HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Fakultät
Ingenieurwissenschaften und Informatik

Bachelorarbeit

Entwurf und High-Level-Synthese einer FPGA-basierten Hardwarebeschleunigung des AprilTag-Algorithmus auf einem Zynq-SoC

Erstprüfer (Themensteller):	Prof. Winfried Gehrke
Zweitprüfer:	Simon Vellas
Bearbeiter:	Marcel Flottmann
Matrikelnummer:	763435
Ausgabedatum:	20.05.2019
Abgabedatum:	12.08.2019

I Kurzfassung

Moderne Raumfahrtsysteme müssen immer aufwendigere Manöver autonom im Weltraum durchführen. Dazu sind Informationen über die Umgebung erforderlich. Mithilfe von AprilTags können beliebige Objekte geortet werden, um deren Positionsdaten für diese Zwecke zu erfassen. Dies sind künstliche optische Bezugspunkte, die vom Aussehen her an QR-Codes erinnern. Mit einer Kamera kann die Position und Drehung der AprilTags bestimmt werden. Diese sollen zunächst als Machbarkeitsnachweis im Institut für Raumfahrtsysteme des Deutschen Zentrums für Luft- und Raumfahrt e.V. (DLR) eingesetzt werden. Der dazugehörige Bildverarbeitungs-Algorithmus, der die Tags in einem Bild findet, identifiziert und ortet, soll auf einem Xilinx Zynq-SoC implementiert werden, da dieser in zukünftigen Raumfahrzeugen des DLR eingesetzt wird. Die offizielle Version des Algorithmus ist auf dem Zynq-SoC als reine Software zu langsam und nicht für eine Regelung geeignet. Daher wird in dieser Arbeit der Algorithmus teilweise in den integrierten FPGA des Zynq-SoC verlegt. Dadurch sollen die Berechnungen beschleunigt werden, um so eine höhere Bildrate, die für den Einsatz in einer Regelung geeignet ist, zu erreichen. In dieser Arbeit wird gezeigt, dass durch eine geschickte Aufteilung die doppelte Bildrate erreicht werden kann.

Abstract

Modern aerospace systems have to execute more and more advanced autonomous maneuvres in outer space. In order to do this, they need information about their environment. AprilTags can be used to locate arbitrary objects and gather their position data to perform those tasks. They are optical fiducial markers that look similar to QR codes. The position and rotation of AprilTags can be determined through a camera. They are intended to be used as a proof of concept by the Institute of Space Systems of the German Aerospace Center (DLR). The corresponding image processing algorithm is supposed to be implemented on a Xilinx Zynq-SoC, wich will be used for future space systems of the DLR. The official version of the algorithm is too slow as a pure software approach on the Zynq-SoC for control tasks. Therefore, the algorithm shall be partly moved to the integrated FPGA on the Zynq-SoC in this research. As a result, the calculations are supposed to be accelerated and thereby increase the frame rate for a possible use in control tasks. This research shows that a thoughtful splitting can lead to a doubled frame rate.

II Inhaltsverzeichnis

III Abbildungsverzeichnis

IV Tabellenverzeichnis

V Abkürzungsverzeichnis

ACP	Accelerator-Coherency-Port
API	Application Programming Interface
CCL	Connected-Components-Labeling
COTS	Components-off-the-shelf
CPU	Central Processing Unit
DLR	Deutsches Zentrum für Luft- und Raumfahrt e.V.
DSP	Digital Signal Processor
EOL	End of Line
FPGA	Field Programmable Gate Array
HLS	High-Level-Synthesis
HPN	High Performance Node
II	Initiation Inerval
IP-Block	Intellectual Property Block
OBC	On-Board Computer
RCN	Reliable Computing Node
ScOSA	Scalable On-Board Computing for Space Avionics
SoC	System-on-Chip
SOF	Start of Frame
STL	Standard Template Library
TPSS	Two-Pass-Single-Storage

VI Glossar

ACP (Accelerator-Coherency-Port) Eine C++ Bibliothek aus Klassen und Funktionen mit Templates, die viele der häufig genutzten Datentypen und Algorithmen implementieren.

API (Application Programming Interface) Eine Schnittstelle einer Softwarebibliothek, um diese in einer Anwendung einzubinden.

CCL (Connected-Components-Labeling) Zusammenhängende Regionen in einem Bild bekommen ein einzigartiges Label zugeordnet.

COTS (Components-off-the-shelf) Seriengefertigte Bauteile, die durch die hohe Stückzahl günstiger als Spezialbauteile sind und meistens für den Consumerbereich hergestellt werden.

DSP (Digital Signal Processor) Dient der kontinuierlichen Verarbeitung von Signalen. In Bezug zu Xilinx FPGAs sind feste Einheiten aus Multiplizierer, Akkumulator, Logik, Multiplexer und Register gemeint.

EOL (End of Line) Kennzeichnet das Ende einer Bildzeile bei der Videoübertragung.

FPGA (Field Programmable Gate Array) Integrierter Schaltkreis, dessen Logik programmierbar ist.

HLS (High-Level-Synthesis) Transformation von C++-Code in Hardwarebeschreibungssprachen wie VHDL und Verilog.

HPN (High Performance Node) ScoSA Knoten mit hoher Rechenleistung.

II (Initiation Inerval) Anzahl der Takte bis eine Funktion oder Schleife wieder von vorne starten kann.

IP-Block (Intellectual Property Block) Wiederverwendbare Funktionsblöcke in der FPGA Entwicklung und Mikroelektronik.

OBC (On-Board Computer) Computer, die in (Raum-)Fahrzeugen genutzt werden.

RCN (Reliable Computing Node) ScoSA Knoten mit hoher Zuverlässigkeit.

ScOSA (Scalable On-Board Computing for Space Avionics) Verteiltes Computer System für Raumfahrzeuge, das vom Institut für Raumfahrtsysteme des DLR entwickelt wird.

SoC (System-on-Chip) Integration mehrerer Teilsysteme wie CPU und Peripherie auf einem Chip.

SOF (Start of Frame) Beginn eines neuen Bilds bei der Videoübertragung.

STL (Standard Template Library) Eine C++ Bibliothek aus Klassen und Funktionen mit Templates, die viele der häufig genutzten Datentypen und Algorithmen implementieren.

TPSS (Two-Pass-Single-Storage) Ein CCL-Verfahren, dass Konflikte während des ersten Durchlaufs behebt und nur einen Zwischenspeicher benötigt.

Verilog s. VHDL

VHDL Hardwarebeschreibungssprache mit der sich die z.B. die Logik eines FPGAs beschreiben lässt.

1 Einleitung

1.1 Vorstellung des Unternehmens

Das Deutsche Zentrum für Luft- und Raumfahrt e.V. (DLR) forscht in den Bereichen Luftfahrt, Raumfahrt, Energie, Verkehr, Sicherheit und Digitalisierung und ist im Auftrag der Bundesregierung für das deutsche Raumfahrtmanagement zuständig. An zwanzig Standorten sind vierzig Institute und Einrichtungen in ganz Deutschland verteilt. Der Hauptsitz ist in Köln. Im Jahr 2017 standen dem DLR 1001 Mio. Euro zur Verfügung. Davon waren 507 Mio. Euro durch Bund und Länder finanziert, der Rest wurde durch Drittmittel eingeworben. Insgesamt sind 8127 Mitarbeiter angestellt. Das Institut für Raumfahrtsysteme in Bremen hat die Aufgabe, zukünftige Raumfahrzeuge und Raumfahrtmissionen zu entwerfen, zu analysieren und zu bewerten [3].

Neue Raumfahrtsysteme benötigen immer mehr Rechenleistung. Deswegen wird im Institut für Raumfahrtsysteme der neue On-Board Computer (OBC) ScOSA (Scalable On-Board Computing for Space Avionics) entwickelt. Dies ist ein verteiltes Rechennetz, das aus zuverlässigen aber langsamen Knoten (Reliable Computing Nodes (RCNs)) und schnellen Knoten (High Performance Nodes (HPNs)) besteht [10]. Die HPN nutzen Components-off-the-shelf (COTS) als Grundbaustein. Es werden die Zynq-SoCs von Xilinx verbaut. Diese sind zwar nicht vor Strahlung geschützt, haben aber mehr Rechenleistung. Daher sind sie für nicht kritische rechenintensive Aufgaben geeignet. Ein Aufgabenbereich ist die Bildverarbeitung, die auch in dieser Arbeit behandelt wird. Der sogenannte AprilTag-Algorithmus soll auf einem solchen Zynq-SoC als Demonstration implementiert werden. Mit diesem können die Position und die Drehung von AprilTags, die so ähnlich aussehen wie QR-Codes, aus einem Bild extrahiert werden.

1.2 Ziele und Aufgabenstellung

Da der Algorithmus für Regelungsaufgaben als reine Software auf dem Zynq-SoC nicht schnell genug ist, soll in dieser Arbeit eine beschleunigte Version des AprilTag-Algorithmus zu entwickeln werden, die den integrierten FPGA auf einem Zynq-SoC nutzt. Die Implementierung soll mithilfe des High-Level-Synthesis (HLS)-Tool von Xilinx umgesetzt werden. Softwareseitig ist eine Bibliothek zu erstellen, die die Komplexität der Ansteuerung verbirgt.

Die Beschleunigung soll durch Aufteilung des Algorithmus auf FPGA und CPU erreicht werden. In einem Vorprojekt [4] wurden mögliche Schritte des Algorithmus identifiziert, die auf den FPGA portiert werden könnten. Die Beschreibung der IP-Blöcke soll in C++ erfolgen, die vom HLS-Tool in eine Hardwarebeschreibungssprache konvertiert wird.

Abschließend sollen die offizielle Version und die hier vorgestellte miteinander verglichen werden. Trotz der Beschleunigung des Verfahrens, soll die Genauigkeit, mit der die Position und die Drehung der AprilTags erfasst werden, nicht geringer ausfallen.

Das Ziel der Arbeit ist es, herauszufinden, wie weit der AprilTag-Algorithmus auf einem Zynq-SoC beschleunigt werden kann, ohne dass die Genauigkeit abnimmt.

1.3 Ausgangssituation

In einem Vorprojekt [4] wurde der AprilTag-Algorithmus hinsichtlich der Beschleunigung durch einen Field Programmable Gate Array (FPGA) untersucht. Es wurden Teilschritte identifiziert, die sich aufgrund ihrer Struktur auf einem FPGA implementieren lassen. Dabei entstand eine Systemarchitektur, die in drei Subsystemen aufgeteilt ist. Davon bilden die Kameraansteuerung und die Videoausgabe zusammen ein wiederverwendbares Grundsystem. Dieses basiert auf dem Beispielprojekt von Digilent, das für das Zybo Z7-20 Board und der Pcam 5C Kamera bereitgestellt wird. Ergänzt wurde es um einen HLS Intellectual Property Block (IP-Block), der die Verzeichnung der Linse korrigiert. Das dritte Subsystem ist der AprilTag-Algorithmus, der in einzelne Schritte aufgeteilt wurde. Die ersten fünf Schritte sollen auf dem FPGA untergebracht werden und die übrigen werden weiterhin als Software ausgeführt.

1.4 Aufbau der Arbeit

In dieser Arbeit soll gezeigt werden, wie der AprilTag-Algorithmus durch den FPGA beschleunigt werden kann. Dazu wird in den Grundlagen zunächst geklärt, wie der Algorithmus strukturiert ist und welche Entwicklungstools eingesetzt werden. Insbesondere erfolgt eine ausführliche Beschreibung der hier genutzten High-Level-Synthesis.

Anschließend werden die Anforderungen an das zu erstellende System vorgestellt. Neben den funktionalen und nicht-funktionalen Anforderungen wird auf die Systemarchitektur eingegangen, die festlegt, welche Schritte des Algorithmus auf dem FPGA implementiert werden sollen. Dabei wird auch erläutert, warum diese ausgewählt wurden.

In Kapitel 4 wird beschrieben, wie die Schritte des Algorithmus im FPGA umgesetzt sind und welche Maßnahmen zur Optimierung der Bildrate durchgeführt worden sind. Am Ende des Kapitels folgen die Ergebnisse der Integration der IP-Blöcke und der Synthese des FPGA-Designs.

Das nachfolgende Kapitel widmet sich dem Softwareteil. Darin wird der Entwurf und die Implementierung der Ansteuerungsbibliothek vorgestellt.

In der Auswertung wird zunächst definiert, wie die Performance des Systems zu

bewerten ist und wie der Performancetest durchgeführt wird, dessen Ergebnisse schließlich analysiert werden.

Abschließend werden das Gesamtergebnis der Arbeit zusammengefasst und ein kurzer Ausblick gegeben.

2 Grundlagen

2.1 AprilTag

AprilTags sind künstliche optische Bezugspunkte, die so ähnlich aussehen wie QR-Codes. Durch das AprilTag-Erkennungsverfahren kann ihre Position und Rotation im Raum relativ zur Kamera bestimmt werden. In den Tags ist eine ID kodiert, um diese zu unterscheiden. Zusätzlich ist die Kodierung so gewählt, dass daraus hervorgeht, welche Kante des Tags „oben“ ist. Dadurch lässt sich die Rotation eindeutig bestimmen.

Das Verfahren für die Erkennung gibt es bereits in der dritten Version [7], die auch als Grundlage für diese Arbeit genommen wird. Diese basiert auf die vorherige Version [12] und schlägt einige Optimierungen vor. Außerdem wird die Möglichkeit von flexiblen Layouts vorgestellt, auf die aber in dieser Arbeit nicht eingegangen wird. Für die erste Version des Algorithmus wurde bereits an einer FPGA-Implementierung geforscht [13], die aber durch die neueren Versionen nicht für diese Arbeit verwendbar ist. Die offizielle Version des Verfahrens ist in C geschrieben und auf GitHub [1] veröffentlicht. Die Erkennung kann in neun Schritte unterteilt werden:

Schritt 1 (Decimate): Das Bild wird um einen beliebigen, zur Laufzeit festlegbaren Faktor N verkleinert. Dabei wird nur jede N-te Zeile und Spalte in ein neues Bild kopiert, das in den folgenden Schritten weiterverarbeitet wird. Das Original wird in den Schritten Refinement und Decode wieder benötigt.

Schritt 2 (BlurSharpen): In diesem Schritt kann das verkleinerte Bild entweder mit einem Gauß-Filter weichgezeichnet oder geschärft werden. Durch einen vor dem Start festlegbaren Parameter ist die Stärke einstellbar. Das Vorzeichen bestimmt, ob weichgezeichnet oder geschärft wird.

Schritt 3 (Threshold): Es erfolgt eine Segmentierung in helle und dunkle Bereiche, sowie Bereiche mit wenig Kontrast. Dazu wird ein lokales Schwellwertverfahren genutzt. Zunächst wird das Bild in vier mal vier Pixel große Kacheln aufgeteilt, deren Minima und Maxima jeweils ein neues Bild ergibt. Das Minima-Bild wird erodiert und das Maxima-Bild dilatiert. Der Schwellwert berechnet sich dann aus dem Mittelwert zwischen Minimum und Maximum. Falls die Differenz zu klein ist, ist der Kontrast nicht ausreichend.

Schritt 4 (Connected Components Labeling): Zusammenhängende Segmente werden zu Komponenten zusammengefasst und bekommen ein eindeutiges Label zugeordnet. In der offiziellen Version wird dazu eine UnionFind-Datenstruktur benutzt.

Schirtt 5: (Gradient Cluster): In diesem Schritt werden alle Subpixel erfasst, die sich auf der Grenze zwischen einer hellen und einer dunklen Komponente befinden (Kantenpixel). Dabei wird die Achter-Nachbarschaft verwendet. Für jede Komponentenkombination wird eine eigene Liste mit den Subpixel geführt, die die Position und die Kantenrichtung speichert. Es wird eine Hash-Tabelle genutzt, um die Daten der passenden Liste zuzuordnen.

Schritt 6 (Quad): Zunächst wird der Mittelpunkt eines Kantenzugs mithilfe einer Bounding Box, die alle Pixel der Liste umspannt, ermittelt. Dann werden die Pixel nach dem Winkel um den Mittelpunkt sortiert und mehrfache Einträge für eine Position entfernt. Anschließend sucht der Algorithmus nach den Eckpunkten des Vierecks. Dazu wird eine Gerade an ein Fenster, das aus aufeinanderfolgenden Kantenpixeln besteht, angepasst. Dieses wird über die gesamte Folge geschoben. An den Stellen, an denen die größten Anpassungsfehler auftreten, sind die Eckpunkte. Am Ende werden Geraden an die Abschnitte zwischen den Eckpunkten angepasst und die Schnittpunkte ergeben dann die finalen Eckpunkte der Vierecke.

Schritt 7 (Refinement): Mithilfe des Originalbildes werden die Kanten der gefundenen Vierecke erneut abgetastet, um so die Genauigkeit zu erhöhen, die durch die Verkleinerung des Bildes beeinträchtigt wurde. Der Algorithmus sucht an Stellen, die gleichmäßig auf der Kante verteilt sind, nach dem größten Gradienten entlang der Normalen. Die Anzahl der Stellen beträgt ein Achtel der Kantenlänge. Dadurch ergeben sich Stützstellen für eine erneute Berechnung von Geraden entlang der Kante, deren Schnittpunkte die neuen Eckpunkte ergeben.

Schritt 8 (Decode): Zuerst wird die Homografie zwischen den Bildkoordinaten und den erkannten Vierecken berechnet. Anhand dieser werden Abtastpunkte in das Originalbild projiziert. Die Abtastpunkte am Rand des Tags haben eine bekannte Farbe (schwarz/weiß). Dadurch kann ein Modell des Farbverlaufs erstellt werden, aus dem die Schwellwerte für die eigentlichen Datenpunkte generiert werden. Die Tag-Familie gibt Informationen darüber an, wo sich die bekannten Punkte und die Datenpunkte befinden. Die Decodierung einer gültigen ID gibt auch die Ausrichtung des Tags an.

Schritt 9 (Pose Estimation): Bei diesem Schritt werden die Kameraparameter genutzt. Nachdem zuvor die Homografie berechnet wurde, kann damit auch die Position und Drehung relativ zur Kamera bestimmt werden. Mithilfe eines iterativen Verfahrens, werden die Rotationsmatrix und der Translationsvektor berechnet.

2.2 Entwicklungsumgebung

Als Entwicklungsboard wird ein Zybo Z7-20 von Digilent verwendet. Dieses besitzt bereits einen CSI2 Kameraanschluss. Daher muss kein eigenes Board entwickelt werden. Auf dem Zybo Z7-20 sitzt ein Xilinx Zynq-7020. Für die Erstellung der IP-Blöcke, die Teil der Verarbeitung sind, wird die HLS von Xilinx verwendet. Damit lässt sich aus C++-Code die Hardwarebeschreibung synthetisieren. Um die IP-Blöcke zu einem System im FPGA zu integrieren, wird der Block-Designer in Xilinx Vivado genutzt. Die Software wird als Standalone-Anwendung mit dem Xilinx SDK erstellt.

2.3 Vivado High-Level-Synthesis

2.3.1 Übersicht

Xilinx liefert bei seinen Entwicklungstools die Vivado High-Level-Synthesis mit, die dazu genutzt werden kann, eine Hardwarebeschreibung aus C/C++-Code zu generieren. Das Design kann mit gewöhnlichem C/C++-Code, SystemC oder der OpenCL Application Programming Interface (API) beschrieben werden. Neben der sogenannten C-Synthese, die den Code in VHDL oder Verilog transformiert, gibt es Funktionen, die das Testen mithilfe einer Simulation ermöglichen. Zum einen kann das Design mit einer Testbench als normales Programm in der sogenannten C-Simulation getestet werden und zum anderen können aus der Testbench Testdaten extrahiert werden, die für eine Simulation der generierten Hardwarebeschreibung, der C/RTL Co-Simulation, genutzt werden. Der User Guide 902 [11] von Xilinx beschreibt ausführlich alle Aspekte der High-Level-Synthesis.

2.3.2 Unterstützte Konstrukte

Unterstützt werden fast alle Sprachkonstrukte vor dem C++11-Standard. Allerdings muss das Design in sich geschlossen sein und alle Ressourcen, die genutzt werden, müssen vollständig beim Compilieren bzw. bei der C-Synthese spezifiziert sein. Das führt dazu, dass sich dynamischer Speicher und Betriebssystemaufrufe, wie z.B. das Öffnen von Dateien oder das Schreiben in die Standardausgabe, nicht synthetisieren lassen. Anstatt Arrays mit dynamischer Länge zu erstellen, müssen diese mit fester Länge deklariert werden. Es sollte darauf geachtet werden, dass die Länge ausreichend groß für die Zwecke des Algorithmus gewählt wird. Einfache Betriebssystemaufrufe, die keine Auswirkung auf die Ausführung haben (z.B. printf) werden bei der C-Synthese automatisch ignoriert. Es wird jedoch empfohlen, alle Aufrufe durch den Präprozessor zu deaktivieren. Dazu kann das Makro `__SYNTHESIS__` verwendet werden, das nur während der C-Synthese definiert ist.

Bei polymorphen Klassen ist nur die statische Bindung möglich. Funktions- und Methodenaufrufe, die erst zur Laufzeit gebunden werden, widersprechen der Regel,

dass alle Ressourcen spezifiziert sein müssen. Grund dafür ist, dass Aufrufe dadurch realisiert sind, dass diese durch die Instanziierung einer Komponente im HDL-Code abgebildet werden. Dadurch ist es auch nicht möglich, Funktionszeiger zu benutzen. Stattdessen kann auf Templates und Funktoren zurückgegriffen werden, um generisch zu programmieren und gleichzeitig das Verhalten statisch für die C-Synthese festzulegen.

Weiterhin werden nur Konvertierungen von Zeigern auf den eingebauten Datentypen erlaubt. So ist die Konvertierung von z.B. `int*` nach `char*` in Ordnung, jedoch nicht von `mystruct*` zu `int*`.

Die Elemente eines Arrays aus Zeigern dürfen nur auf Skalare oder Arrays aus Skalaren zeigen. Nicht synthetisierbar ist ein Array aus Zeigern, die wieder auf Zeiger verweisen.

Rekursive Aufrufe einer Funktion sind auch nicht möglich. Die Algorithmen, die dies benötigen, müssen daher umgewandelt werden.

Aufgrund der vorherigen Einschränkungen können auch viele Teile der C++ Standard Template Library (STL) nicht benutzt werden. Stattdessen muss auf eigene Implementierungen zurückgegriffen werden. Anstelle der C-Mathe-Bibliotheken math.h bzw. cmath.h wird empfohlen, die HLS-Version hls_math.h zu benutzen. Alle Varianten können benutzt werden, aber zwischen den beiden Simulationsarten kann es bei den Standardbibliotheken zu kleinen Abweichungen kommen. Die HLS-Bibilothek spiegelt die genaue Implementierung auf dem FPGA wieder.

Neben der Synthese aus C/C++ Code werden Designs, die mit SystemC oder der OpenCL API beschrieben werden, auch unterstützt.

2.3.3 Workflow

Ein HLS-Projekt besteht aus den Design-Sources, der Testbench und den Solutions. In einer Solution werden die Synthese- und Simulationseinstellungen gespeichert. Dazu zählen das Ziel-FPGA, die Taktperiode, Compileroptionen und Direktiven zur Optimierung. Es gibt die Möglichkeit, mehrere Solutions anzulegen, um verschiedene Varianten zu vergleichen. Jede liegt in einem eigenen Unterordner des Projekts und enthält alle Ausgabedateien und Reports der Synthese, der Simulationen und des IP-Exports.

Zunächst wird der Algorithmus unter Beachtung der im letzten Abschnitt beschriebenen Einschränkungen in den Design-Sources implementiert. Es muss genau eine Top-Funktion angegeben werden, die die gesamte Funktionalität enthält.

Die Testbench ist optional und wird daher bei der C-Synthese nicht beachtet. Diese wird nur für die Simulationen genutzt. Dennoch ist es vorteilhaft, eine zu schreiben, um die Korrektheit zu überprüfen. In der Testbench muss die main-Funktion definiert sein, die die Tests aufruft. Die Top-Funktion kann beliebig oft mit verschiedenen Eingabedaten ausgeführt werden, um verschiedene Situationen zu testen.

Für die C-Simulation werden die Design-Sources und die Testbench zusammen als gewöhnliches Programm compiliert. Dieses wird ausgeführt und der Rückgabewert überprüft. Bei einer Null sind die Tests erfolgreich und bei jedem anderen Wert ist ein Fehler aufgetreten. Die C-Simulation ist auch die schnellste Methode, das Design zu überprüfen, da das Compilieren und die Ausführung nur einen Bruchteil der Zeit dauert, die die C-Synthese und die Co-Simulation benötigen würde.

Bei der C-Synthese wird aus den Design-Sources die Hardwarebeschreibung generiert. Das Ergebnis wird dabei sowohl in VHDL als auch in Verilog abgespeichert. Am Ende der Synthese wird ein Report ausgegeben. Dieser enthält die geschätzten Werte des Timings und des Ressourcenverbrauchs. Aus diesem geht auch hervor, welche Anweisungen im Code welche Ressourcen nutzen.

Die Co-Simulation verbindet das synthetisierte Design mit der C/C++-Testbench. Das HLS-Tool führt die Testbench aus und extrahiert dabei die Parameter der Aufrufe der Top-Funktion. Diese werden dazu als Eingabedaten genutzt, um eine RTL-Simulation der generierten Hardwarebeschreibung durchzuführen. Mehrfache Aufrufe in der Testbench führen auch zu mehrfachen Aufrufen während der Simulation, mit denen die Latenz und das Intervall zwischen den Aufrufen gemessen werden. Die Ausgabedaten werden aufgezeichnet und in die Testbench als Ergebnis der Top-Funktion injiziert, die diese überprüft. Wie bei der C-Simulation gibt der Rückgabewert der main-Funktion an, ob die Co-Simulation erfolgreich ist.

Sobald das Design den Anforderungen entspricht, kann es als IP-Block für Vivado exportiert werden. Dabei kann zwischen der VHDL und Verilog Version ausgewählt werden. Optional kann der exportierte IP-Block für den Ziel-FPGA synthetisiert und implementiert werden. Der dadurch erstellte Report gibt Auskunft über die ungefähre Anzahl der benötigten Ressourcen und, ob das geforderte Timing eingehalten wird.

2.3.4 Direktiven und Performance Optimierung

Mit Direktiven kann das Verhalten der C-Synthese beeinflusst werden. Dadurch ist es möglich das Design in zwei verschiedene Richtungen zu optimieren: Größe und Datendurchsatz. Es gibt zwei Möglichkeiten, diese im Design zu spezifizieren. Zum einen können sie als TCL-Befehle in einem Script aufgelistet sein, zum anderen können sie auch als Pragma direkt im Code eingebettet sein. Im Folgenden werden einige Konzepte erklärt, die bei der Implementierung genutzt werden.

In C/C++ können Parameter nur als Wert, Zeiger oder Referenz übergeben werden. Durch Interface-Direktiven lässt sich die Art der IP-Ports, die schließlich in der Hardwarebeschreibung verwendet werden, festlegen. Beispielsweise wird durch setzen der Direktive `INTERFACE axis port=input` der Parameter input als AXI4-Stream (axis) implementiert. Weitere in dieser Arbeit verwendete Typen sind `s_axilite`, durch dem die Paramater durch ein AXI4-Lite-Interface ansprechbar sind, und `m_axi`, der einen AXI4-Master erstellt, um z.B. auf dem Hauptspeicher zuzugreifen.

Durch die Direktive `PIPELINE` wird eine Pipeline erzeugt, die mehrere Daten parallel

in den einzelnen Stufen verarbeitet. Dadurch lässt sich der Datendurchsatz steigern. Eine Funktion hat beispielsweise drei Operationen, die jeweils einen Taktzyklus dauern. Ohne Pipelining benötigt die Funktion daher drei Taktezyklen, bis die nächsten Daten eingelesen werden können (Initiation Inerval (II) = 3, Latenz = 3). Nach dem ersten Taktzyklus hat die erste Operation nichts mehr zu tun. Daher können mithilfe des Pipelinings bereits im zweiten Taktzyklus neue Daten in der ersten Stufe verarbeitet werden, während die zweite Stufe das Ergebnis der ersten Stufe auswertet. Dadurch verkürzt sich das II auf 1 mit gleichbleibender Latenz. Durch kürzere Taktperioden kann die Datenrate weiter gesteigert werden. Das führt jedoch zu mehr Stufen in der Pipeline und dadurch zu einer höheren Latenz.

	Daten					
Operation 1	A			B		
Operation 2		A			B	
Operation 3			A			B
Takt	1	2	3	4	5	6

	Daten					
Operation 1	A	B	C	D	E	F
Operation 2		A	B	C	D	E
Operation 3			A	B	C	D
Takt	1	2	3	4	5	6

Tabelle 2.1: Scheduling der Operationen einer Funktion. Links: Ohne `PIPELINE`-Direktive. Rechts: Mit `PIPELINE`-Direktive.

Die `DATAFLOW`-Direktive ist ähnlich zur Vorherigen und lässt mehrere Unterfunktionen statt Operationen parallel ablaufen, die untereinander mit Pingpong-Buffern oder FIFOs verbunden werden.

Standardmäßig werden die Elemente einer Struktur jeweils für sich behandelt. Das bedeutet, dass z.B. ein Array einer Struktur mit zwei Elementen durch zwei Speicher im Block-RAM abgebildet wird, die jeweils die Breite der Elemente besitzen. Durch die `DATAPACK`-Direktive wird die Struktur zusammengefasst. Dadurch entsteht ein Speicher mit der gesamten Strukturbreite. Dies spart Block-RAM ein. Allerdings müssen dadurch alle Elemente gleichzeitig geschrieben und gelesen werden.

Mit `ARRAY_PARTITION factor=N block/cyclic/complete` gibt es noch eine Direktive, die ein Array auf getrennte Block-RAM-Zellen aufteilt. Für die Option `complete` ist jedes Element des Arrays eigenständig und damit auch parallel zugreifbar. Daher wird dies für gewöhnlich nicht als Block-RAM, sondern als Register synthetisiert. Dies eigne sich besonders für kleine Arrays, dessen Elemente gleichzeitig geschrieben

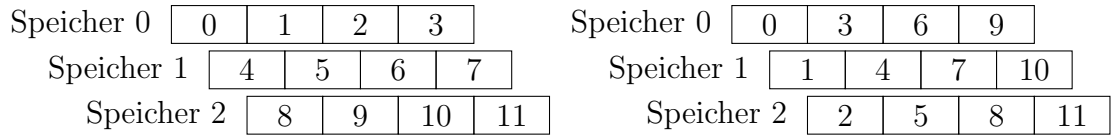

Abbildung 2.1: Optionen der Direktive `ARRAY_PARTITION`. Links: `block`. Rechts: `cyclic`

oder gelesen werden. Ein Beispiel dafür sind Schieberegister. Für die übrigen Optionen gibt N die Anzahl der Speicher an. Während bei der `block`-Option das Array in N gleichgroße Speicher aufgeteilt wird, werden bei der `cyclic`-Option jeweils N aufeinanderfolgende Elemente zyklisch auf die verschiedenen Speicher aufgespalten (s. Abb. 2.1). Dadurch kann das Pipelining durch parallele Zugriffsmuster optimiert werden.

3 Anforderungsanalyse

3.1 Systemarchitektur

Im Vorprojekt ist eine Systemarchitektur entstanden (s. Abb. A.1) [4]. Darin sind alle benötigten Funktionen als Blöcke abgebildet. Zwischen den Blöcken müssen Daten ausgetauscht werden, die durch Pfeile gekennzeichnet sind. In einer detaillierteren Spezifikation der Blöcke wird deren Aufgabe beschrieben (s. Unterabschnitt A.2.1). Darunter fallen auch die Daten, die in den Block hineingehen und vom Block ausgegeben werden.

Die offizielle Version des AprilTag-Algorithmus wurde untersucht und in Teilschritte gegliedert. Jeder dieser Schritte wird durch einen Block dargestellt. Das Ziel der Untersuchung war es, die Blöcke zu identifizieren, die sich auf dem FPGA implementieren lassen und dadurch die Laufzeit beschleunigen.

Der erste Schritt (Decimate) verarbeitet das Bild sequenziell. Daher lässt sich dieser Schritt gut als Pipeline im FPGA implementieren. Dasselbe gilt auch für die nächsten beiden Schritte. Bei diesen müssen allerdings mehrere Zeilen zwischengespeichert werden, da sich der neue Wert eines Pixels aus der Umgebung berechnet. Die Verarbeitung des BlurSharpen-Schritts ist durch eine 2D-Faltung realisiert. Dafür gibt es bereits eine Implementierung in Vivado HLS. Auch bei dem Threshold-Schritt basiert die Verarbeitung auf mehreren Faltungen, die aber in der benötigten Form nicht bereitstehen. Daher müssen diese selbst implementiert werden. Für das Connected-Components-Labeling (CCL) gibt es verschiedene Verfahren, die für FPGAs entwickelt worden sind. Das in dieser Arbeit genutzte Verfahren basiert, wie die offizielle Version, auf einer UnionFind-Datenstruktur [9]. Der GradientCluster-Schritt wurde noch einmal in einen FPGA- und CPU-Teil aufgeteilt. Der FPGA-Teil durchsucht das Bild sequenziell nach Grenzpixeln ab, die an die CPU ausgegeben werden. Diese sortiert die Daten dann entsprechend ihrer Zugehörigkeit. Die Hash-Tabelle, die dafür genutzt wird, lässt sich effizienter durch die CPU erstellen. Die übrigen Schritte nutzen intensiv dynamischen Speicher, sowie einige Sortieralgorithmen. Allgemein ist der Speicherzugriff weniger vorhersagbar als bei der sequenziellen Bildverarbeitung. Daher verbleiben diese auf der CPU.

Dadurch, dass die ersten Schritte auf dem FPGA und die nachfolgenden auf der CPU ausgeführt werden, können diese parallel laufen. Das führt dazu, dass während die CPU das erste Bild verarbeitet, der FPGA bereits beim zweiten Bild ist.

3.2 Funktionale Anforderungen

- Das System muss AprilTags in einem vorgegebenen Bild finden.
- Das System muss AprilTags in einem von einer Kamera empfangenen Bild finden.
- Das System muss die ID aller gefundenen AprilTags bestimmen.
- Das System muss die Lage aller gefundenen AprilTags relativ zur Kamera bestimmen.
- Das System muss die Lage und ID aller gefundenen AprilTags ausgeben.
- Funktionale Anforderungen für die einzelnen Blöcke sind in der Spezifikation zum Blockdiagramm gegeben (s. A.2.1).
- Die AprilTag-Verarbeitungskette muss durch die Software initialisiert werden.
- Die Schnittstelle der AprilTag-Verarbeitungskette muss die Ergebnisse bereitstellen

3.3 Nichtfunktionale Anforderungen

- Das System muss auf einem Xilinx Zynq SoC laufen
- Jeder Block soll unabhängig durch automatisierte Tests getestet sein.
- Das System sollte fähig sein, mit der FPGA-Implementierung mehr Bilder pro Sekunde als die reine Softwareimplementierung zu verarbeiten.
- Die Schnittstelle der AprilTag-Verarbeitungskette sollte sich von außen nicht zwischen Software- und FPGA-Implementierung unterscheiden.
- Die AprilTag-Verarbeitungskette muss eine gemeinsame Schnittelle bereitstellen, über die die Verarbeitung gesteuert werden kann.
- Es ist keine Integration in ein übergeordnetes System vorgesehen.
- Es ist nicht notwendig, die HDMI Ausgabe zu optimieren und das Ergebnis mit derselben Bildwiederholrate des eigentlichen Erkennungsvorgangs auszugeben.

4 Realisierung der IP-Cores

4.1 Allgemeine Struktur

In den HLS-Projekten befinden sich jeweils drei Quellcodedateien. Die Implementierung der Top-Level-Funktion und deren Unterfunktionen ist in der <*Projektname*>.cpp-Datei zu finden. Die <*Projektname*>.h-Datei enthält Deklarationen der Datentypen, die benutzt werden, sowie den Prototypen der Top-Level-Funktion. Beide Dateien sind für die Synthese im Projekt eingebunden. Die tb_<*Projektname*>.cpp-Datei ist die Testbench, die für die Ausführung der Simulation zuständig ist.

Für die Übertragung der Bilddaten zwischen den IP-Blöcken wird das AXI-Video-Protokoll verwendet [2]. Es basiert auf dem AXI-Stream-Protokoll. Über die TDATA-Signale werden die Pixeldaten übertragen. In dieser Arbeit entspricht die Datenbreite des Streams die der Pixel. Somit wird pro Takt ein Pixel übertragen. Die Signale TVALID und TREADY sind für das Handshake vorgesehen. Bei der Videoübertragung steht das Signal TUSER für den Start of Frame (SOF) und ist für das erste Pixel im Bild gesetzt. Das Signal TLAST kennzeichnet, wenn es gesetzt ist, das letzte Pixel einer Bildzeile (End of Line (EOL)).

Die Top-Level-Funktion ist als Dataflow-Region gekennzeichnet. Damit wird erreicht, dass die Unterfunktionen parallel ausgeführt werden und die Daten per FIFO oder Stream von einer Funktion zur nächsten gelangen. Für die Videoverarbeitung wird zunächst die hls::AXIvideo2Mat Funktion aufgerufen. Diese ist für das AXI-Video-Protokoll zuständig und überprüft, ob das eingehende Bild der angegebenen Größe entspricht. Ausgegeben wird ein hls::Mat, über das auf den Pixel-Stream zugegriffen werden kann. Dieses wird an die Verarbeitungsfunktion übergeben. Am Ende wird die Funktion hls::Mat2AXIvideo aufgerufen, die die Bilddaten mit dem AXI-Video-Protokoll wieder ausgibt.

Das Bild wird zeilenweise übertragen. Daher werden zwei verschachtelte for-Schleifen genutzt, um das Bild zu verarbeiten. Die äußere Schleife zählt die Anzahl der Zeilen und die innere Schleife die Anzahl der Spalten. Im Folgenden werden diese als äußere bzw. innere Pixelschleifen bezeichnet.

Für alle IP-Blöcke, die einen Digital Signal Processor (DSP)-Slice nutzen, ist das Konfigurationskommando `config_core -latency 2 DSP48` gesetzt. Aufgrund der kurzen Taktperiode von 6,67ns (150MHz) ist die Signallaufzeit der Operationen, die auf einem DSP-Slice abgebildet sind, zu lang für einen Takt. Daher wird die Latenz auf zwei Takte erhöht.

4.2 Vorverarbeitung

4.2.1 Decimate

Als Eingabe wird ein RGB-Stream mit acht Bit pro Kanal erwartet. Der Algorithmus arbeitet aber mit Graustufenbildern. Daher wird zunächst aus den RGB-Kanälen die Luminanz berechnet [6]:

$$Y = 0.299 * R + 0.587 * G + 0.114 * B \tag{4.1}$$

Um dies auf dem FPGA mit ganzzahligen Variablen zu berechnen, werden die Koeffizienten mit 256 multipliziert und das Ergebnis am Ende durch 256 dividiert:

$$Y = \frac{77 * R + 150 * G + 29 * B}{256} \tag{4.2}$$

Die abschließende Division kann effizient im FPGA durch Abschneiden der unteren acht Bits erfolgen.

Es gibt zwei Zähler, die von Null anfangen zu zählen und beim Erreichen des Decimate-Faktors zurückgesetzt werden. Der erste Zähler wird für jedes Pixel inkrementiert, der zweite Zähler nach dem Ende einer Zeile. Nur wenn beide auf Null stehen wird ein Pixel ausgegeben. Der Original-Stream ist davon nicht betroffen und gibt jedes Pixel als Grauwert aus.

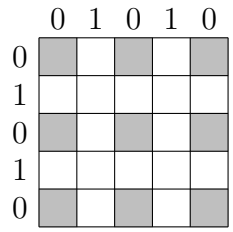

Abbildung 4.1: Bei einem Decimate-Faktor von 2 wird nur jedes zweite Pixel aus jeder zweiten Zeile verarbeitet. Graue Pixel sind aktiv.

Die Optimierungsdirektive `LOOP_FLATTEN off` sorgt dafür, dass die beiden verschachtelten Schleifen nicht zu einer Schleife mit einem gemeinsamen Zähler zusammengefasst werden. Dadurch würde sonst das HLS-Tool das geforderte Timing nicht erreichen. Sobald die Zähler im Körper nicht verwendet werden, kann das Timing erreicht werden. Diese werden aber benötigt, um auf Zeilenende und Bildende zu prüfen.

Des Weiteren ist die Direktive `PIPELINE II=1` gesetzt. Dadurch wird die Berechnung als Pipeline implementiert, die in jedem Takt (II=1) ein Pixel einlesen kann und dadurch in jeder Pipelinestufe ein anderes Pixel verarbeitet.

In der Testbench wird zunächst ein Testbild mit aufsteigenden Pixelwerten generiert. Dieses wird der decimate-Funktion übergeben. Am Ende wird überprüft, ob das in

Graustufen konvertierte Originalbild und das verkleinerte Bild die richtigen Größen und Bildinhalte haben.

Das finale Design nutzt nur einen sehr geringen Teil der verfügbaren Ressourcen (s. Tabelle 4.1). Mit dem DSP-Slice wird die Berechnung des Grauwerts eines Pixels durchgeführt.

Typ	Anzahl	Verwendung (%)
BRAM_18K	0	0
DSP48	2	0,9
FF	863	0,8
LUT	1343	2,5

Tabelle 4.1: Ressourcen des Decimate-IP nach der C-Synthese. Timing: 5,22ns (Soll: 6ns)

4.2.2 BlurSharpen

Da für die Weichzeichnung bzw. das Schärfen eine 2D-Faltung durchgeführt wird, muss nur die Funktion hls::Filter2D aufgerufen werden. Der Kernel, der die Art der Filterung bestimmt, wird von der Software über die AXIlite-Schnittstelle gesetzt.

Der Kernel wird durch die zweidimensionale Gauß-Funktion bestimmt:

$$k(x,y) = e^{\frac{x^2+y^2}{2\sigma^2}}, \quad x,y \in \{-2,-1,0,1,2\} \tag{4.3}$$

Dadurch wird das Bild weichgezeichnet. Um das Bild zu schärfen, werden alle Elemente des Kernels negiert und anschließend das Zentrum um zwei inkrementiert. Das bildet die Funktionsweise ab, wie sie in der offiziellen Version implementiert ist: Die Pixelwerte des Originalbildes werden verdoppelt und danach wird das weichgezeichnete Bild pixelweise abgezogen. Der Unterschied ist, dass auf dem FPGA aufgrund eines geringeren Ressourcenverbrauchs die einzelnen Schritte im Kernel zusammengefasst werden.

Als Datentyp für den Kernel werden zwölf Bit Fixed-Point Variablen verwendet. Vier der Bits repräsentieren den ganzzahligen Anteil. Der Bereich der damit darstellbaren Zahlen ist für einen Gauß-Kernel ausreichend. In der Software wird dieser normalisiert, wodurch die Beträge aller Elemente kleiner als eins sind. Nur das Zentrum nimmt größere Werte an, falls das Bild geschärft werden soll.

Aufgrund der vielen Multiplikationen benutzt das Design die meisten DSPs gegenüber den anderen IP-Blöcken (s. Tabelle Tabelle 4.2). Der Block-RAM wird zur Zwischenspeicherung der Zeilen genutzt.

Typ	Anzahl	Verwendung (%)
BRAM_18K	7	2
DSP48	25	11
FF	3735	3
LUT	5614	10

Tabelle 4.2: Ressourcen des BlurSharpen-IP nach der C-Synthese. Timing: 5,25ns (Soll: 6ns)

4.2.3 Threshold

Die Segmentierung in helle und dunkle Bereiche, sowie Bereiche ohne ausreichenden Kontrast erfolgt durch drei Unterfunktionen. In `makeTiles`, der ersten Unterfunktion, wird das Bild in vier mal vier Pixel große Kacheln aufgeteilt. Alle sechzehn Pixel sowie das Maximum und das Minimum werden als ein 144 Bit breites Pixel an die nachfolgende Funktion weitergegeben. Da das Bild zeilenweise eingelesen wird, müssen vier Zeilen in einem Zeilenpuffer (`buffer`) zwischengespeichert werden. Ein weiterer Zeilenpuffer (`minMaxBuffer`) mit nur einer Zeile speichert temporäre Minima und Maxima. Es werden immer vier Pixel, die zu einer Kachel gehören, in einem Schleifendurchlauf gelesen und im `buffer` abgelegt. Deswegen zählt die innere Schleife nur bis zu einem Viertel der Bildbreite. Mit deren Minimum und Maximum wird der `minMaxBuffer` an der entsprechenden Spaltenposition aktualisiert bzw. in der ersten Zeile neu gesetzt. Während der vierten Zeile sind die Kacheln vollständig und werden an die nachfolgende Funktion weitergegeben.

Die innere Schleife erhält die Optimierungsdirektive `PIPELINE II=4`. Dadurch wird erreicht, dass vier Pixel pro Schleifendurchlauf eingelesen werden. Für den `minMaxBuffer` ist die `DATA_PACK` Direktive gesetzt, die dafür sorgt, dass die zugrunde liegende Struktur, die aus zwei Acht-Bit-Variablen besteht, im FPGA als ein Speicher mit 16 Bit Datenbreite anstatt zwei Speichern mit jeweils acht Bit abgelegt wird und daher weniger BRAM-Zellen benötigt. Durch die Direktive `ARRAY_PARTITION variable=buffer cyclic factor=4 dim=2` wird der Speicher in der zweiten Dimension des Zeilenpuffers in vier einzelne zerlegt, sodass jeder vierte Eintrag zu demselben Speicher gehört (`cyclic`). Dadurch kann im C++-Code auf vier hintereinander liegende Einträge parallel zugegriffen werden.

Die zweite Funktion ist dafür zuständig, dass auf die Minima des Kachelbildes die Erosion und auf die Maxima die Dilatation angewendet wird. Dazu werden zwei Zeilenpuffer benötigt. Der erste speichert für zwei Zeilen nur die Minima und Maxima des Kachelbildes, während der zweite Zeilenpuffer für eine Zeile den gesamten Kachelinhalt speichert. Der Algorithmus arbeitet mit einem Fenster aus 3x3 Kacheln. Dabei ist der Mittelpunkt das zu verarbeitende Element und unten rechts befindet sich das neu eingelesene Element. Während der ersten Zeile ist der Mittelpunkt außerhalb des Bildes, sodass nur die beiden Zeilenpuffer gefüllt werden. Ab der

zweiten Zeile und Spalte kann die eigentliche Verarbeitung starten. Es gibt jeweils ein Verarbeitungsfenster für die Minima und Maxima. Die Elemente des Fensters werden pro Spalte immer um eins nach links verschoben und die drei Elemente auf der rechten Seite aus dem Zeilenpuffer geladen. Damit das Bild komplett verarbeitet wird, zählen die Schleifen jeweils eine Zeile und Spalte mehr. In diesen Fällen befinden sich die zu ladenden Elemente außerhalb des Bildes und werden mit durch die größte Zahl beim Minimafenster bzw. kleinste Zahl beim Maximafenster ersetzt. Auf diese Weise beeinflussen sie nicht das Ergebnis der Minima-/Maximaberechnung, die das kleinste Minimum und das größte Maximum im Fenster sucht. Die beiden Werte sind dann das neue Minimum und Maximum der zu verarbeitenden Kachel, die aus dem entsprechenden Zeilenpuffer gelesen und ausgegeben wird.

Die innere Schleife erhält die Optimierungsdirektive `PIPELINE II=1`, damit pro Takt eine Kachel eingelesen wird. Die `DEPENDENCE variable=minMaxLineBuf inter false` Direktive für den Minima/Maxima-Zeilenpuffer gibt an, dass durch die Programmierung sichergestellt ist, dass in einem Schleifendurchgang ein Element geschrieben und im nächsten direkt gelesen wird. In diesem Fall wird ein Element nach einem Schreibvorgang erst in der nächsten Zeile wieder gelesen. Der Grund für diese Einschränkung ist, dass das Schreiben zwei Taktzyklen benötigt, aber einen Takt später schon gelesen wird. Das HLS-Tool kann diese Bedingungen nicht korrekt automatisch prüfen. Die `ARRAY_PARTITION variable=minMaxLineBuf complete dim=1` Direktive sorgt dafür, dass auf beiden Zeilen des Zeilenpuffers gleichzeitig zugegriffen werden kann. Für beide Zeilenpuffer ist wieder die Direktive `DATA_PACK variable=minMaxLineBuf` gesetzt, um Block-RAM zu sparen. Da der Zeilenpuffer genau eine Zeilenlänge abspeichern kann, die älteste Kachel aber erst eine Spalte später benötigt wird, wird diese durch ein Schieberegister zwischengespeichert. Das wird durch ein Array mit der Direktive `ARRAY_PARTITION variable=lastTiles complete` und verkettete Zuweisung der Elemente erreicht.

Die letzte Funktion, `applyThreshold`, ermittelt aus den Minima und Maxima, ob der Kontrast ausreichend hoch ist. Falls das zutrifft, wird ein Schwellwert, der sich als Mittelwert zwischen Minimum und Maximum berechnet, auf alle Pixel der Kachel angewendet. In der ersten Zeile werden die Kacheln von der vorhergehenden Funktion eingelesen und jeweils die ersten vier Pixel, die ursprünglich zur ersten Zeile gehörten, werden verarbeitet und ausgegeben. Die übrigen Pixel sowie das Minimum und Maximum werden in Zeilenpuffern abgelegt. In den drei darauf folgenden Zeilen werden die Pixel wieder ausgelesen und verarbeitet, ohne neue Kacheln einzulesen. Beide Schleifen zählen bis zur vollen Größe des Bildes. Wie in der ersten Funktion ist die OptimierungsDirektive `PIPELINE II=4` für die innere Schleife gesetzt, da vier Pixel pro Schleifendurchlauf ausgegeben werden sollen, genauso wie die `DATA_PACK` Direktive für den `minMaxBuffer`. Der Zeilenpuffer erhält die Direktive `ARRAY_PARTITION variable=buffer complete dim=1`. Die erste Dimension stellt die Zeilen dar. Diese sollen in jeweils eigene Speicher im FPGA abgebildet werden, damit beim Abspeichern einer Kachel alle drei Zeilen gleichzeitig geschrieben werden können. Die zweite

Dimension wird nicht unterteilt, weil ausreichend Takte bis zum nächsten Einlesen zur Verfügung stehen und bei der Ausgabe jeweils genau ein Pixel pro Takt zur Verarbeitung ausgelesen wird.

Zum Testen des IP-Blocks wird ein Testbild generiert. Dieses wird von der offiziellen Version des Algorithmus verarbeitet und generiert ein Referenzbild. Anschließend läuft die FPGA-Funktion mit dem Testbild und das Ergebnis wird mit der Referenz verglichen.

Da aufgrund der sequentiellen Verarbeitung viele Zeilen zwischengespeichert werden, nutzt das Design die meisten RAM-Blöcke des internen Speichers (s. Tabelle 4.3).

Typ	Anzahl	Verwendung (%)
BRAM_18K	15	5
DSP48	0	0
FF	4310	4
LUT	7046	13

Tabelle 4.3: Ressourcen des Threshold-IP nach der C-Synthese. Timing: 5,25ns (Soll: 6ns)

4.3 Connected Components Labeling

4.3.1 Funktionsweise

Ein Connected-Components-Labeling-Algorithmus soll zusammenhängende Regionen im Bild markieren. Zusammenhängend sind zwei Pixel, wenn beide zur selben Klasse gehören und sie sich in vierer bzw. achter Nachbarschaft befinden. Sie sind Teil einer Komponente, die aus allen Pixeln besteht, die untereinander durch einen Pfad von zusammenhängenden Pixeln verbunden sind und nicht mehr erweitert werden können. Das heißt, dass keines der Pixel einen Nachbarn besitzen darf, der zur selben Klasse gehört, aber nicht zur Komponente. Ihr wird dann eine ID zugewiesen. In der sogenannten Labelmaske, der Ausgabe des Algorithmus, entspricht der Wert eines Pixels der ID der Komponente, zu der es gehört.

Es gibt verschiedene Algorithmen, die das Problem lösen. In [5] wird ein Verfahren vorgestellt, dass das Bild mehrfach von vorne und hinten durchläuft. Dies lässt sich zwar einfach auf einem FPGA implementieren, hat aber je nach Anzahl der benötigten Durchläufe eine lange Laufzeit. Ein Two-Pass-Verfahren, das das Bild genau zweimal durchläuft, wird von Schwenk und Huber [9] vorgestellt. Es nennt sich Two-Pass-Single-Storage (TPSS). Wie der Name aussagt, wird nur ein Zwischenspeicher für ein ganzes Bild benötigt. Klassische Verfahren wie von Rachakonda, Athanas und Abbott benötigen einen Zwischenschritt. Bei dem ersten Durchlauf können Konflikte auftreten. Dies passiert z.B. bei einem „U“ im Bild (s. Abb. 4.2). Die beiden vertikalen

Striche werden zunächst als zwei verschiedene Komponenten erfasst. Sobald an der unteren Linie erkannt wird, dass diese jedoch zusammengehören, müssen beide Labels zusammengefasst werden. Diese Konflikte werden in einer Konflikttabelle gespeichert, aus der im Zwischenschritt eine Loop-Up-Tabelle erstellt wird. Im zweiten Durchlauf wird diese benutzt, um die temporären Labels zu ersetzen.

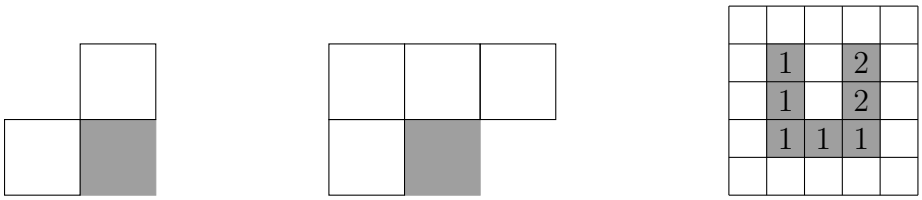

Abbildung 4.2: Links: Maske für Vierer-Konnektivität. Mitte: Maske für Achter-Konnektivität. Rechts: Kollision bei „U“-Form mit Vierer-Konnektivität.

Im Gegensatz dazu basiert der TPSS-CCL-Algorithmus wie die offizielle Version des AprilTag-Algorithmus auf eine UnionFind-Datenstruktur und wurde speziell für die Implementierung auf einem FPGA entwickelt. Dadurch kann die Look-Up-Tabelle und die Konflikttabelle wegfallen. sodass nur ein Speicher benötigt wird.

Im ersten Durchlauf wird eine temporäre Labelmaske im Speicher erzeugt. Dabei gibt der Wert des Labels die Position im Bild des ersten Pixels an, das als erstes gefunden wurde. Ein Pixel, das laut Maske (noch) keine Nachbarn hat, zeigt auf sich selbst und ist damit das erste Pixel einer Komponente. Wenn es für einen Pixel einen Nachbarn gibt, wird dessen Wert übernommen. Sollte es mehr als einen Nachbarn geben, die auf verschiedene Startpunkte zeigen, wird der kleinste Wert übernommen. Dies stellt einen Konflikt dar und die Startpunkte der übrigen Komponenten werden so geändert, dass sie auch auf diesen zeigen (s. Abb. 4.3).

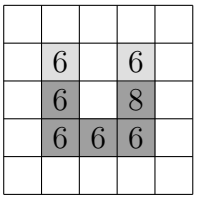

Abbildung 4.3: Labelmaske nach dem ersten Schritt (UnionFind) des TPSS-CCL Algorithmus. Die oberen Enden sind jeweils Startpunkte. Nach der Konfliktlösung zeigt der rechte auf den linken Startpunkt. Das Pixel rechts-mittig zeigt aber weiterhin auf dem darüber.

Der zweite Durchlauf liest die temporäre Labelmaske ein und erstellt für jedes Pixel, das auf sich selbst zeigt, ein neues Label. Für alle anderen wird das Label übernommen, auf das gezeigt wird.

4.3.2 Implementierung

Da der zweite Durchlauf erst starten kann, wenn der erste fertig ist, würde immer einer der beiden nicht aktiv sein. Um das zu umgehen und die Bildrate zu erhöhen, sollen beide parallel an unterschiedlichen Bildern arbeiten. Nachdem der erste Durchlauf mit einem Bild fertig ist, startet dieser direkt mit dem nächsten Bild und der zweite Durchlauf verarbeitet das soeben erstellte Ergebnis des ersten Bildes. Deshalb werden beide in eigene IP-Blöcke untergebracht. Es werden zwei Framebuffer genutzt. Nachdem der erste IP-Block die Verarbeitung abgeschlossen hat, gibt dieser den Framebuffer aus, in dem geschrieben wurde. Das valid-Signal wird gesetzt. Der zweite IP-Block wartet vor der Verarbeitung auf das Signal und übernimmt den Framebuffer. Nach der Bestätigung des Empfangs mit dem ack-Signal fängt der erste IP-Block mit dem nächsten Bild an und der zweite startet mit dem aktuellen. Der Framebuffer wird als bool-Parameter der Top-Funktionen deklariert. Mit der Direktive `INTERFACE ap_hs port=framebuffer` werden die Handshake-Signale (valid, ack) automatisch vom HLS-Tool generiert.

Im Gegensatz zu den anderen Bildverarbeitungsverfahren gibt der erste IP-Block keinen Stream aus, sondern speichert das Ergebnis durch ein AXI-Master-Interface im Hauptspeicher ab. Die Software legt die Basisadresse der Framebuffer fest. Intern ist in der Funktion eine statische bool-Variable deklariert, die den Framebuffer vorgibt, auf dem gearbeitet werden soll. Das `static`-Schlüsselwort bedeutet für das HLS-Tool, dass diese beim Start nicht zurückgesetzt wird und daher den Wert immer behält. Am Ende der Funktion wird der Wert umgeschaltet und für den zweiten IP-Block ausgegeben. Wenn sie auf `true` gesetzt ist, wird die Basisadresse um ein Bild verschoben. Es handelt sich um 32 Bit große Pixel, die im Speicher abgelegt werden. Darum kann es auch bei beliebiger Bildgröße nicht zu Alignment-Fehlern kommen.

Der Algorithmus ist mit der Vierer-Konnektivität, wie er auch in [9] beschrieben wurde, implementiert. Ein Zeilenpuffer wurde angelegt, um auf das Pixel über der aktuellen Position zuzugreifen. Allerdings wird nicht das Eingabebild zwischengespeichert, sondern die Ausgabe. Dadurch muss nicht über das AXI-Master-Interface das Pixel geladen werden, das durch die Latenz stark die Performance beeinflusst. Dies ist auch der Grund dafür, dass der inneren Pixelschleife zwei weitere Schleifen untergeordnet werden. In der ersten findet die eigentliche Verarbeitung statt und in der zweiten wird das Ergebnis in den Speicher geschrieben. Letztere ist mit der Direktive `PIPELINE` gekennzeichnet. Dies bewirkt in Verbindung mit einem Speicherzugriff mit aufsteigender Addresse und fester Schleifenbegrenzung, dass der Transfer als Burst-Zugriff auf dem Bus ausgeführt wird und damit die Latenz kompensiert. Die Burst-Länge beträgt 16. Dies ist auch die Länge der Schleifen, denn es werden immer 16 Pixel am Stück verarbeitet.

Die Direktive lässt sich nicht auf die erste Unterschleife anwenden, da die head-Funktion eine Schleife enthält, die eine rekursive Suche abbildet und daher keine feste

Begrenzung hat. Es wird also ein Burst-Buffer angelegt, der die Ausgabe zwischenspeichert. Da es bei Konfliktfällen vorkommen kann, dass Pixel, die bereits verarbeitet wurden, aktualisiert werden sollen, muss überprüft werden, ob sich eine Kopie im Zeilenpuffer oder Burst-Buffer befindet. Diese müssen auch aktualisiert werden. Wenn sich das Pixel noch im Burst-Buffer befindet, muss kein Zugriff auf dem Hauptspeicher erfolgen, da diese noch nicht abgespeichert worden sind. Die head-Funktion sucht rekursiv nach dem Startpunkt einer Komponente. Rekursion wird vom HLS-Tool nicht unterstützt, deswegen wird sie mit einer Schleife implementiert. Diese lädt ein Pixel aus dem Speicher und prüft, ob es auf sich selbst zeigt. Wenn dies nicht zutrifft, wird das Pixel geladen, auf das gezeigt wird und es wird wieder überprüft.

Beim zweiten IP-Block wird das Bild aus dem Speicher gelesen und das Ergebnis wird als Stream weitergegeben. Auch hier gibt es wieder zwei Unterschleifen. Die erste lädt das Eingabebild, gibt das Ergebnis aus der vorherigen Iteration weiter und schreibt es zurück in den Hauptspeicher. Die zweite Schleife verarbeitet die eingelesenen Daten nach dem Algorithmus in [9]. Der Grund für das Zurückschreiben des Ergebnisses in den Hauptspeicher ist das Zeigen der Pixel auf vorherige Stellen, die nicht immer der linke oder obere Nachbar sind (Konfliktfall). Der Block-RAM des FPGA ist nicht ausreichend groß, um alle Ziele zu speichern. Deshalb wird die resultierende Labelmaske wieder in den Hauptspeicher geschrieben. Durch das AXI-Master-Interface kann gleichzeitig sowohl geschrieben und als auch gelesen werden, wodurch beides parallel erfolgt. Die `PIPELINE` Direktive sorgt wieder für den Burst-Zugriff. Die zweite Schleife ist auch damit gekennzeichnet. Trotzdem müssen beide getrennt sein, da in der zweiten Schleife auch einzelne, nicht sequenzielle Speicherzugriffe stattfinden. Diese können aber nicht zusammen mit den sequenziellen in einer gemeinsamen Schleife ausgeführt werden. Das HLS-Tool würde sonst keine Bursts erzeugen und dadurch die Geschwindigkeit stark reduzieren.

Voraussetzung für eine korrekte Funktion beider IP-Blöcke ist, dass die Bildbreite einem Vielfachen der Burst-Länge entspricht. Ansonsten werden im ersten IP-Block zu viele Pixel eingelesen, da in dem Fall, dass weniger als 16 in einer Zeile übrig bleiben, bereits aus der nächsten Zeile gelesen wird und sich das Bild dadurch verschiebt. Dasselbe gilt auch für den zweiten IP-Block, nur werden hier mehr Pixel ausgegeben als erwartet. Eine korrekte Burstlängen-Variation am Zeilenende wurde aus Zeitgründen nicht implementiert.

Die verwendeten Ressourcen der beiden IP-Blöcke sind in Tabelle 4.4 und Tabelle 4.5 zu finden. Die DSP-Slices werden zur Berechnung der Framebuffer-Adressen verwendet. Etwa die Hälfte der Flip-Flops und LUTs, sowie zwei der Block-RAMs werden jeweils durch den AXI-Master belegt.

Typ	Anzahl	Verwendung (%)
BRAM_18K	5	1,7
DSP48	1	0,5
FF	3931	3
LUT	4728	8

Tabelle 4.4: Ressourcen des CCL-One-IP nach der C-Synthese. Timing: 5,25ns (Soll: 6ns)

Typ	Anzahl	Verwendung (%)
BRAM_18K	8	2
DSP48	1	0,5
FF	4307	4
LUT	3643	6

Tabelle 4.5: Ressourcen des CCL-Two-IP nach der C-Synthese. Timing: 5,74ns (Soll: 6ns)

4.4 Gradient Cluster

Beim GradientClustering sollen die Subpixel zwischen einem hellen und einem dunklen Bereich ausgegeben werden. Dazu sind die vier Pixel rechts-oben, oben, links-oben und links des zu verarbeitenden Pixels für die Verarbeitung erforderlich (s. Abb. 4.2 Achter-Konnektivität). Die oberen werden durch einen Zeilenpuffer zwischengespeichert und der linke Nachbar durch eine einfache Variable. Um auf alle vier gleichzeitig zuzugreifen, wird das Element der nächsten Spalte bzw. der ersten, falls das Ende der Zeile erreicht wurde, aus dem Zeilenpuffer geladen und einem Schieberegister zugeführt. Dieses besteht aus drei Elementen, die die oberen Pixel darstellen. Für jeden Nachbarn, der innerhalb des Bildes liegt, wird überprüft, ob das Paar aus einem hellen und einem dunklen Pixel besteht. Wenn dem so ist, werden die Gradientenrichtung, die Position und die beiden Labels aus der Labelmaske ausgegeben. Das letzte Datenpaket hat alle Bits der Label- und Positionsdaten sowie das last-Flag des AXI-Streams auf eins gesetzt, um der Software bzw. durch das Flag dem DMA-IP mitzuteilen, an welcher Stelle die Daten aufhören.

Da bis zu viermal pro Schleifendurchgang Daten ausgegeben werden, erhält die innere Schleife die Direktive `PIPELINE II=4`. Für den Zeilenpuffer ist die Direktive `DEPENDENCE variable=lineBuf inter false` gesetzt, um dem HLS-Tool den Hinweis zu geben, dass durch das Design nach einem Schreibvorgang nicht direkt ein Lesevorgang folgt. Mit der Direktive `ARRAY_PARTITION variable=lastRow complete` für das Schieberegister ist gewährleistet, dass auf alle Elemente gleichzeitig zugegriffen werden kann.

Dieses Design nutzt wie das erste nur wenige Ressourcen (s. Tabelle 4.6). Im genutzten Block-RAM befindet sich die zwischengespeicherte Zeile.

Typ	Anzahl	Verwendung (%)
BRAM_18K	3	1
DSP48	0	0
FF	962	0,9
LUT	1978	3

Tabelle 4.6: Ressourcen des GradientCluster-IP nach der C-Synthese. Timing: 5,25ns (Soll: 6ns)

4.5 Integration

Das Blockdesign basiert auf ein Beispielprojekt von Digilent, das Bilder von einer angeschlossenen Kamera auf einem HDMI-Monitor ausgibt. In einem Vorprojekt wurde dies um eine Korrektur der Linsenverzeichnung ergänzt. Die Taktrate für die Video-IP-Blöcke beträgt 150MHz. Die AXI-lite-Schnittstellen, die für die Konfiguration durch die integrierte CPU des Zynq-SoC vorgesehen sind, sind getrennt mit 50MHz getaktet.

Die in dieser Arbeit erstellten IP-Blöcke sind der Übersicht halber in einem untergeordneten Blockdesign integriert. Neben diesen befinden sich noch drei Xilinx-IP-Blöcke im Design. Ein VDMA-IP-Block ist dafür zuständig, das Graustufen-Originalbild, das vom Decimate-IP-Block kommt, in den Speicher zu schreiben. Ein DMA-IP-Block, der nur einfache Transaktionen ohne Scatter-Gather-Engine ausführt, schreibt die Randpixel des GradientCluster-IP-Blocks in den Speicher. Beide sind am Accelerator-Coherency-Port (ACP) angeschlossen, um sicherzustellen, dass der Cache bei neuen Daten aktualisiert wird. Die Interruptsignale beider IP-Blöcke sind herausgeführt, um die Software zu benachrichtigen, wenn Daten bereitgestellt worden sind. Außerdem gibt es noch einen AXI Interconnect, der die AXI-lite Schnittstellen der einzelnen IP-Blöcke als eine gemeinsame nach außen bereitstellt.

Für den Performancetest wurde noch ein VDMA-IP-Block und ein AXIS-Switch hinzugefügt. Der Grund dafür ist, dass der IP-Block für die Korrektur der Linsenverzeichnung nicht schnell genug ist, um die maximale Bildrate der AprilTag-Verarbeitungskette zu erreichen. Daher wird der AXIS-Switch dafür eingesetzt, dass zwischen beiden Bildquellen per Software gewechselt werden kann.

Das geforderte Timing wird mit den Standardeinstellungen für die Synthese und der Implementierung nicht erreicht. Um einen Spielraum zu bekommen, wurde die C-Synthese der HLS-IP-Blöcke mit 6ns statt den 6,67ns für 150MHz durchgeführt. Als Voreinstellung für die Synthese wurde die Strategie FlowPerfOptimized_High gewählt und für die *-directive*-Option *AlternateRoutability*. Des Weiteren ist das Retiming

eingeschaltet. Die Einstellungen für die Synthese sind in Abb. 4.4 abgebildet.

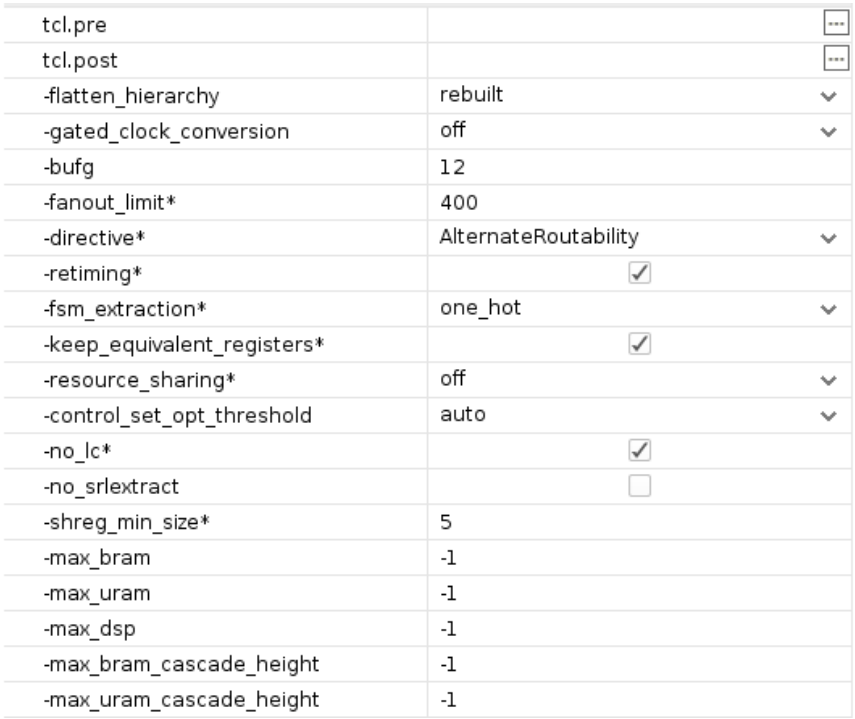

Abbildung 4.4: Einstellungen der Synthese.

Für die Implementierung ist als Voreinstellung *Performance_Retiming* ausgewählt. Zusätzlich sind die *-directive*-Optionen der einzelnen Implementierungsschritte angepasst (s. Abb. 4.5. Der opt_design-Schritt ist auf ExploreWithRemap gesetzt, um die Logik im FPGA zu reduzieren. Bei place_design sorgt die Einstellung ExtraNetDelay_high dafür, dass längere Strecken pessimistischer geschätzt werden und damit eine größere Spanne beim Routing überbleibt. Der Post-Place phys_opt-Schritt wird mit der Direktiven AlternateFlowWithRetiming ausgeführt, das dazu führt, dass Register aus einem DSP-Block oder Block-RAM heraus bzw. zwischen Logikschichten verschoben werden, um das Timing einzuhalten. Während dem Schritt route_design verhindert die Direktive NoTimingRelaxation, dass der Router das Timing lockert, um vorzeitig zu beenden. Im letzten Schritt mit dem Namen Post-Route phys_opt wird erneut das Retiming durch die Direktive AddRetime angewendet.

Aus Abb. 4.6 kann abgelesen werden, dass das System mehr als die Hälfte der vorhandenen LUT-Ressourcen verwendet und 41% der Flip-Flops. Der interne Speicher des FPGA (BRAM) ist zu 29% belegt. Von den DSPs sind nur 14% in Benutzung. Daher könnten noch andere Subsysteme im FPGA untergebracht werden. Von allen belegten Ressourcen sind etwas weniger als die Hälfte dem AprilTag-Algrithmus zugeordnet (s. Abb. 4.7).

Opt Design (opt_design)	
is_enabled	☑
tcl.pre	...
tcl.post	...
-verbose	☐
-directive*	ExploreWithRemap
More Options	
Power Opt Design (power_opt_design)	
Place Design (place_design)	
tcl.pre	...
tcl.post	...
-directive*	ExtraNetDelay_high
More Options	
Post-Place Power Opt Design (power_opt_design)	
Post-Place Phys Opt Design (phys_opt_design)	
is_enabled*	☑
tcl.pre	...
tcl.post	...
-directive*	AlternateFlowWithRetiming
More Options	

Route Design (route_design)	
tcl.pre	...
tcl.post	...
-directive*	NoTimingRelaxation
More Options	
Post-Route Phys Opt Design (phys_opt_design)	
is_enabled*	☑
tcl.pre	...
tcl.post	...
-directive*	AddRetime
More Options	

Abbildung 4.5: Einstellungen der Implementierung.

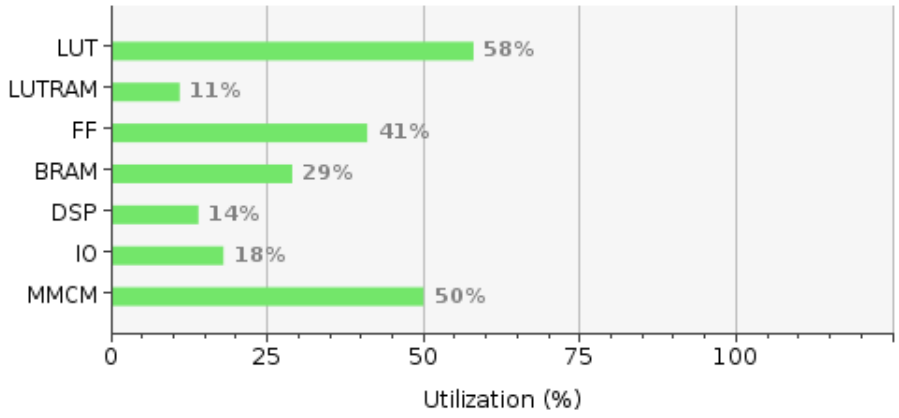

Abbildung 4.6: Ressourcennutzung des gesamten Systems (Zynq 7020).

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	LUT Flip Flop Pairs (53200)	Block RAM Tile (140)	DSPs (220)
system_i (system)	30985	43399	1122	272	12380	28998	1987	16584	40	30
AprilTag (AprilTag_imp_HWU...	13874	20128	469	118	5849	13133	741	8112	23.5	29
axi_dma_cluster (system_...	1188	2053	1	0	549	1107	81	831	3	0
axi_interconnect_0 (syste...	663	628	0	0	295	602	61	272	0	0
axi_vdma_orig (system_ax...	1456	2040	1	0	651	1356	100	866	1.5	0
axis_dwidth_converter_0 ...	325	1113	132	0	313	325	0	97	0	0
blur_sharpen (system_bl...	1803	2427	0	0	727	1583	220	1076	3.5	25
ccl_pass_one (system_ccl...	3243	4338	138	46	1179	3186	57	2220	2.5	1
ccl_pass_two (system_ccl...	1958	3119	192	72	923	1857	101	1015	4	1
decimate (system_decim...	347	472	5	0	174	347	0	201	0	2
gradient_cluster (system...	753	999	0	0	333	753	0	326	1.5	0
threshold (system_thres...	2140	2939	0	0	863	2019	121	1196	7.5	0

Abbildung 4.7: Ressourcennutzung der einzelnen IP-Blöcke des AprilTag-Verfahren in Bezug zum gesamten System.

Das führt zum Power-Report des gesamten Systems, das in Abb. 4.8 abgebildet ist. Es wird eine Leistungsaufnahme von etwas über zwei Watt geschätzt. Davon bezieht sich der Großteil auf die CPU des Chips (PS7: 66%).

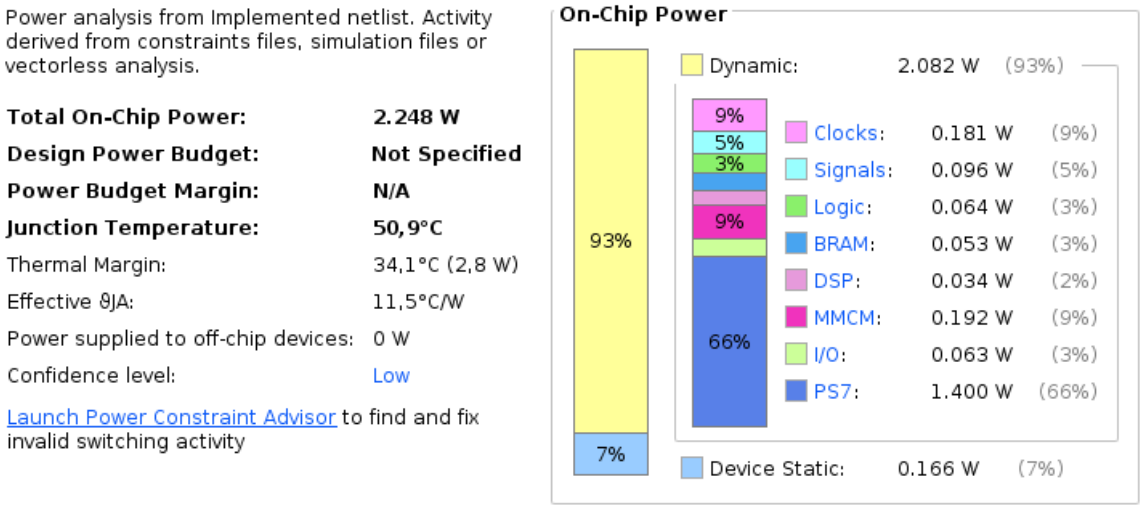

Abbildung 4.8: Power-Report des gesamten Systems.

5 Ansteuerungsbibliothek

5.1 Entwurf

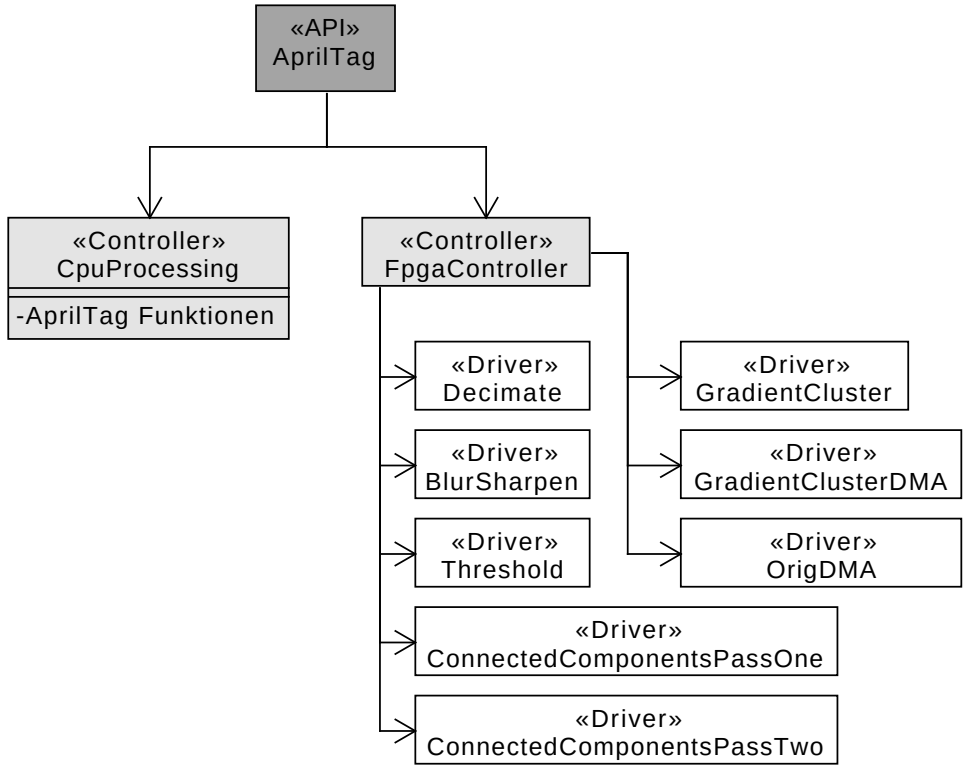

Abbildung 5.1: Klassendiagramm der Ansteuerungsbibliothek.

Die Architektur der Ansteuerungsbibliothek ist in drei Schichten unterteilt. An der Spitze ist die API, die dem Entwickler zur Verfügung steht, um die AprilTag-Verarbeitung zu verwenden. Darunter liegt die Controller-Schicht, die dafür zuständig ist, die Daten an die richtigen Blöcke und Funktionen zu verteilen. Ganz unten befinden sich die Treiber für die IP-Blöcke und die Funktionen des AprilTag-Algorithmus, die auf der CPU ausgeführt werden.

Die Treiberklassen erhalten jeweils eine „configure“-Funktion, die die Parameter der Blöcke, wie z.B. Breite und Höhe des Bildes, festlegt. Darüber hinaus werden noch die zwei Methoden „start“ und „stop“ benötigt. Die Klasse OrigDMA und GradientClusterDMA erhalten zusätzliche Methoden, um den letzten Buffer, in dem geschrieben wurde, anzugeben.

Die Klasse FpgaController fasst die Steuerung der IP-Blöcke zusammen und erhält Methoden zum Konfigurieren, Starten, Stoppen, sowie der Angabe des Buffers. Die CpuProcessing-Klasse benötigt nur eine Funktion, die die übrigen Schritte auf der CPU ausführt.

In der übergeordneten Klasse AprilTag befinden sich wieder die Konfigurations-, Start- und Stopp-Methode. Zusätzlich können alle Parameter des AprilTag-Algorithmus einzeln gesetzt werden. Damit der CPU-Teil ausgeführt werden kann, gibt es die Funktion „detect“, die ständig aufgerufen werden muss, um die Informationen über die erkannten Tags zurückzugeben.

5.2 Implementierung

Da das HLS-Tool automatisch Treiberfunktionen für die IP-Blöcke generiert, werden lediglich die Aufrufe und eine Treiberinstanz in den Treiberklassen gekapselt. Für jeden Parameter, der über die AXIlite-Schnittstelle erreichbar ist, gibt es set- und get-Funktionen, die die Daten an die richtige Speicherstelle schreiben bzw. davon lesen. Die Basisadresse ist in der Treiberinstanz gespeichert, die durch eine einfache Struktur abgebildet wird. Im Regelfall ist neben der Basisadresse nur noch ein Flag zu finden, das angibt, ob die Instanz initialisiert worden ist. Eine statische Liste mit allen Instanzen eines IP-Blocks enthält alle Konfigurationseinstellungen (Basisadresse, etc.) aus dem Vivado Block-Design. Mithilfe einer Device-ID kann die zugehörige Treiberinstanz initialisiert werden. Beim Starten wird jeweils das Start- und Autorestart-Bit gesetzt. Die HLS IP-Blöcke laufen immer bis zum Ende des Bildes, nachdem sie gestartet wurden. Falls das Autorestart-Bit gesetzt ist, fangen diese danach mit dem nächsten Bild an. Daher wird in der stop-Funktion das Autorestart-Bit gelöscht.

In der BlurSharpen-Klasse wird zusätzlich der Kernel für die Weichzeichnung anhand der Formel 4.3 berechnet. Diese unterscheidet sich von der offiziellen Version dadurch, dass direkt ein zweidimensionaler Kernel erstellt wird, anstatt eines eindimensionalen, der horizontal und vertikal angewendet wird.

Da das Gradient-Cluster-DMA-IP Interrupts auslösen kann, ist in der entsprechenden Treiberklasse zusätzliche Logik, die diese behandelt. Ein Transfer besteht aus allen Kantenpixeln eines Bildes. Am Ende wird ein Interrupt ausgelöst. Die Software reagiert darauf und startet, sofern nicht zuvor die stop-Funktion aufgerufen wurde, einen neuen Transfer für das nächste Bild. Um Ressourcen zu sparen, wurde der IP-Block ohne Scatter-Gather konfiguriert. Daher muss die Software den nächsten Transfer initiieren. Dabei stehen drei Buffer zur Verfügung, die zyklisch verwendet werden und so ein gleichzeitiges Lesen der CPU und Schreiben des FPGA eines einzelnen Buffers verhindern.

Das FPGA schreibt an zwei unterschiedlichen Stellen der Verarbeitungskette Daten in den Speicher, die von der CPU weiterverarbeitet werden. Das Originalbild wird bereits nach dem ersten IP-Block geschrieben. Die Grenzpixel für den GradientCluster-

Schritt stehen erst am Ende zur Verfügung, nachdem die Daten durch die CCL-IP-Blöcke, die das Bild in den Speicher schreiben und wieder auslesen, verzögert wurden. Daher muss es mindestens drei Buffer geben: Einer wird durch die CPU verarbeitet, für den zweiten werden zur gleichen Zeit die Grenzpixel geschrieben und der dritte empfängt bereits das nächste Bild. Um beide Datenquellen zu synchronisieren, nutzt der GradientClustering-Schritt die gleiche Anzahl an Buffern. Dadurch ist sichergestellt, dass die Buffernummer des einen IP-Blocks auch auf den richtigen Buffer des anderen verweist. Aufgrund des festen Verarbeitungsverlaufs durch Handshakes kann keiner der beiden den anderen überholen.

Die FpgaController-Klasse verwaltet Instanzen der Treiberklassen. Bei der Konfiguration werden unter anderem die Höhe und die Breite des Bildes gesetzt. Da der Decimate-Schritt das Bild verkleinert, muss der FpgaController die passende Größe an die nachfolgenden IP-Blöcke übergeben. Die neue Größe des Bildes errechnet sich aus der Originalgröße (w, h) und dem Decimate-Faktor f:

$$w_d = 1 + \frac{w_o - 1}{f}, \quad h_d = 1 + \frac{h_o - 1}{f} \tag{5.1}$$

Beim Anhalten der Verarbeitung werden die IP-Blöcke von vorne nach hinten gestoppt. Diese laufen weiter, bis sie das Ende des Bildes erreicht haben. Daher muss erst auf Beendigung des Decimate-IP gewartet werden, bevor das Orig-DMA angehalten wird. Ansonsten kann das Decimate-IP keine Daten mehr senden und hält daher nie an.

Die Bearbeitungsschritte, die auf der CPU ausgeführt werden sollen, wurden aus der offiziellen Version übernommen. Nicht inbegriffen ist der zweite Bearbeitungsschritt des GradientCluster-Algorithmus, der die Daten des FPGAs in das erforderliche Format der offiziellen Version konvertiert. Außerdem ist die der fit quads-Schritt so angepasst, dass anstelle des Bildes, das nach dem zweiten Schritt verkleinert und/oder weichgezeichnet bzw. geschärft ist, das Graustufenoriginalbild verwendet wird.

6 Auswertung

6.1 Aufbau des Performancetests

Im Vorfeld wurden bereits Testbilder erstellt. Auf diesen ist jeweils ein Würfel mit AprilTags zu sehen. Der Würfel ist zwischen zehn und zweihundert Metern von der Kamera entfernt. Die Größe der Tags beträgt genau einen Meter. Auf vierzig der achtzig Bilder ist der Würfel mit einer Fläche parallel zur Bildfläche, sodass nur ein einziges Tag sichtbar ist. In den übrigen Bildern ist er um 30° um die y-Achse des Bildes gedreht. Zusätzlich ist bei einer Hälfte der Bilder noch ein Schachbrettboden eingefügt, um die Kantenzahl zu erhöhen. Abstand, Drehung und Boden sind in allen Kombinationen als Testbild vorhanden.

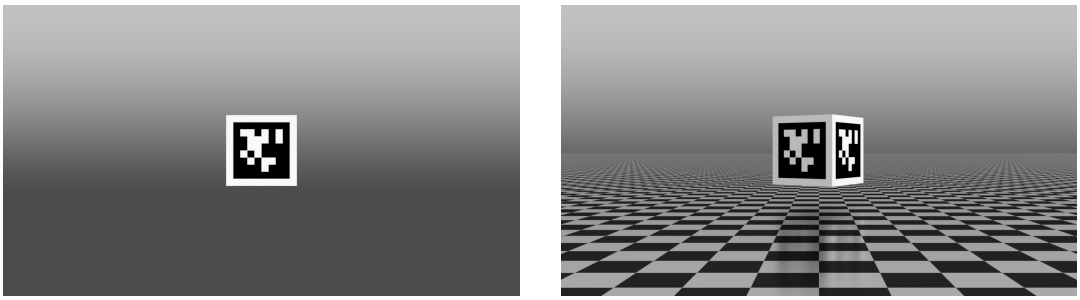

Abbildung 6.1: Beispiele der Testbilder.

Die Bilder wurden künstlich mit dem 3D-Rendering Programm Blender erstellt. Der Vorteil daran ist, dass die Kameraparameter bekannt sind, da sie selbst eingegeben wurden. Dasselbe gilt auch für die Position und Drehung des Würfels im Bild.

Zur Durchführung des Performancetests werden die Bilder von einer SD-Karte geladen und einzeln in einem Framebuffer abgelegt. Die Kamera und die Videoausgabe werden nicht benutzt. Der AxisSwitch wird so konfiguriert, dass dieser die Bilder des Video-DMA an den ersten AprilTag-IP-Block weiterleitet. Dann wird die Verarbeitungskette im FPGA gestartet und zehnmal die detect-Methode der AprilTag-Klasse aufgerufen, um die CPU-Verarbeitung zu starten. Es wird das Intervall zwischen zwei Aufrufen gemessen, aus der die Bildrate hervorgeht. Nach der Verarbeitung wird das Ergebnis ausgewertet. Anhand der Bildnummer ist der Inhalt bekannt und das Ergebnis kann überprüft werden. Aus der Rotationsmatrix werden die Euler-Winkel berechnet, um die Abweichung in Bogenmaß bzw. in Grad anzugeben. Sowohl Position als auch Drehung liegen dann als 3D-Vektor vor. Für jedes Bild gibt es Referenzvektoren. Die Abweichung für die spätere Auswertung der Ergebnisse wird durch den euklidischen Abstand zwischen dem berechneten Vektor und der Referenz angegeben.

Im Unterschied zur FPGA-Implementierung, die nur RGB-Bilder verarbeitet, nimmt

die offizielle Version nur Graustufenbilder an. Deshalb wird vor dem Start das Testbild mit derselben Formel konvertiert, die auch im FPGA genutzt wird (s. Gleichung 4.2). Dadurch ist sichergestellt, dass sich die zu verarbeitenden Daten nicht unterscheiden.

6.2 Ergebnisse

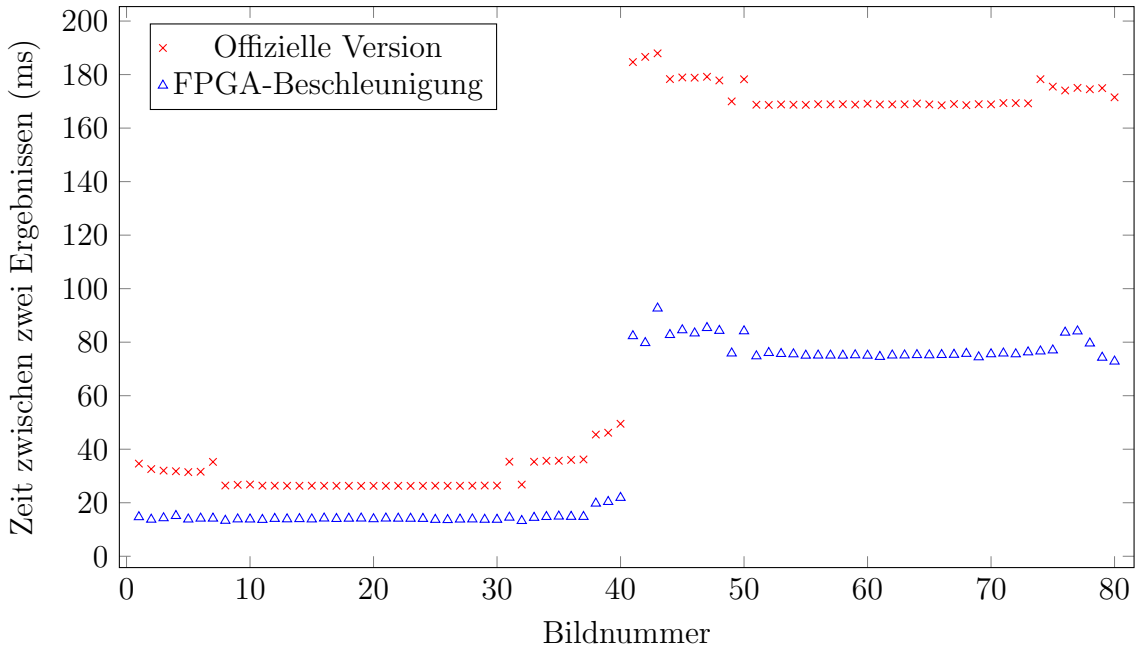

Abbildung 6.2: Gemessene Zeit zwischen zwei Ergebnissen. Durchschnitt aus jeweils zehn Messungen pro Bild.

Die Ergebnisse des Performancetests zeigen, wie in Abb. 6.2 ersichtlich, dass die FPGA-Implementierung ein im Schnitt halb so großes Zeitintervall zwischen den Bildern benötigt. Ab Bild 41 ist das Schachbrettmuster auf dem Boden abgebildet. Das Intervall steigt gegenüber den Bildern ohne Boden auf ein Sechsfaches. Beide Implementierung sind gleichermaßen betroffen. Durch die deutlich höhere Anzahl an Kanten bricht die Bildrate von 71 BPS (FPGA) bzw. 37 BPS (CPU) auf 13 BPS (FPGA) bzw. 6 BPS (CPU) ein.

Bei der FPGA-Implementierung und den Bildern ohne Boden muss die Software auf den FPGA warten. Die Softwarezeit beträgt nur wenige Millisekunden und zusammengerechnet mit der Wartezeit liegt sie größtenteils um die 14ms bzw. 71 Bilder pro Sekunde. Bei den Bildern mit Boden dauert der Softwareteil so lange, dass der FPGA vor Ende der Verarbeitung bereits das nächste Bild verarbeitet hat und die Bildrate nur vom Aufwand der Software abhängig ist.

		Testbild 1		Testbild 41	
Nr.	**Schritt**	**Zeit**	**Akkumuliert**	**Zeit**	**Akkumuliert**
0	init	0.00 ms	0.00 ms	0.00 ms	0.00 ms
1	FPGA result	7.85 ms	7.85 ms	0.00 ms	0.00 ms
2	gradient_cluster_cpu	1.64 ms	9.50 ms	23.26 ms	23.26 ms
3	fit quads	1.48 ms	10.99 ms	36.42 ms	59.68 ms
4	decode+refinement	0.68 ms	11.67 ms	5.65 ms	65.34 ms
5	reconcile	0.00 ms	11.67 ms	0.01 ms	65.36 ms
6	debug output	0.00 ms	11.67 ms	0.00 ms	65.36 ms
7	cleanup	0.00 ms	11.67 ms	0.01 ms	65.37 ms
8	pose estimation	4.25 ms	15.93 ms	16.83 ms	82.21 ms

Tabelle 6.1: Verarbeitungszeit der FPGA-Implementierung. (decimate=2, sigma=0, minDiff=20)

		Testbild 1		Testbild 41	
Nr.	**Schritt**	**Zeit**	**Akkumuliert**	**Zeit**	**Akkumuliert**
0	init	0.00 ms	0.00 ms	0.00 ms	0.00 ms
1	decimate	4.95 ms	4.95 ms	5.00 ms	5.00 ms
2	blur/sharp	0.00 ms	4.95 ms	0.00 ms	5.01 ms
3	threshold	6.27 ms	11.23 ms	7.39 ms	12.40 ms
4	unionfind	9.41 ms	20.65 ms	15.91 ms	28.31 ms
5	make clusters	7.37 ms	28.02 ms	33.80 ms	62.12 ms
6	fit quads	1.42 ms	29.45 ms	99.85 ms	161.97 ms
8	decode+refinement	0.70 ms	30.17 ms	4.99 ms	167.08 ms
9	reconcile	0.00 ms	30.17 ms	0.01 ms	167.09 ms
10	debug output	0.00 ms	30.17 ms	0.00 ms	167.09 ms
11	cleanup	0.00 ms	30.17 ms	0.01 ms	167.11 ms
12	pose estimation	4.29 ms	34.46 ms	18.01 ms	185.12 ms

Tabelle 6.2: Verarbeitungszeit der offiziellen Version. (decimate=2, sigma=0, minDiff=20)

Zwischen der offiziellen Version und der FPGA-Implementierung gibt es eine sehr starke Abweichung der Zeit beim „fit quads“-Schritt (Tabelle 6.1 Nr. 3, Tabelle 6.2 Nr. 6). Die offizielle Version benötigt fast drei mal so lange für die Berechnung. Beide Versionen nutzen allerdings dieselbe Funktion. Das deutet darauf hin, dass die Cluster aus dem vorherigen Schritt verschieden sind. Eine erste Untersuchung zeigt, dass der FPGA, anders als erwartet, deutlich mehr Kantenpixel ausgibt als die entsprechenden Softwarefunktionen. Außerdem benutzt der betroffene Schritt das Graustufenoriginalbild und nicht das vorverarbeitete Bild.

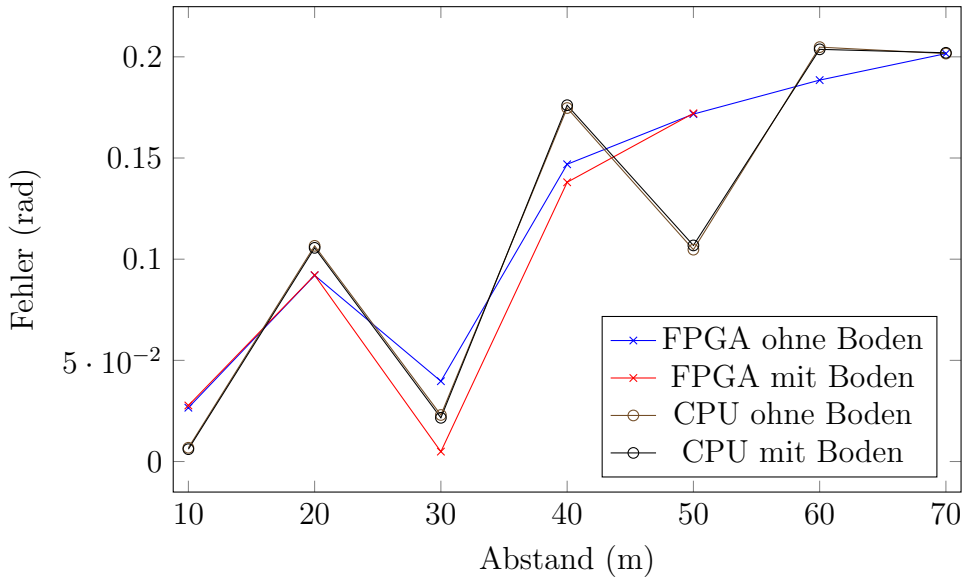

Abbildung 6.3: Abweichung der Drehung ohne Drehung des Tags.

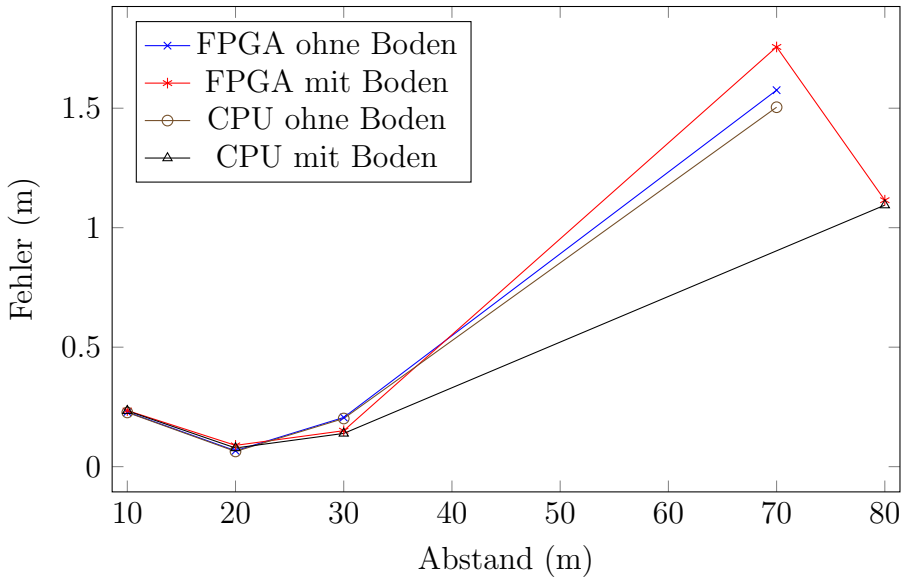

Abbildung 6.4: Abweichung des Abstands bei 60°.

Das macht sich auch bei der Genauigkeit bemerkbar. Wie in Abb. 6.3 zu sehen ist, weicht diese zwischen FPGA- und Software-Implementierung ab. Im Fall „FPGA mit Boden“ wird das Tag nur bis 50m Entfernung erkannt, während in den anderen Testreihen das Tag bis 70m erkannt wird. Dazu muss auch berücksichtigt werden, dass auch die offizielle Version die Tags zum Teil nicht erkennen kann. Im Allgemeinen und

insbesondere bei kurzen Abständen liegen die Messwerte dicht beieinander (s. weitere Abb. im Anhang). Es kann keine Aussage getroffen werden, dass eine Implementierung genauerer ist, als die andere. Beide haben an verschiedenen Stellen ihre Vorteile.

Eine weitere Besonderheit ist, dass bei bestimmten Abständen die gedrehten April-Tags nicht erkannt werden. Bei 30°sind es die Bilder mit 70m bis 90m, die teilweise nicht erkannt werden. Das Tag in 100m Entfernung wird jedoch immer gefunden. Bei 60°-Drehung der Tags werden diese mit Abständen ab 40m nicht immer erkannt. Der Bereich von 40m bis 60m wird überhaupt nicht gefunden. AprilTags mit mehr als 100m Entfernung werden nie erkannt.

7 Zusammenfassung und Ausblick

7.1 Mögliche weitere Schritte

In dieser Arbeit wurde die Multitaskingfähigkeit der CPU außen vor gelassen. Demnach ist es denkbar, dass das hier erstellte System die Leistungsfähigkeit des Zynq-SoC noch nicht ganz ausreizt. Die offizielle Version unterstützt bereits Multithreading durch POSIX-Threads, das aber bei der Portierung fallen gelassen wurde, da bei der Standalone-Anwendung kein Betriebssystem eingesetzt wird, das diese Funktion bereitstellt. Durch die Nutzung des zweiten Kerns der CPU könnte eine weitere Leistungssteigerung erzielt werden.

7.2 Zusammenfassung

Das Ziel der Arbeit war es, den AprilTag-Algorithmus zu beschleunigen und dadurch eine höhere Bildrate zu erreichen, um diese für Regelungsaufgaben von Raumfahrtsystemen zu benutzen. Tatsächlich konnte gezeigt werden, dass sich der AprilTag-Algorithmus durch geschickte Aufteilung auf FPGA und CPU auf einem Zynq-SoC beschleunigen lässt und dadurch eine doppelte Bildrate erreicht werden kann.

Dafür wurden aus der Systemarchitektur die IP-Blöcke mit dem HLS-Tool von Xilinx in C++ realisiert und herausgearbeitet, welchen Aufbau und welche Optimierungsdirektiven für hohe Bildraten nötig sind. Außerdem wurde eine Ansteuerungsbibliothek entwickelt, die die Komplexität der Konfiguration und der Steuerung der IP-Blöcke, sowie die übrigen Schritte in Software hinter einer leicht zu benutzbaren API verbirgt, sodass das System in einem übergeordneten System eingesetzt werden kann.

Für den abschließenden Performancetest wurden künstliche Testbilder mit einem 3D-Programm erstellt, um präzise Referenzdaten für die Auswertung zu erhalten. Das Ergebnis war eine Verdopplung der Bildrate, sodass diese Version des Algorithmus für Regelungsaufgaben geeignet ist. Kritisch zu betrachten sind allerdings die z.T. verschiedenen Genauigkeiten bei einigen Bildern und auch eine konkrete Aussage darüber, welche Version des Algorithmus besser ist, kann nicht getroffen werden, da beide bei verschiedenen Bildern Aussetzer hatten.

8 Literatur

[1] *AprilTag 3 GitHub Repository*. 2018. URL: https://github.com/AprilRobotics/apriltags (besucht am 15. 03. 2019).

[2] *AXI4-Stream Video IP and System Design Guide*. Version 2.2. Xilinx, Inc. 10. Dez. 2018. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_videoip/v1_0/ug934_axi_videoIP.pdf (besucht am 22. 07. 2019).

[3] *Das DLR in Zahlen und Fakten*. Sep. 2017. URL: https://www.dlr.de/dlr/de/Portaldata/1/Resources/documents/2019/DLR_FactsFigures_2017_DE_web.pdf (besucht am 13. 03. 2019).

[4] Marcel Flottmann. "Entwurf einer Systemarchitektur und Implementierung des Grundsystems für eine FPGA basierte Bildverarbeitung." Projektbericht. Hochschule Osnabrück, Fakultät Ingenieurwissenschaften und Informatik, 22. Mai 2019.

[5] R. M. Haralick. "Some Neighborhood Operators". In: *Real-Time Parallel Computing: Imaging Analysis*. Hrsg. von Morio Onoe, Kendall Preston und Azriel Rosenfeld. Boston, MA: Springer US, 1981, S. 11–35. ISBN: 978-1-4684-3893-2. DOI: 10.1007/978-1-4684-3893-2_2. URL: https://doi.org/10.1007/978-1-4684-3893-2_2.

[6] *ITU-R BT.1700 Characteristics of composite video signals for conventional analogue television systems*. 6. Juli 2007. URL: https://www.itu.int/rec/R-REC-BT.1700-0-200502-I/en (besucht am 23. 07. 2019).

[7] Maximilian Krogius, Acshi Haggenmiller und Edwin Olson. "Flexible Layouts for Fiducial Tags". In: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2019.

[8] Ramana V. Rachakonda, Peter M. Athanas und A. Lynn Abbott. "High-speed region detection and labeling using an FPGA-based custom computing platform". In: *Field-Programmable Logic and Applications*. Hrsg. von Will Moore und Wayne Luk. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, S. 86–93. ISBN: 978-3-540-44786-3.

[9] Kurt Schwenk und Felix Huber. "Connected Component Labeling algorithm for very complex and high-resolution images on an FPGA platform". In: *SPIE Remote Sensing. International Society for Optics and Photonics, 2015*. Bd. 9646. SPIE Proceedings. SPIE, Sep. 2015. URL: https://elib.dlr.de/100363/.

[10] Carl Johann Treudler u. a. "ScOSA - Scalable On-Board Computing for Space Avionics". In: *IAC 2018*. Okt. 2018. URL: https://elib.dlr.de/122492/.

[11] *Vivado Design Suite User Guide: High-Level Synthesis*. Version 2017.4. Xilinx, Inc. 2. Feb. 2018. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf (besucht am 22. 07. 2019).

[12] John Wang und Edwin Olson. "AprilTag 2: Efficient and robust fiducial detection". In: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Okt. 2016.

[13] Raymond Zhang. "FPGA SoC Fiducial System for Unmanned Aerial Vehicles". Diss. ROCHESTER INSTITUTE OF TECHNOLOGY, 2018.

A Anhang

A.1 Inhalt der CD

Bachelorarbeit/Bachelorarbeit.pdf Bachelorarbeit in PDF-Form

Bachelorarbeit/Bachelorarbeit.tex Bachelorarbeit in Latex-Form

Bachelorarbeit/Glossar.tex Glossar der Bachelorarbeit in Latex-Form

Bachelorarbeit/Bachelorarbeit.bib Verwendete Literatur der Bachelorarbeit in BibTex-Form

Blockdesign/system.pdf Gesamtsystem als Blockdesign aus Vivado

Blockdesign/AprilTag.pdf AprilTag Subsystem als Blockdesign aus Vivado

Ergebnisse/Auswertung.txt Ausgabe des Performancetests, aus der die Diagramme abgeleitet sind.

Referenzen/* Für die Bachelorarbeit genutzte Literatur und Quellen. Die Nummer am Anfang des Namens entspricht der Nummer im Bericht.

Testbilder/* Die im Performancetest verwendeten Bilder

A.2 Systemarchitektur

A.2.1 Spezifikation

Beschreibung der Blöcke im Blockdiagramm aus [4]:

- **Undistort (FPGA)** (aus Vorprojekt)

  Eingabe: AXI-Master mit Zugriff auf Framebuffer

  Verarbeitung: Korrigiert die Verzeichnung durch die Linse. Das Eingabebild wird anhand der Kameraparameter und Verzeichnungskoeffizienten korrigiert. Dazu werden die Funktionen cv::InitUndistortRectifyMap und cv::Remap von OpenCV bzw. die HLS Äquivalente genutzt.

  Ausgabe: AXI-Video-Stream mit 8-Bit RGB Bild

- **Decimate (FPGA)**

  Eingabe: AXI-Video-Stream mit 8-Bit RGB Bild

  Verarbeitung: Wandelt das Kamerabild in Grauwerte um und verkleinert bei Bedarf das Bild. Die Verkleinerung erfolgt durch Weglassen von Zeilen und Spalten. Es wird daher jedes zweite (dritte,...) Pixel in einer Zeile nicht ausgegeben, sowie jede zweite (dritte,...) ganze Zeile. Parallel dazu muss das Graustufenbild in Originalgröße aber weiterhin durch den zweiten Stream in den Speicher geschrieben werden.

  Ausgabe: 2 AXI-Video-Stream mit 8-Bit Graustufenbilder in Originalgröße und verkleinert

- **Blur/Sharpen (FPGA)**

  Eingabe: AXI-Video-Stream mit 8-Bit Graustufenbild

  Verarbeitung: Zeichnet das Bild weich oder schärft das Bild. Es gibt die Kernelgrößen 1 (= Keine Änderung), 3 und 5. Dies bestimmt die Anzahl der Zeilen, die im Zeilenpuffer gespeichert werden. Das Bild wird mit dem Gauß-Kernel gefaltet. Die Randbehandlung erfolgt durch Fortsetzung der Randpixel. Beim Schärfen wird das weichgezeichnete Bild vom Doppelten des Originals abgezogen.

  Ausgabe: AXI-Video-Stream mit 8-Bit Graustufenbild

- **Threshold (FPGA)**

  Eingabe: AXI-Video-Stream mit 8-Bit Graustufenbild

  Verarbeitung: Unterteilt das Bild in schwarze, weiße und graue Bereiche, die durch 2-Bit Werte repräsentiert werden. Dabei steht 01 für schwarz, 10 für weiß und 00 für grau.

1. In 4x4 Blöcke aufteilen und minimale und maximale Werte als neues Min-/Max-Bild erstellen.
2. 3x3 Min-/Max-Faltung auf Min-/Max-Bild anwenden.
3. Mittelwert aus Min/Max für jeden einzelnen 4x4 Blöcke als Schwellwert für schwarz und weiß bzw. grau bei zu geringem Abstand zwischen Min/Max.

Ausgabe: AXI-Video-Stream mit 2-Bit Segmentwertbild

- **Connected Coponents (FPGA)**

  Eingabe: AXI-Video-Stream mit 2-Bit Segmentwertbild

  Verarbeitung: Zusammenhängende Segmente werden unter einem Label zusammengefasst.

  Ausgabe: AXI-Video-Stream mit 22-Bit Labelmaske und 2-Bit Segmentwert als Pixelwerte

- **Gradient Cluster (FPGA)**

  Eingabe: AXI-Video-Stream mit 22-Bit Labelmaske und 2-Bit Segmentwert als Pixelwerte

  Verarbeitung: Gibt die Grenzpixel zwischen schwarzen und weißen Bereichen an die Software weiter. Für jedes benachbarte schwarz-weiße Pixelpaar werden beide Labels, der Mittelpunkt des Paares und die Gradientenrichtung über eine AXI-Master Schnittstelle in den Speicher geschrieben.

  Ausgabe: Array mit 32-Bit Label 1, 32-Bit Label 2, X/Y 16-Bit Mittelpunkt (doppelter Wert, da 0.5er Schritte durch Mittelwert), X/Y 16-Bit Gradientenrichtung

- **Gradient Cluster (CPU)**

  Eingabe: Array mit 32-Bit Label 1, 32-Bit Label 2, X/Y 16-Bit Mittelpunkt, X/Y 16-Bit Gradientenrichtung

  Verarbeitung: Pixelpaare mit gleicher Label-Kombination werden zusammengefasst. Mithilfe einer Hashtabelle werden die Pixelpaare zugeordnet. Der Schlüssel ist die Verkettung der beiden Labels. Die Hashtabelle wird als Liste weitergegeben.

  Ausgabe: Liste aus Listen, die zusammengehörende Randpunkte (struct pt) enthalten.

- **Fit Quads (CPU)**

  Eingabe: Liste aus Listen, die zusammengehörende Randpunkte (struct pt) enthalten.

Verarbeitung: Für jedes Cluster aus der Liste wird geprüft, ob die Pixelpaare ein Viereck ergeben. Dazu wird versucht, diese in vier Linien aufzuteilen. Anhand der Fläche, Gradientenrichtung (schwarzes Feld umgeben von weißem Rand) und Winkelsumme wird ermittelt, ob es sich tatsächlich um ein Viereck handelt. Dann werden auch die Eckpunkte (Schnittpunkt von zwei Linien) berechnet, die weitergegeben werden.

Ausgabe: Liste mit Vierecken repräsentiert durch die Eckpunkte (struct quad)

- **Refinement (CPU)**

  Eingabe: Liste mit Vierecken repräsentiert durch die Eckpunkte (struct quad)

  Verarbeitung: Für jedes Viereck werden optional die Kanten mithilfe des Originalbildes neu berechnet. Starke Helligkeitsübergänge in der Nähe einer ursprünglichen Kante werden als Stützpunkte genommen und es wird eine neue Linie berechnet. Damit werden die Eckpunkte neu definiert.

  Ausgabe: Liste mit Vierecken repräsentiert durch die Eckpunkte (struct quad)

- **Decode (CPU)**

  Eingabe: Liste mit Vierecken repräsentiert durch die Eckpunkte (struct quad)

  Verarbeitung: Die Homographie Matrix wird berechnet und dafür genutzt, die Abtastpunkte des Tags auf das Originalbild zu projizieren. Aus Abtastpunkten mit bekannter Farbe wird ein Schwellwert für die übrigen Punkte festgelegt. Die ID des Tags wird mit der TagFamily decodiert.

  Ausgabe: Liste aus AprilTag Detektionen (struct apriltag_detection)

- **Estimate Pose (CPU)**

  Eingabe: Kameraparameter und Liste aus AprilTag Detektionen (struct apriltag_detection)

  Verarbeitung: Aus den Kameraparametern und der Homographie eines Tags wird die Lage relativ zur Kamera berechnet.

  Ausgabe: Liste aus AprilTag Detektionen mit Lage relativ zur Kamera

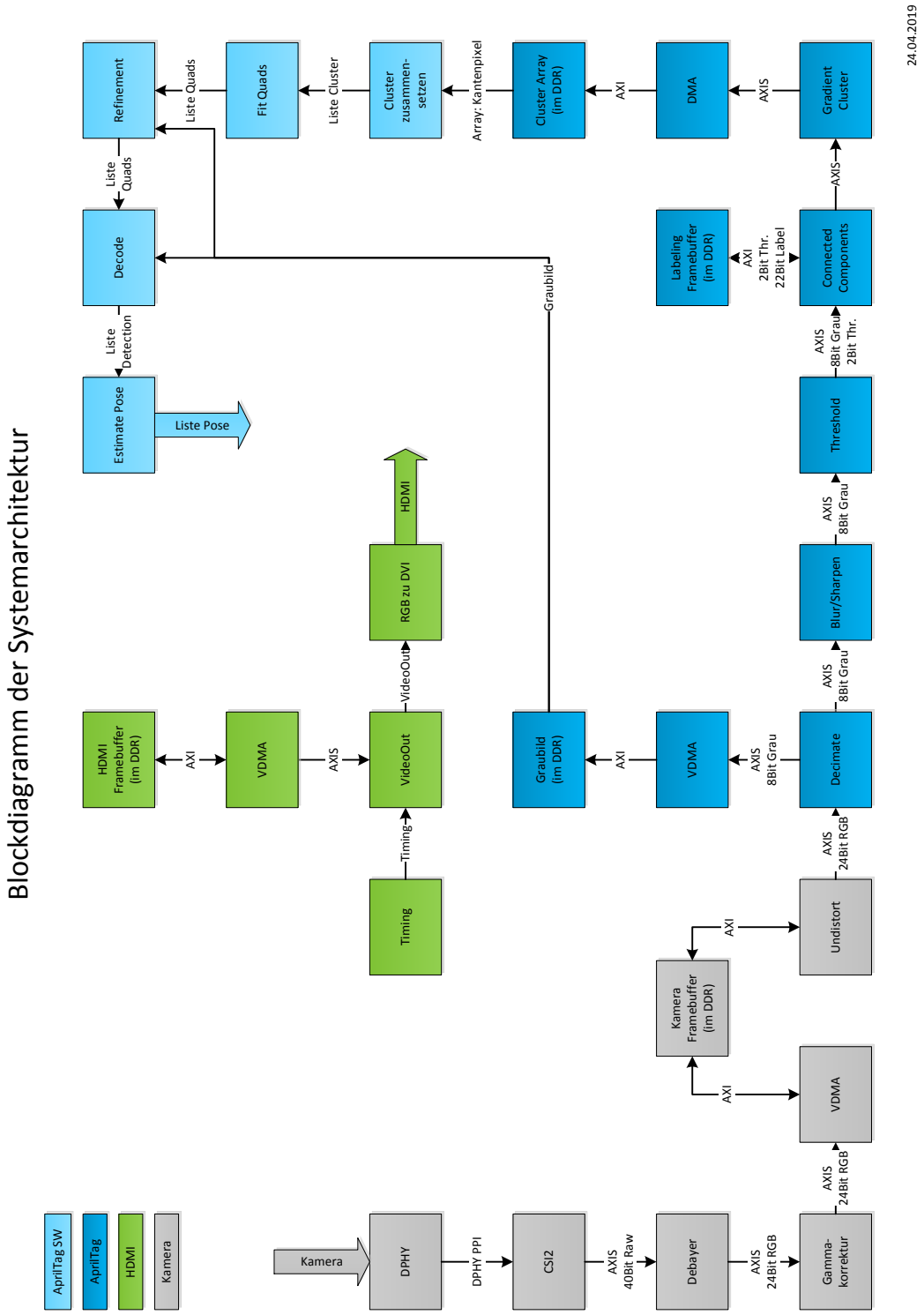

Abbildung A.1: Blockdiagramm der Systemarchitektur.

A.3 Auswertung

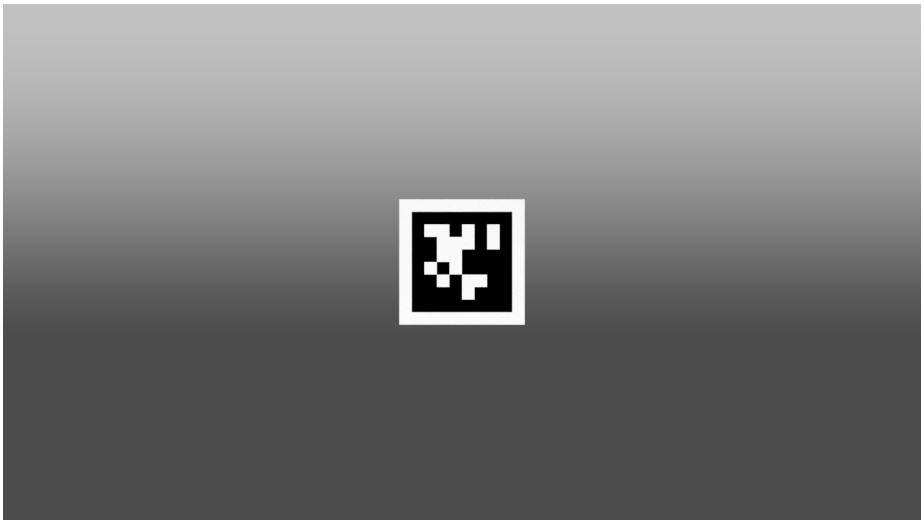

Abbildung A.2: AprilTag-Würfel nicht gedreht und ohne Boden.

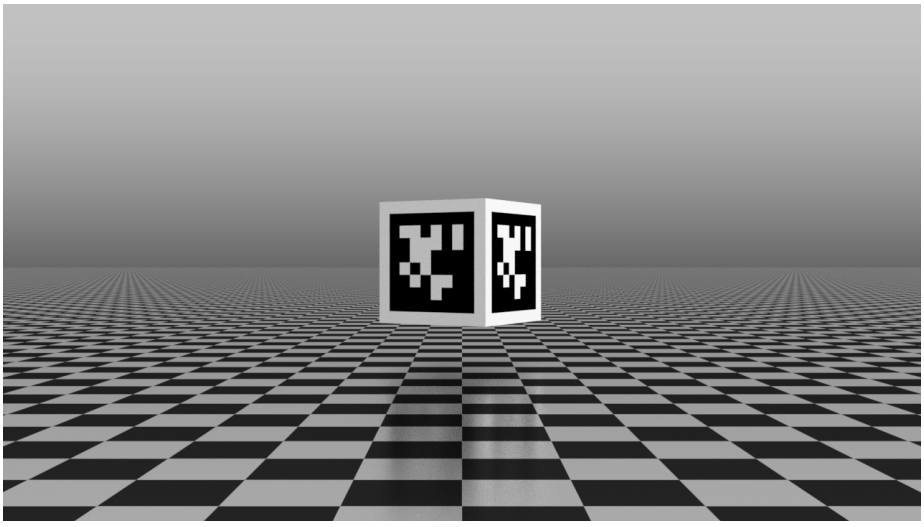

Abbildung A.3: Apriltag-Würfel gedreht und mit Boden. Das linke Tag ist um 30°gedreht, das rechte um 60°.

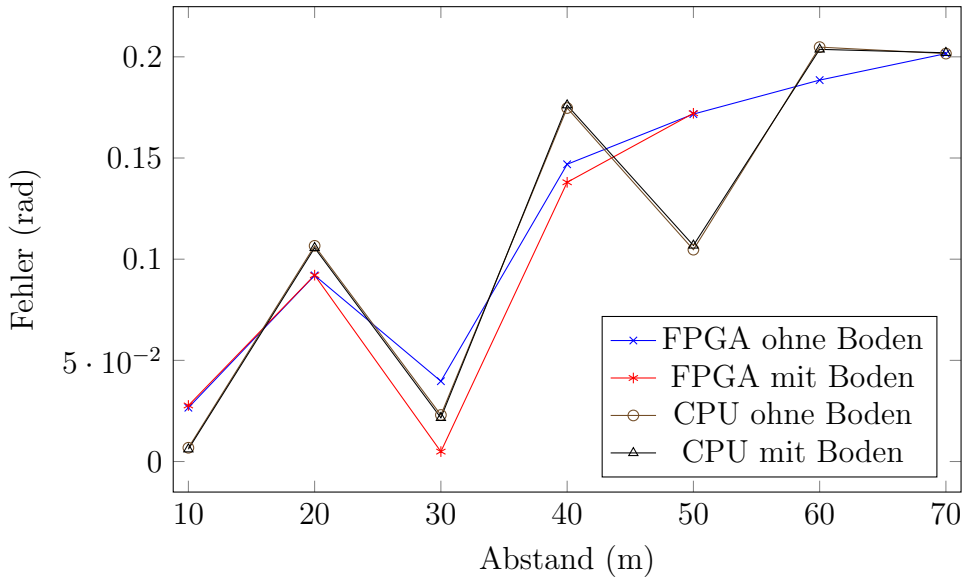

Abbildung A.4: Abweichung der Drehung ohne Drehung des Tags.

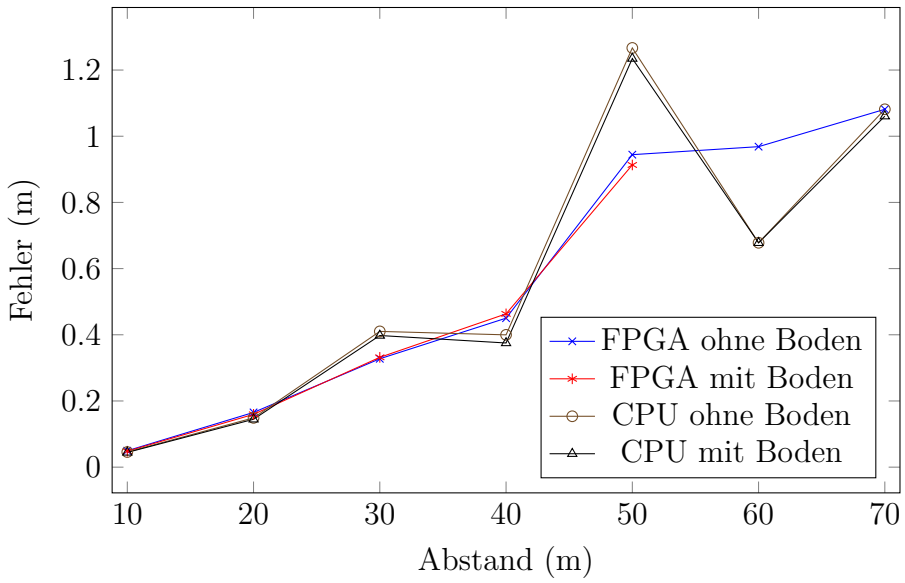

Abbildung A.5: Abweichung des Abstands ohne Drehung des Tags.

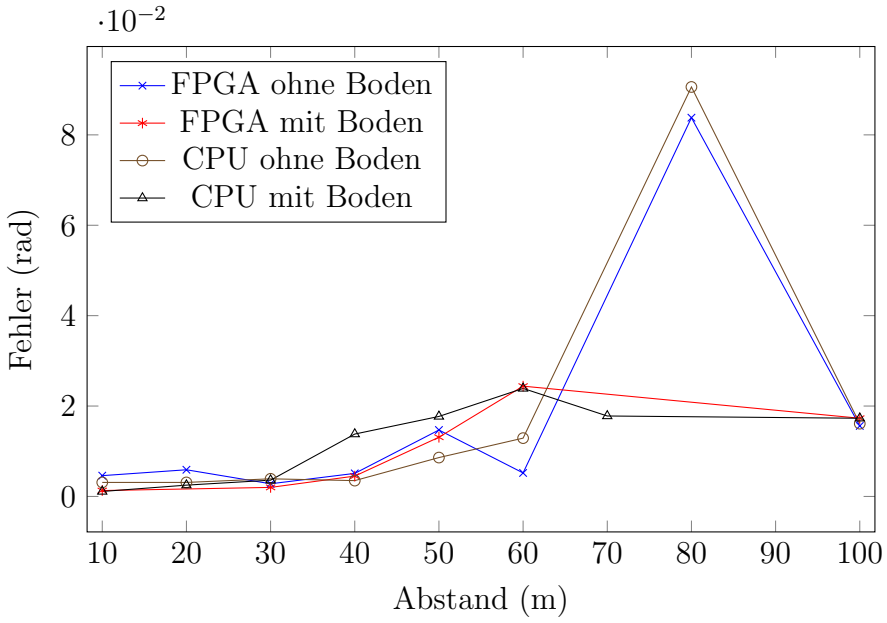

Abbildung A.6: Abweichung der Drehung bei 30°.

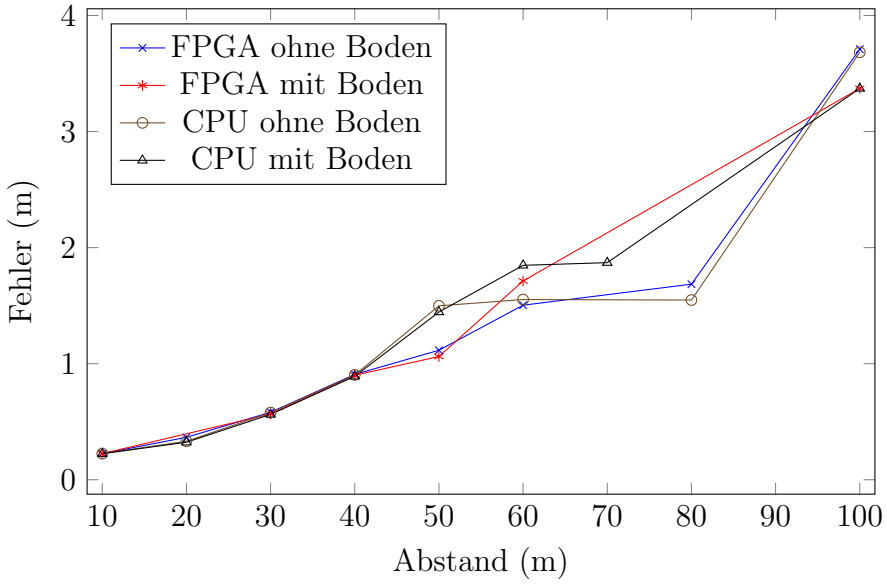

Abbildung A.7: Abweichung des Abstands bei 30°.

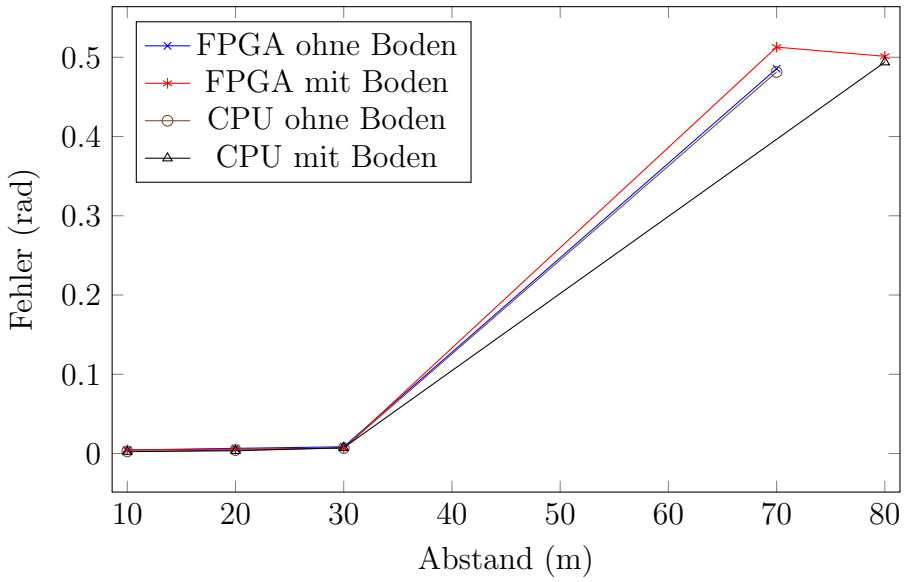

Abbildung A.8: Abweichung der Drehung bei 60°.

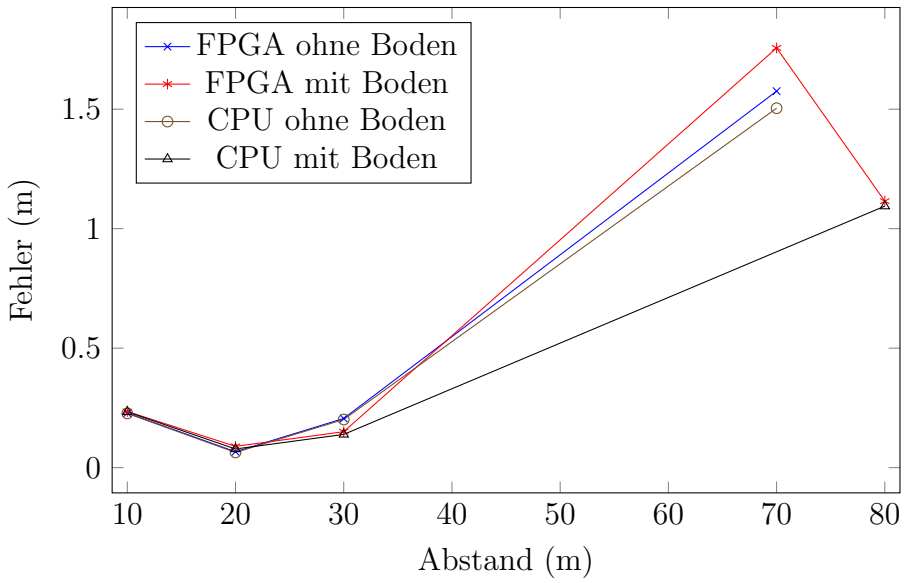

Abbildung A.9: Abweichung des Abstands bei 60°.

Eidesstattliche und Urheberrechts-Erklärung

- **Eidesstattliche Erklärung**
  (bestätigt den Ausschluss unzulässiger fremder Hilfe und die selbstständige Bearbeitung)

  Hiermit erkläre ich/ Hiermit erklären wir an Eides statt, dass ich/ wir die vorliegende Arbeit selbständig und ohne fremde Hilfe angefertigt habe/ haben. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche einzeln kenntlich gemacht. Es wurden keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

  ....
  Ort, Datum Unterschrift

  ....
  Ort, Datum Unterschrift

- **Urheberrechtliche Einwilligungserklärung**
  (ermöglicht die dauerhafte Archivierung)

  Hiermit erkläre ich/ Hiermit erklären wir, dass ich/wir damit einverstanden bin/sind, dass meine/ unsere Arbeit zum Zwecke des Plagiatsschutzes bei der Fa. Ephorus BV bis zu 5 Jahren in einer Datenbank für die Hochschule Osnabrück archiviert werden kann. Diese Einwilligung kann jederzeit widerrufen werden.

  …….. ……
  Ort, Datum Unterschrift

  Hinweis: Die urheberrechtliche Einwilligungserklärung ist freiwillig.