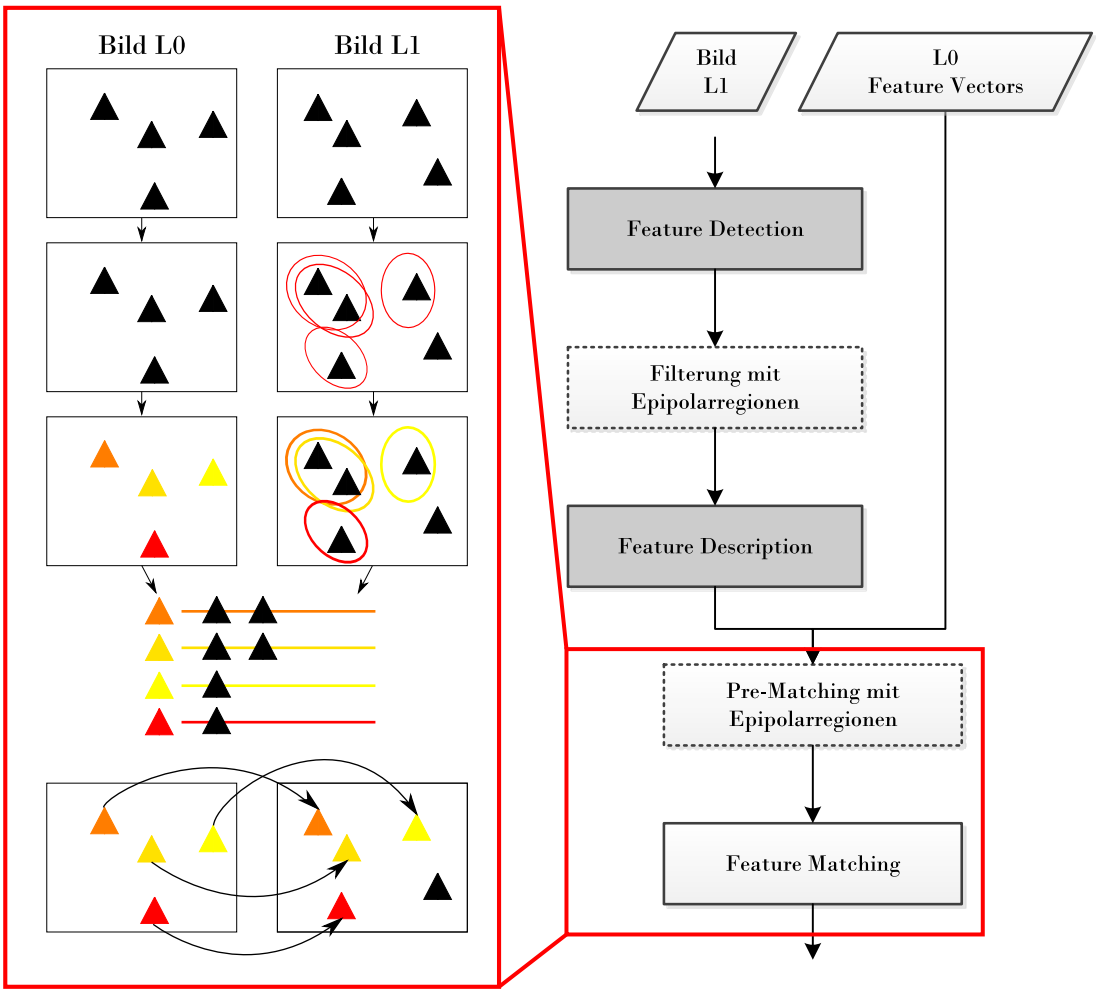


Abbildung 4.5: Konzept für den Feature-Tracker.

5 Implementierung und Validierung von ORB auf der GPU

Dieses Kapitel stellt die Implementierung von ORB auf der GPU vor. Dabei stehen die Besonderheiten bei der Nutzung der Grafikkarte durch OpenCL im Vordergrund.

Im Datenfluss der Prozessierungskette sind die einzelnen Verarbeitungsschritte in der Implementierung beschrieben und im darauf folgenden Abschnitt auf ihre Ausführungszeit untersucht. Die detaillierte Betrachtung unterstützt die Identifikation von Engpässen und zeigt Möglichkeiten zur Optimierung auf. Der dargestellte Datenfluss setzt das in Kapitel 4 vorgestellte Konzept um.

5.1 Speichermanagement auf CPU und GPU

Der Host und das Device besitzen eigene Speicher mit jeweils getrenntem Speichermanagement. Die Kommunikation zwischen beiden Speicherbereichen wird durch die Implementierung umgesetzt. Es ist dabei nicht möglich Klassen oder deren Instanzen zwischen Host und Device auszutauschen. Es können ausschließlich auf Byteebene definierte Datenblöcke genutzt werden. An diese Anforderungen ist die Kommunikation der mithilfe der GPU beschleunigten Version von ORB angepasst.

Um Daten zwischen beiden Seiten zu transferieren, müssen Speicherbereiche derselben Größe auf beiden Seiten definiert werden. Diese Speicher sind auf Seite des Device nicht veränderlich und müssen dementsprechend passend dimensioniert sein. Zu kleine Speicherbereiche führen bei einem Speicherüberlauf zum Abbruch des ausgeführten Kernels, zu große Speicherbereiche verbrauchen den knapp bemessenen Speicher von integrierten Grafikkarten, wie sie auf dem Zielsystem vorhanden sind. Es existiert jeweils ein Speicherblock für die einzelnen Buffer des Devices auf Seiten des Hosts. Dieser ist durch einen Vektor der C++-Standardbibliothek umgesetzt und garantiert einen kontinuierlichen Speicherbereich im Arbeitsspeicher.

Das folgende Klassendiagramm 5.1 zeigt eine vereinfachte Darstellung der aktuell implementierten Klasse `ORB`. Die beschriebenen Buffer und das dazugehörige Pendant auf der Hostseite sind die zentralen Elemente über die die Speicherzugriffe ablaufen. Der

Zugriff auf die Funktionalität der Grafikkarte geschieht über einen OpenCL-Context und die dazugehörige Schnittstelle zur Verarbeitung von Befehlen, der `Queue`. Diese sind an ein `Device` auf der entsprechenden `Platform` gebunden. Zur Kapselung des Zugriffs sind die benannten, zusammengehörigen Objekte in einem zentralen Objekt zusammengefasst.

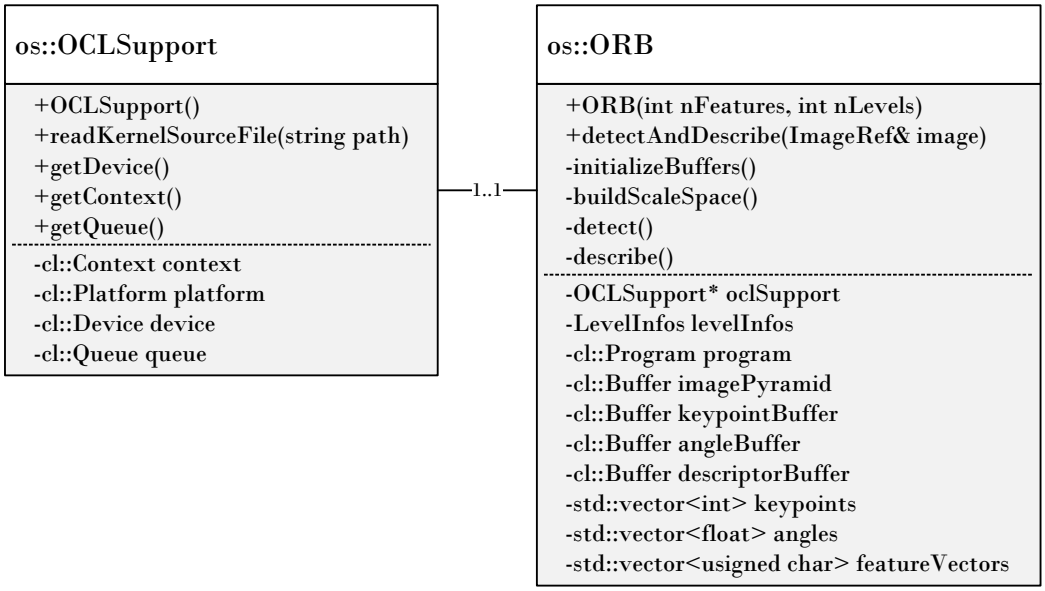


Abbildung 5.1: Abstrahierte ORB-Klasse mit OpenCL-Unterstützung.

Zu jedem Speicherblock auf Seite des Hosts gehört ein Speicherblock auf dem Device. Dieser ist dem Host über eine Referenz in einem OpenCL-Buffer bekannt. Das Bufferobjekt dient der Kommunikation mit dem Devicespeicher. Die Lese- und Schreiboperationen werden über dieses Objekt koordiniert und durch den Treiber ausgeführt.

Für den Nutzer ist eine möglichst gute Handhabbarkeit der Feature relevant. Wissen über den internen Speicheraufbau der Datenblöcke kann nicht vorausgesetzt werden. Aus diesem Grund werden die Speicherbereiche in einzelne Feature Objekte kopiert und diese als Liste nach der Prozessierung zurückgegeben. Dazu werden der Vector der Koordinaten und der Beschreibung aufgelöst und die Daten in zusätzliche Klassen, die Features und Listen von Features repräsentieren ausgelagert.

5.2 Datenfluss der Prozessierungskette

Die aktuelle Implementierung setzt das in der folgenden Abbildung 5.2 dargestellte Aktivitätsdiagramm um. Für eine echtzeitfähige Implementierung ist es nötig, gesondert auf die Wiederverwendung von bereits reservierten Speicherbereichen zu achten. Damit können sowohl Speicherlecks verhindert, als auch die Geschwindigkeit erhöht werden. Der dargestellte Datenfluss beginnt mit dem Aufruf der Funktion `detectAndDescribe` und einem übergebenen Bild.

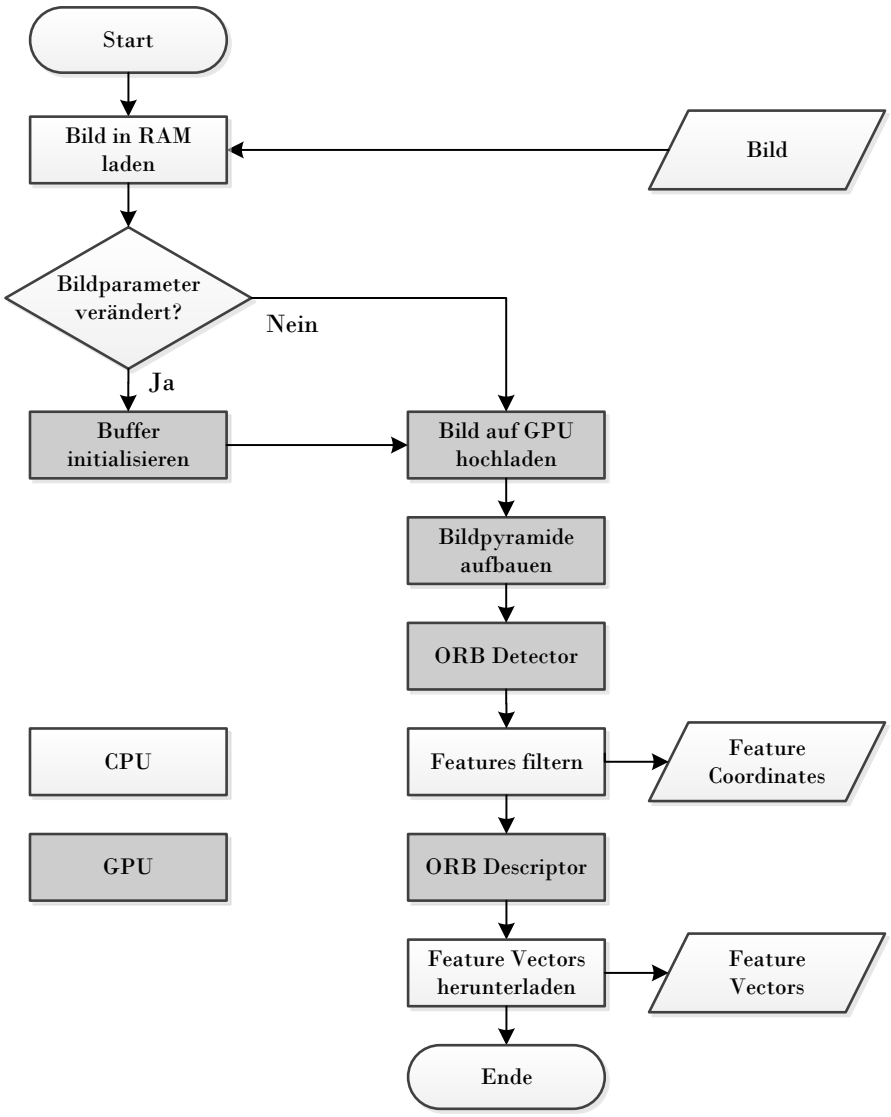


Abbildung 5.2: Datenfluss des ORB Detector und Descriptor.

Das Bild liegt im Arbeitsspeicher und seine Eigenschaften werden mit den Eigenschaften des vorhergehenden Bildes verglichen. Handelt es sich um die erste Verwendung der Funktion oder hat das Bild eine andere Größe als das vorhergehende, so werden alle Buffer auf der Grafikkarte initialisiert. Bereits existierende Buffer werden gelöscht und neu allokiert.

Bei der Initialisierung reserviert der Treiber der Grafikkarte Speicher für die Bildpyramide und deren Eigenschaften sowie für die Features und deren Orientierung. Der Speicherbereich für die Features umfasst initial 200.000 Features, um genügend Kapazität auch für hochauflösende Bilder bereitzustellen. Ein einzelnes Feature setzt sich aus zwei 32 Bit großen Integerwerten zusammen. Der allokierte Speicher beträgt damit 1,6 MB.

Nach der Initialisierung lädt der Host das Bild in den Speicher des Devices und startet den Aufbau der Bildpyramide. Das Device generiert auf Basis des Bildes die Bildpyramide. Diese bleibt unverändert und steht in weiteren Schritten zur Verfügung.

Der ORB Detector findet mithilfe von FAST Features auf der Bildpyramide und berechnet zu jedem Feature den Harris Corner Score. Die Liste von Koordinaten wird zusammen mit den Scores in den Speicher des Hosts geladen. Dieser sortiert beide Listen nach den größten Scores und behält die besten Features. Alle anderen Features werden verworfen. Die Übrigen werden sortiert wieder auf den Device-Speicher geladen.

Die verbleibenden Features dienen nun als Grundlage für die Berechnung der Intensity Centroids, die zu jedem Feature die Orientierung repräsentieren. Anhand dieser Orientierung wird das Descriptor-Muster gedreht und angewendet. Im Speicher des Devices liegen nun die Feature Vectors hintereinander. Dieser Speicherblock wird in den Speicher des Hosts übertragen. Der Host extrahiert die einzelnen Vectors und bildet aus den Koordinaten und den Feature Vectors die nutzbaren Features.

5.3 Laufzeitanalyse der Implementierung

Dieser Abschnitt diskutiert die Verarbeitungszeiten der einzelnen Glieder in der Prozessierungskette der aktuellen Implementierung. Dadurch können vorhandene Engpässe und mögliche Optimierungen aufgedeckt werden. Die folgende Tabelle 5.1 fasst die Ergebnisse aus der Laufzeitanalyse zusammen. Die Analyse nutzt ein Bild mit einer Auflösung von 0,35 MP. Das System umfasst die in Abschnitt 1.3 beschriebene Hardware.

Typ	Prozessierungsschritt	Erläuterung	Zeit
Transfer	Bildtransfer Host-Device	0,35 MB	0,4 ms
GPU	Bildpyramide	4 Layer	3,1 ms
GPU	FAST Detector	4 Layer (911.360 Pixel)	12,3 ms
GPU	Non-Maximum Suppression	13.000 Features	4,6 ms
GPU	HCRM	13.000 Features	2,8 ms
Transfer	Feature-Transfer Device-Host	52 KB + 13 KB	1,6 ms
CPU	Features sortieren und filtern	13.000 Features	0,1 ms
Transfer	Feature-Transfer Host-Device	4 KB	0,2 ms
GPU	Intensity Centroids	1.000 Features	1,4 ms
GPU	Descriptor anwenden	1.000 Features	2,1 ms
Transfer	Feature Vectors	32 KB	0,2 ms
CPU	Feature Vectors extrahieren	1.000 Features	0,5 ms
	Gesamt-Prozessierungszeit		29,3 ms

Tabelle 5.1: Zeit für einzelne Prozessierungsschritte.

Die vier Datentransfers zwischen Host und Device brauchen in dem beschriebenen Szenario zusammen 2 ms. Diese Zeit fällt bei einer reinen CPU-Implementierung nicht an. Auf die Summe der Zeiten von 29,3 ms bezogen umfasst die Zeit zum Datentransfer 7 % der Gesamtausführungszeit.

Die längste Berechnungsdauer hat der FAST Detector in Kombination mit Non-Maximum Suppression. Mit insgesamt 16,9 ms brauchen beide Schritte über 50 % der Gesamtlaufzeit. Hier besteht Optimierungsbedarf. Eine genauere Analyse des Detectors ist dafür notwendig.

Alle anderen Prozessschritte liegen in einem vertretbaren Zeitfenster. Trotz des hohen Zeitverbrauchs im Detector zeigt die Laufzeitanalyse, dass mithilfe der GPU ORB, wie es von den Autoren des Verfahrens Rublee et al. veröffentlicht wurde, beschleunigt werden kann. Die angestrebte Prozessierungszeit von unter 100 ms (siehe 1.2) ist mit 30 ms um das dreifache unterschritten.

Die folgende Abbildung 5.3 stellt die Entwicklung der Ausführungszeit in Abhängigkeit zur Bildgröße dar. ORB nutzt als feste Konfiguration vier Ebenen der Bildpyramide mit

einem Skalierungsfaktor von 2 und detektiert und beschreibt 1.000 Features. Als kleinste Bildgröße dient ein Bild mit den Maßen 680×512 Pixel und damit einer Auflösung von 0,35 MP. Die Messwerte sind über 1.000 Messungen gemittelt und repräsentieren die durchschnittliche Laufzeit für ein Bild.

Die Analyse zeigt, dass die Laufzeit bei Bildgrößen im einstelligen Megapixelbereich linear wächst und bei größeren Datenmengen exponentiell ansteigt. Dies lässt sich mit der Architektur von Grafikkarten erklären (siehe 2.1.1). Durch die begrenzte Prozessorzahl können nur begrenzt viele Prozesse gleichzeitig ausgeführt werden. Bei großen Datenmengen ist eine parallele Verarbeitung nicht mehr möglich und wird sequenziell durchgeführt.

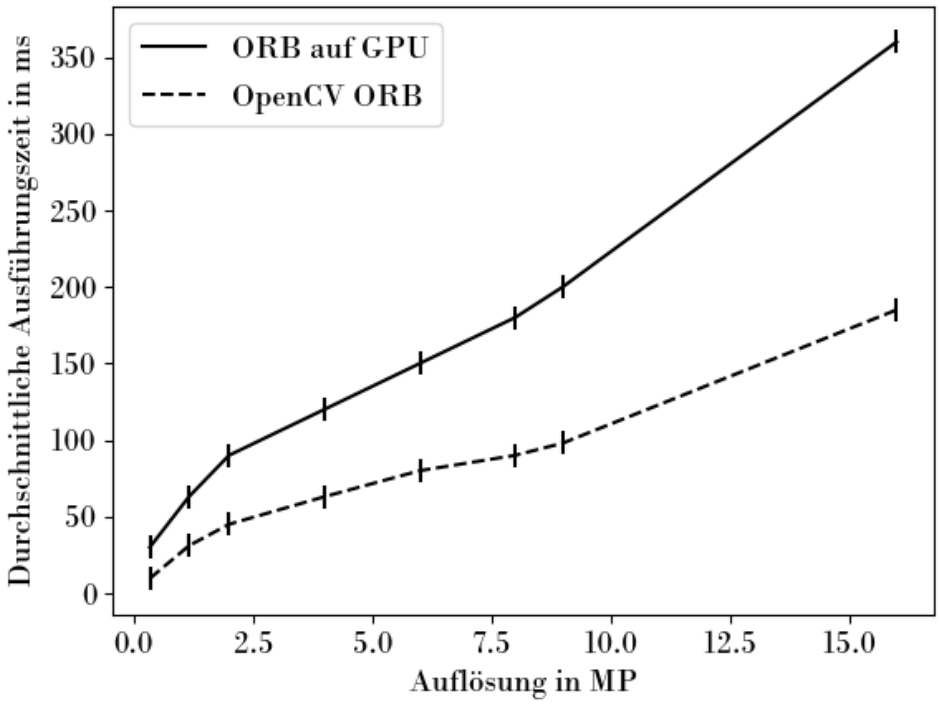


Abbildung 5.3: Laufzeit bei verschiedenen Bildgrößen.

Die Anzahl zu detektierender Features beeinflusst die Laufzeit, wie Abbildung 5.4 zeigt. Das untersuchte Bild hat eine Größe von 1260×924 Pixeln. Die verwendete Konfiguration entspricht der Konfiguration aus der Analyse der Laufzeit bei verschiedenen Bildgrößen.

Die Laufzeit des Detectors ist unabhängig von der Anzahl detektierter Features, da im

ersten Schritt alle FAST Features im Bild betrachtet und auf diese die Non-Maximum Suppression und HCRM anwendet werden (siehe 4.1). Erst die Filterfunktion und der Descriptor beeinflussen die Laufzeit bei unterschiedlicher Anzahl an Features. Der Einfluss wächst dabei linear mit der Zahl der beschriebenen Features.

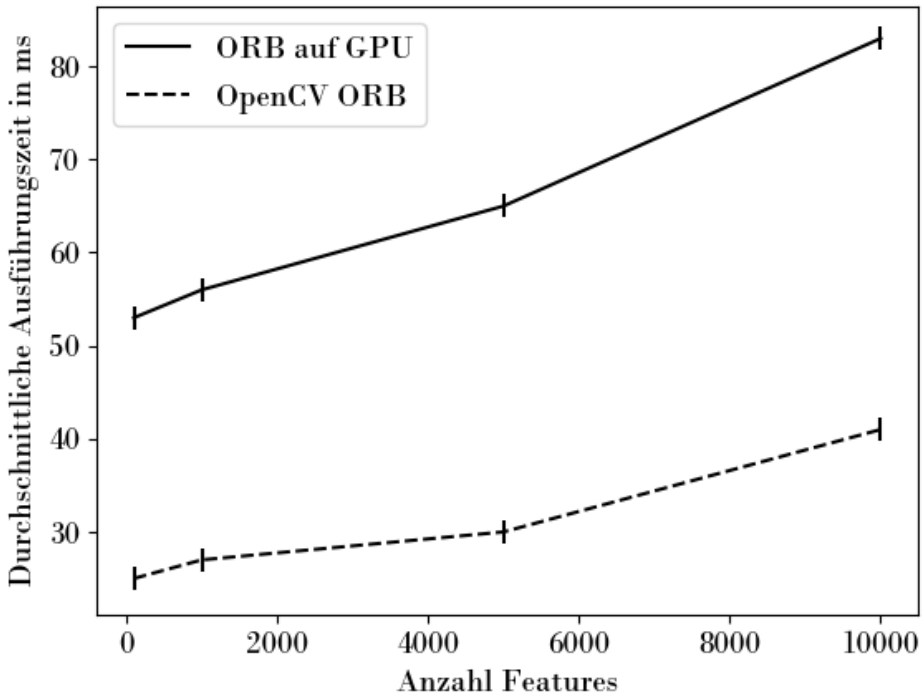


Abbildung 5.4: Laufzeit bei verschiedener Anzahl Features.

5.4 Vergleich der GPU-Implementierung mit der CPU-Implementierung von OpenCV

Zur Validierung des implementierten Verfahrens werden die Laufzeiten von ORB auf der GPU und ORB auf der CPU von OpenCV verglichen. Beide Implementierungen stellen eine zentrale Funktion zur Erkennung und Beschreibung von Merkmalen in einem eingegebenen Bild zur Verfügung. Die gleichen Prüfverfahren wie für die Laufzeitanalyse der GPU-Implementierung wurden auf die Implementierung von OpenCV angewendet. Die Ergebnisse sind den Abbildungen 5.3 und 5.4 zu entnehmen.

Die Implementierung von OpenCV ist allen Testfällen um den Faktor drei schneller als die GPU-Implementierung. Die Laufzeitanalyse hat gezeigt, dass der größte Anteil dem FAST Feature Detector zuzuschreiben ist. Die Rechenzeit des Detectors allein ist höher als die gesamte Laufzeit der OpenCV-Implementierung. Hier besteht Optimierungsbedarf.

Die Kommunikation zwischen Host und Device stellt in dem vorgestellten Prozess keinen Engpass in der Verarbeitung dar. Damit sollte die Optimierung vorrangig in der Prozessierung ansetzen. OpenCV zeigt hier, dass Laufzeiten von 10 ms pro Bild erreicht werden können.

Während der Laufzeitanalyse wurde zusätzlich die Auslastung der CPU und GPU betrachtet. Während der Prozessierung belegt die OpenCV-Version einen Kern des Prozessors komplett. 25 % der CPU sind ausgelastet. Die GPU-Version belegt den Prozessor mit 5 % und einzelnen Spitzen von 10 %. Diese sind auf die Sortierung mit der CPU zurückzuführen. Die GPU-Nutzung liegt während der Prozessierung bei 50 %. Um die Kommunikation weiter zu verringern und ausschließlich die GPU für die Prozessierung von Detector und Descriptor zu nutzen, sollte eine Implementierung einer Sortierfunktion nach dem HCRM auf der GPU in Erwägung gezogen werden.

6 Fazit

Im Rahmen dieser Arbeit wurde eine funktionsfähige Version von ORB unter Nutzung von OpenCL implementiert. Die entwickelte Lösung nutzt die GPU, um rechenintensive Schritte parallel zu verarbeiten. Die CPU wird dabei zu 5 % ausgelastet.

Zusätzlich stellt diese Arbeit verschiedene Konzepte vor, um die Qualität beim Matching von Features basierend auf Vorwissen aus der Aufnahmegeometrie zu erhöhen. ORB wird hier für die Zuordnung von Features sowohl in Stereobildpaaren, als auch in Bildpaaren, die aus verschiedenen Positionen aufgenommen wurden, verwendet.

Dieses Kapitel fasst die Ergebnisse dieser Arbeit zusammen und beschreibt mögliche zukünftige Entwicklungen im Ausblick.

6.1 Zusammenfassung

Die Implementierung ermöglicht die Nutzung des ORB Detectors und Descriptors für die Verarbeitungskette des IPS. ORB erlaubt ein Matching von Features aus Bildern unterschiedlicher Perspektive, wodurch nun ein Tracking von Features in Bildern, die nicht direkt hintereinander aufgenommen sind, möglich ist. Die Implementierung nutzt OpenCL zur Beschleunigung des Verfahrens durch die GPU.

Die Vorarbeit hat gezeigt, dass OpenCL sich für diese Implementierung eignet. Die Auslegung der Sprache auf GPGPU ist hier von Vorteil. Auch OpenGL und OpenVX sind für die Anwendung im IPS in Betracht gezogen worden.

Die initiale Implementierung von ORB unter Nutzung von OpenCL konnte trotz der Komplexität des Feature Detectors und Descriptors abgeschlossen werden. Das ursprüngliche Ziel einer Laufzeit von unter 100 ms für die Erkennung und Beschreibung der Features in einem Bildpaar ist erreicht. Im Falle des gegebenen Testsystems nutzt die Implementierung die integrierte Grafikkarte Intel HD 4000 und erlaubt eine Prozessierung von 30 Bildern pro Sekunde. Dabei wird die CPU zu 5 % genutzt. Die Grafikkarte übernimmt den Großteil der Berechnungen und ist zu 50 % ausgelastet.

Für die Prozessierung existiert eine Verarbeitungskette, die auf Basis des eingegebenen Bildes Speicherbereiche auf Host- und Device reserviert und diese zur effizienten Nutzung wiederverwendet. Die Verarbeitung findet in einer zentralen ORB-Klasse statt.

Die OpenCL-Schnittstelle ist in eine eigene Klasse ausgelagert und kann für zukünftige Entwicklungen, die OpenCL nutzen, im Institut verwendet werden.

Der Vergleich der Laufzeit der OpenCV-Implementierung von ORB hat Optimierungsbedarf aufgezeigt. Eine Prozessierung von 100 Bildern in der Sekunde ist erreichbar. Dazu sind vorrangig Optimierungen des Detectors notwendig. Die aktuelle Implementierung des FAST Detectors benötigt länger als die gesamte Prozessierung von OpenCV auf der CPU. Die beschriebenen Verbesserungen werden im Anschluss an die Arbeit durchgeführt.

Zusätzlich zur Implementierung von ORB sind drei Konzepte zur Verbesserung von ORB erarbeitet worden. Das IPS verfügt durch die fest verbaute Stereokamera und die bekannte Position im Raum über Vorwissen aus der Aufnahmegeometrie. Diese Informationen werden genutzt, um verschiedene Filterfunktionen zu ermöglichen, die das Matching der von ORB gefundenen Features verbessern.

Die erarbeiteten Konzepte verringern die Anzahl an zu beschreibenden Features indem Features, die nicht in beiden Bildern sichtbar sind entfernt werden. Auf Basis der Epipolargeometrie, die direkt aus der Aufnahmegeometrie berechnet werden kann, können Epipolarlinien und -regionen in den betrachteten Bildern aufgespannt werden. Jedem Feature im ersten betrachteten Bild wird ein Bereich im zweiten Bild zugeordnet, in dem das korrespondierende Feature liegt. Durch diese Filterung wird die Anzahl möglicher Matches und damit die Wahrscheinlichkeit eines Fehlmatches reduziert. Andererseits erhöht die Einschränkung des Suchraums für mögliche Matches den Rechenaufwand in der Verarbeitung.

Neben der Implementierung von ORB und der Erarbeitung der Konzepte zur Nutzung der Epipolargeometrie enthält die Arbeit wichtige Erkenntnisse für die Entwicklung von GPU-beschleunigten Algorithmen. Die grundlegende Einführung in die GPGPU hat gezeigt, dass bei der Entwicklung besonders auf die Datenhaltung und Verteilung der Berechnungen auf Host und Device geachtet werden sollte. Hier ist zusätzlicher Aufwand im Design einzuplanen, um spätere Probleme in der Entwicklung zu vermeiden. In diesem Zusammenhang ist OpenCL erfolgreich in institutsinterne Bibliotheken integriert worden, sodass zukünftig Entwickler in der Lage sind ohne zusätzlichen Aufwand OpenCL direkt in der Bibliothek zu nutzen.

6.2 Ausblick

Die Implementierung von ORB erlaubt die Nutzung des Verfahrens für verschiedene Anwendungsgebiete wie Stereo-Matching und Loop-Closing. Sie bildet die Grundlage für zukünftige Entwicklungen des IPS im Bereich der Verarbeitung von Bildmerkmalen von Bildern verschiedener Perspektive in Echtzeit. Die aktuelle Implementierung ermöglicht durch die Nutzung von OpenCL die Verteilung der Gesamtlast auf dem System. Der Vergleich mit der Implementierung von OpenCV zeigt allerdings, dass die implementierte Version verbessert werden sollte. Insbesondere der FAST Detector bedarf einer grundlegenden Überarbeitung. Die Optimierung steht im Zentrum der folgenden Entwicklung. In diesem Zusammenhang sind zusätzliche Verbesserungen in der Prozessierung anzugehen.

Die Sortierung der Features kann durch einen parallelen Sortieralgorithmus wie den Bitonischen Sortieralgorithmus [34] auf der GPU durchgeführt werden. Dadurch reduziert sich die gesamte Kommunikation zwischen Host und Device auf zwei Datentransfers. Das Ausgangsbild wird auf das Device und die Features sowie die dazugehörigen Feature Vectors auf den Host übertragen. Der implementierte Sortieralgorithmus kann auch unabhängig von der Nutzung in ORB verwendet werden.

Durch die Integration von OpenCL können weitere Algorithmen auf der Grafikkarte umgesetzt werden. Erste Ansätze zur Parallelisierung der Größenskalierung und der Entzerrung von Bildern mithilfe von OpenGL haben das Potenzial dieser Technologie aufgezeigt. Die Implementierung eines Frameworks, dass mit OpenCL umgesetzte und GPU-beschleunigte Algorithmen zusammenfasst kann die aktuelle Verarbeitungskette des IPS verbessern.

Die Optimierung umfasst zusätzlich eine erneute Betrachtung der Architektur, die eine modulare Zusammensetzung von verschiedenen Detectors und Descriptors ermöglichen soll. Auf diese Weise können neue und bessere Verfahren effektiv integriert werden.

An die Optimierung der Implementierung schließt sich die Umsetzung der vorgestellten Konzepte an. Diese sollen gut in die modulare Architektur integrierbar sein, da die Konzepte auch auf andere Verfahren im Umgang mit Features anwendbar sind. Beim Stereo-Matcher und Feature-Tracker steht die schnelle Generierung der Listen mit möglichen Zuordnungen im Vordergrund. Hier ist zu prüfen ob eine GPU-beschleunigte Implementierung anzustreben ist.

Literatur- und Quellenverzeichnis

[1] M. G. Wing, A. Eklund und L. D. Kellogg, "Consumer-grade global positioning system (gps) accuracy and reliability", *Journal of Forestry*, Bd. 103, Nr. 4, S. 169–173, 2005, ISSN: 0022-1201. Adresse: http://www.ingentaconnect.com/search/download?pub=infobike://saf/jof/2005/00000103/00000004/art00004%7B%5C&%7Dmimetype=application/pdf%7B%5C&%7DexitTargetId=1326123442764.

[2] D. Grießbach, "Stereo-vision-aided inertial navigation", Diss., Freie Universität Berlin, 2015. Adresse: http://www.diss.fu-berlin.de/diss/receive/FUDISS_thesis_000000099531.

[3] H. Zhang, J. Wohlfeil und D. Grießbach, "Extension and evaluation of the agast feature detector", *ISPRS*, Bd. III, Nr. 4, S. 133–137, 2016. DOI: 10.5194/isprsannals-III-4-133-2016.

[4] D. Blythe, "Rise of the graphics processor", *Proceedings of the IEEE*, Bd. 96, Nr. 5, S. 761–778, 2008, ISSN: 00189219. DOI: 10.1109/JPROC.2008.917718.

[5] J. Nickolls und W. J. Dally, "The gpu computing era", *IEEE Micro*, Bd. 30, Nr. 2, S. 56–69, 2010, ISSN: 02721732. DOI: 10.1109/MM.2010.41.

[6] J. Fung, "Computer vision on the gpu", *GPU Gems*, Bd. 2, S. 649–666, 2005.

[7] J. Fung und S. Mann, "Computer vision signal processing on graphics processing units", *Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04). IEEE International Conference on*, Bd. 5, V–93–6 vol.5, 2004. DOI: 10.1109/icassp.2004.1327055.

[8] R. Kalarot und J. Morris, "Comparison of fpga and gpu implementations of real-time stereo vision", in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops, CVPRW 2010*, 2010, S. 9–15, ISBN: 9781424470297. DOI: 10.1109/CVPRW.2010.5543743.

[9] S. Sinha, J. Frahm, M. Pollefeys und Y. Genc, "Gpu-based video feature tracking and matching", *EDGE, Workshop on Edge ...*, Bd. 012, Nr. May, S. 1–15, 2006, ISSN: 0932-8092, 1432-1769. DOI: 10.1.1.107.3260. Adresse: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.107.3260%7B%5C&%7Drep=rep1%

7B%5C&%7Dtype=pdf%7B%5C$%7D%7B%7D5C%7B%5C$%7Dnhttp://www.cs.unc.edu/techreports/06-012.pdf.

[10] *Nvidia tesla v100*, Accessed: 2017-07-12. Adresse: https://www.nvidia.com/en-us/data-center/tesla-v100/.

[11] *Geforce gtx 1080 founders edition: Premium construction and advanced features*, Accessed: 2017-07-12. Adresse: https://www.geforce.com/whats-new/articles/geforce-gtx-1080-founders-edition.

[12] Intel, *The compute architecture of intel ® processor graphics gen9*, 2015.

[13] Y. Fujii, T. Azumi, N. Nishio, S. Kato und M. Edahiro, "Data transfer matters for gpu computing", in *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, 2013, S. 275–282, ISBN: 9781479920815. DOI: 10.1109/ICPADS.2013.47.

[14] Khronos Group, "The opengl graphics system: a specification", Techn. Ber., 2017, S. 805. Adresse: https://www.khronos.org/registry/OpenGL/specs/gl/glspec45.core.pdf.

[15] B. Gaster, L. Howes, D. Kaeli, P. Mistry und D. Schaa, *Heterogeneous Computing with OpenCL*. 2013, ISBN: 9780124058941. DOI: 10.1016/C2012-0-03322-4. arXiv: arXiv:1011.1669v3.

[16] Khronos Group, "The opencl specification", Techn. Ber., 2017, S. 230. Adresse: https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.pdf.

[17] ——, "The openvx™ specification", Techn. Ber., 2017. Adresse: https://www.khronos.org/registry/OpenVX/specs/1.2/OpenVX_Specification_1_2.pdf.

[18] G. Bradski und A. Kaehler, *Learning OpenCV*, 1. Aufl. O'Reilly, 2008, S. 571, ISBN: 978-0-596-51613-0. Adresse: http://www-cs.ccny.cuny.edu/~wolberg/capstone/opencv/LearningOpenCV.pdf.

[19] M. Hassaballah, A. A. Abdelmgeid und H. A. Alshazly, *Image Feature Detectors and Descriptors*. 2016. DOI: 10.1007/978-3-319-28854-3_2.

[20] T. Tuytelaars und K. Mikolajczyk, "Local invariant feature detectors: a survey", *Computer Graphics and Vision*, Bd. 3, Nr. 3, S. 177–280, 2008, ISSN: 1572-2740. DOI: 10.1561/0600000017.

[21] A. Neubeck und L. Van Gool, "Efficient non-maximum suppression", in *Proceedings - International Conference on Pattern Recognition*, Bd. 3, 2006, S. 850–855, ISBN: 0769525210. DOI: 10.1109/ICPR.2006.479.

[22] *Détection des contours d'une image*, Accessed: 2017-08-02. Adresse: http://mp.cpgedupuydelome.fr/document.php?doc=D%C3%A9tection%20de%20contours.txt.

[23] C. Harris und M. Stephens, "A combined edge and corner detector", *Proc 4th Alvey Vision Conference*, 1988.

[24] B. Jähne, "Digitale bildverarbeitung", *Igarss 2014*, Nr. 1, S. 1–5, 2014, ISSN: 0717-6163. DOI: 10.1007/s13398-014-0173-7.2. arXiv: arXiv:1011.1669v3.

[25] P. L. Rosin, "Measuring corner properties", *Computer Vision and Image Understanding*, Bd. 73, Nr. 2, S. 291–307, 1999, ISSN: 10773142. DOI: 10.1006/cviu.1998.0719. Adresse: http://linkinghub.elsevier.com/retrieve/pii/S1077314298907196.

[26] E. Rublee, V. Rabaud, K. Konolige und G. Bradski, "Orb: an efficient alternative to sift or surf", in *Proceedings of the IEEE International Conference on Computer Vision*, 2011, S. 2564–2571, ISBN: 9781457711015. DOI: 10.1109/ICCV.2011.6126544.

[27] E. Rosten, R. Porter und T. Drummond, "Faster and better: a machine learning approach to corner detection", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Bd. 32, Nr. 1, S. 105–119, 2010, ISSN: 01628828. DOI: 10.1109/TPAMI.2008.275. arXiv: 0810.2434v1.

[28] E. Rosten, *Fast corner detection*, Accessed: 2017-08-02. Adresse: https://www.edwardrosten.com/work/fast.html.

[29] M. Calonder, V. Lepetit, C. Strecha und P. Fua, "Brief: binary robust independent elementary features", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Bd. 6314 LNCS, 2010, S. 778–792, ISBN: 364215560X. DOI: 10.1007/978-3-642-15561-1_56.

[30] O. Schreer, "Stereoanalyse und bildsynthese", *Transformation*, Bd. 57, Nr. 50, S. 1–74, 2005, ISSN: 1422-6405. DOI: 10.1159/000142566. Adresse: http://link.springer.com/content/pdf/10.1007/3-540-27473-1.pdf.

[31] R. Fritzsche, *Entwicklung und implementierung eines algorithmus für die vermessung von 3d-objekten mit einem multikamerasystem*, 2015.

[32] D. G. Lowe, "Distinctive image features from scale-invariant keypoints", *International Journal of Computer Vision*, Bd. 60, Nr. 2, S. 91–110, 2004, ISSN: 09205691. DOI: 10.1023/B:VISI.0000029664.99615.94. arXiv: 0112017 [cs].

[33] Z. Peng, *Efficient matching of robust features for embedded slam*, 2012. Adresse: http://elib.uni-stuttgart.de/opus/volltexte/2012/7617/.

[34] B. Jan, M. Bartolomeo, C. Ragusa, F. G. Khan und O. Khan, "Fast parallel sorting algorithms on gpus", *International Journal of Distributed and Parallel Systems IJDPS*, Bd. 3, Nr. 6, S. 107–118, 2012. DOI: 10.5121/ijdps.2012.3609.