DIPLOMARBEIT

Hardware/Software Partitionierung auf einer System-on-Chip Plattform

Eine Untersuchung der Umsetzbarkeit von Echtzeit-Bildverarbeitung für ein Stereo-Kamera-System

ausgeführt zur Erlangung des akademischen Grades
eines Diplom-Ingenieur unter der Leitung von

Univ.Prof. Dipl.-Ing. Dr.techn. Axel Jantsch
Univ.Ass. Dipl.-Ing. Dr.techn. Martin Pongratz

am

Institut für Computertechnik (E384)
der Technischen Universität Wien

durch

Andrej Hanic BSc.
Matr.Nr. 0725734
Obere Augartenstraße 18A/6/17,
1020 Wien, Österreich

11.November 2016 _____

Kurzfassung

Das Verlangen nach individualisierten Produkten wächst ständig. Um dieses zu befriedigen, müssen die Produktionslinien deutlich flexibler aufgebaut werden. Ein vielversprechender Ansatz, um diese Flexibilität zu erreichen, könnte der „Wurftransport“-Ansatz sein, der dem menschlichen Werfen und Fangen nachempfunden ist. Dabei werden die zu transportierenden Güter zwischen den Maschinen hin und her geworfen. Damit ein sicherer und zuverlässiger Transport auf diesem Wege stattfinden kann, müssen die Maschinen das transportierte Gut selbständig erkennen und auffangen können.
Diese Aufgabe kann grob in drei Teile aufgeteilt werden: Bilderfassung mittels zwei Kameras, Bildverarbeitung zur Objekterkennung und ein Prognosesystem um die Flugbahn des Objektes vorhersagen zu können. Da diese Aufgaben rechenintensiv sind und das System echtzeitfähig sein soll (eine langsame Verarbeitung der Daten führt zu einem nicht erfolgreichen Fangvorgang), wird für die Implementierung eine System-on-Chip (SoC) Plattform, bestehend aus CPUs und einem FPGA untersucht. Diese soll den hohen Rechenaufwand bewältigen können.
In dieser Diplomarbeit wird gezeigt, welche der Aufgaben in Hardware (FPGA) und welche in Software (CPU) realisiert werden. Außerdem wird gezeigt, dass auf der verwendeten SoC Plattform alle entworfenen Verarbeitungsblöcke im FPGA schnell genug die Daten verarbeiten können und die geforderte Bildrate von 100 Bilder pro Sekunde erreichbar ist. Jedoch kann das Gesamtsystem, trotz der funktionierenden Verarbeitungsblöcke im FPGA, nicht die erwarteten Resultate erreichen, da die verwendeten Kameras mit dem Treiber die CPUs auslasten. Es wird eine sehr stark schwankende Bildrate zwischen 1 und 25 Bilder pro Sekunde erreicht, welche weit unter den erwarteten Resultaten liegt. Die Kameras benutzen eine USB 3.0 Schnittstelle, welche auf der SoC Plattform nur mittels einer USB 3.0 Erweiterungskarte für den PCI-Express Steckplatz zur Verfügung steht. In Verbindung mit den verbauten ARM CPUs, welche mit einer Frequenz von 800 MHz getaktet sind, wurde an die Grenzen der verwendeten Technologie gestoßen.
Durch eine Reduktion der von den Kameras übermittelten Daten, indem nur der relevante Bereich vom Bild übertragen wird, könnte die verwendete SoC Plattform bessere Resultate zeigen. Diese Methode ist aber derzeit, mit dem aktuell verfügbaren Kameratreiber, nicht optimal nutzbar und führt zu Fehlern bei der Aufnahme. Eine etwaige weitere Möglichkeit der besseren Funktionalität kann sich durch die Portierung, der in dieser Diplomarbeit entworfenen Architektur, auf eine neue Generation der SoC Plattform ergeben, die leistungsstärkere ARM CPUs zur Verfügung stellt.

Abstract

The demand for individualized products is constantly growing. To satisfy this, the production lines need to be built much more flexible. A promising approach to achieve this flexibility, could be a „transport-by-throwing“ approach, which is based on the human throwing and catching. The goods, which need to be transported between machines, would be thrown back and forth by these. To ensure a safe and reliable transportation this way, the machine must be able to recognize and catch the transported objects by itself.
This task can be roughly divided into three parts: image capturing using two cameras, image processing in order to identify the thrown object and a forecast system to predict the trajectory of the object. Because these tasks are associated with a high computational effort and they have to be calculated in real time (a slow calculation leads to an unsuccessful catching of the object), their implementation on a system-on-chip (SoC) platform, which consists of CPUs and a FPGA, is tested for feasibility. This platform should be able to handle the needed computational effort.
In this diploma thesis it is shown, which of the tasks are implemented in Hardware (FPGA) and which in Software (CPU). The results show, that on the used SoC platform, all designed FPGA-parts are able to process the data fast enough, in order to achieve the needed frame rate of 100 frames per second. However the whole system, despite the fact that the FPGA-parts by itself are fast enough, is not able to reach the expected results, because the cameras and the related driver load the two CPUs fully. The achieved frame rate is highly variable between 1 and 25 frames per second, which is far from the expected results. The cameras use a USB 3.0 interface, which is made available through a USB 3.0 expansion card in a PCI-Express slot. Combined with the built-in ARM CPUs, which are clocked at a frequency of 800 MHz, the limits of the used technology has been reached.
Implementing a reduction of the data, which is transmitted by the cameras, so that only the relevant part of the picture is sent, the used SoC platform should show better results. However, this approach is, with the currently available camera driver, not optimally usable and leads to errors during the capture. Another possible approach to reach better functionality, would be to port the designed architecture of this diploma thesis to the next generation of the SoC platform, which offers more powerful ARM CPUs.

Danksagung

Ich möchte mich an dieser Stelle herzlichst bei meinen Eltern bedanken, die mich über meine lange Studienzeit unterstützt haben. Die Widmung dieser Arbeit geht an meine langjährige Freundin Karolina, der ich auf diesem Weg auch ein großes Danke aussprechen möchte, vor allem für ihre unendliche Geduld und Motivation.

Weiters möchte ich mich bei meinen Studienkollegen Elvin Sebastian und Karl Osterseher, für ihre Hilfe und Ideen, bedanken.

Ein spezielles Danke geht an meinen Betreuer Martin Pongratz für seine Hilfe, Ideen und Motivation.

Inhaltsverzeichnis

Abkürzungen

AoI	Area of Interest
ASIC	Application Specific Integrated Circuit
AXI	Advanced eXtensively Interface
BRAM	Block Random Access Memory
BUFG	Global Buffer
CPU	Central Processing Unit
DMA	Direct Memory Access
DSP	Digital Signal Processor
FF	Flip-Flop
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
GT	Gigabit Transceiver
HDMI	High-Definition Multimedia Interface
HF	Hard Float
HD	High-Definition
HDL	Hardware Description Language
HLS	High Level Synthese
HP	High Performance
HW	Hardware
IO	Input/Output
IP	Intellectual Property
k-NN	k-Nearest Neighbors
LED	Light-Emitting Diode
LUT	Look-Up Table
LUTRAM	Look-Up Table RAM
MMCM	Mixed-Mode Clock Manager
PC	Personal Computer
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
PL	Programmable Logic
PLL	Phase-Locked Loop
PS	Processing System
RAM	Random Access Memory
RTL	Register-Transfer Level
SDK	Software Development Kit
SoC	System on Chip
SW	Software
UART	Universal Asynchronous Receiver/Transmitter
UDP	User Datagram Protocol

USB	Universal Serial Bus
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

1 Einleitung

Wir leben in einer Zeit, die vom Konsum geprägt ist. Viele der Produkte, mit denen wir täglich im Kontakt sind, werden durch Massenproduktion hergestellt. Als Beispiel können an dieser Stelle Lebensmittel, Möbel, Elektronik, Automobile und viele mehr genannt werden. Ohne Massenproduktion würde die Zeit, die für die Herstellung dieser Produkte und somit auch der Preis deutlich höher sein und nur wenige Leute könnten sich diese Produkte leisten. Die Massenproduktion ermöglicht also den Massenkonsum von Produkten zu stillen. Einer der größten Vertreter der Massenproduktion ist die bereits erwähnte Automobilindustrie. Vom ersten Automobil im Jahre 1885 bis zum Einsatz des Fließbandes in der Autoproduktion im Jahre 1913 durch Henry Ford [31], wurde tatsächlich jedes Fahrzeug komplett zusammengebaut und erst danach wurde mit der Arbeit an dem nächsten Fahrzeug begonnen. Bei der Größe des heutigen Automobilmarktes kann man sich so etwas nicht mehr vorstellen. Durch die Einführung des Fließbandes in die Produktion konnte damals die Zeit für die Herstellung eines Automobils von 12 Stunden auf nur 93 Minuten gesenkt werden [31]. Die Produktion wurde deutlich effizienter und dadurch wurde jedes Auto im Laufe der Zeit auch billiger.
Individualität wird in der heutigen Zeit großgeschrieben. Menschen streben nach Produkten die sie ausmachen, die sie definieren, durch die sie sich von den anderen unterscheiden. Durch das vermehrte Verlangen nach solchen individualisierten Produkten, müssen sich auch die Hersteller mit ihren Produktionsschritten diesem Trend anpassen. Es wird nach Produkten verlangt, die alle ähnlich sind, aber sich doch voneinander unterscheiden. Diese Produkte sollen in einer Fabrik und am besten auf einer Fertigungsstraße hergestellt werden. Um diesen Trend der Individualisierung zu befriedigen, müssen die Produktionslinien flexibler aufgebaut werden, ohne den Vorteil der automatisierten und präzisen Produktion zu verlieren. Es gilt einen Ansatz zu finden, welcher die Variabilität der Produkte in der Herstellung erlaubt und gleichzeitig die Kostenvorteile der Massenproduktion beibehält [Pon09, 1].

1.1 Motivation

In vielen Fabriken kommt es zunehmend zum Einsatz von Robotern [17] in der Produktion und für den Transport der Güter wird das Fließband benutzt. Dadurch wird die Herstellung der Produkte vereinfacht, die Produkte werden qualitativ hochwertiger hergestellt und die Produktion wird zunehmend beschleunigt. Doch auch wenn das Fließband viele Vorteile mit sich bringt, existiert ein ganz großer Nachteil und das ist die geringe Flexibilität. Der Fertigungsprozess wird bei der Fließbandfertigung in kleinere aufeinanderfolgende Stationen mit definierten Arbeitsschritten

aufgeteilt, die in einer definierten Reihenfolge ausgeführt werden müssen. Dabei werden die individualisierten Produkte bei der Herstellung nicht zwingend durch die gleichen Arbeitsschritte geführt und durchlaufen nicht die gleichen Stationen. Wenn aber die Fertigungsstraße einmal aufgebaut ist, ist es schwer, weitere zusätzliche Produktionsschritte in diese einzubauen, oder bestehende zu ändern, ohne dass die Produktion unterbrochen wird um die Fertigungsstraße neu zu konfigurieren. Durch so eine Unterbrechung, bis die Fertigungsstraße neu konfiguriert wird, entstehen signifikante Kosten für den Betreiber [9], [Pon09, 1].

Ein anderer, häufig benutzter Ansatz um Güter zwischen den Stationen zu transportieren, ist die Benutzung autonomer Wägen, welche die geladenen Güter selbständig von A nach B transportieren. Diese erlauben das Hinzufügen oder Ändern von Produktionsschritten in der Fertigung, da ihre Route leichter änderbar ist, als die Route vom Fließband. Jedoch ist diese Änderung im laufenden Betrieb auch nicht möglich und es ist mit einer Unterbrechung der Produktion zu rechnen [Pon09, 1]. Diese fällt aber im Vergleich mit der Änderung beim Fließband geringer aus.
Durch den verstärkten Einsatz von Robotern in der Produktion, bietet sich die Möglichkeit, diese auch zum Transport von Gütern zu verwenden [PPS12]. Die Roboter würden nicht nur für die notwendigen Schritte in der Fertigung benutzt, sondern könnten auch zum Transport verwendet werden. Ein einfaches Prinzip, das dem menschlichen Werfen und Fangen nachempfunden ist: Ein Roboter wirft das Objekt, ein anderer fängt es auf. Dieser Prozess wird solange wiederholt, bis das Objekt, auch über mehrere Zwischenstationen, sein Ziel erreicht hat. So ein System würde deutlich flexibler als ein Fließband oder ein Transport mit dem autonomen Wagen sein. Es würde viel einfacher sein, im Falle von Änderungen der Transportwege, diese ohne einer Unterbrechung der Produktion durchzuführen. Die Rekonfiguration ist beschränkt auf die Aufgaben dem Wurfroboter ein neues Ziel und dem Fangroboter eine neue Quelle vorzugeben [PKFB10, 685]. Zusätzlich wird durch so ein System an Robotern die Zuverlässigkeit des gesamten Transportweges erhöht. Falls einer der Roboter ausfällt und nicht mehr an dem Transport der Güter teilnehmen kann, ist es möglich diesen einfach auszulassen und andere Roboter zum Transport zu benutzen [BFPK09, 680]. Die Abbildung 1.1 zeigt diesen Fall. Der Roboter B fällt aus, der Transportweg wird ohne Unterbrechung über die Roboter D und E stattfinden. Wenn der Roboter B wieder funktionsfähig ist, findet der Transport erneut auf dem Weg A, B, C statt. Auf diese Weiße kann auch eine Lastverteilung durchgeführt werden.

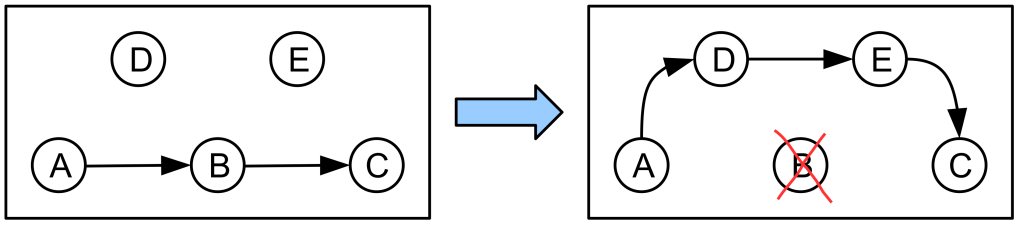

Abbildung 1.1: Ausfall des Roboters B kann über einen Umweg über die Roboter D und E kompensiert werden. Geändert von [Göt15, 3].

Damit dieses Transportsystem funktionieren kann, ist eine Kommunikationsinfrastruktur zwischen den einzelnen Robotern notwendig. Durch diese können die Roboter Informationen austauschen, beispielsweise ob der Roboter bereit ist ein neues Objekt aufzufangen, oder im Falle einer Störung, um die anderen Roboter über den geänderten Transportweg zu informieren. So ein Transportsystem könnte die schnellste Möglichkeit sein, um Güter zwischen zwei Stationen

zu transportieren. Das Werfen und Fangen von Objekten stellt eine direkte Verbindung zwischen zwei Punkten im Raum dar, da es ungefähr dem kürzesten Abstand zwischen diesen Punkten folgt [BFPK09, 680].

1.2 Problemstellung

Die Forschung in diesem Bereich ist nicht nur für akademische Zwecke von großer Bedeutung, sondern auch für die Industrie [PPS12, 136]. Es existieren viele verschiedene Forschungen im Bereich des Materialtransports, der Computer Vision, der Objekterkennung und des Wurftransport-Ansatzes. Trotz der ausgiebigen Forschung wurde noch keine zufriedenstellende Lösung entwickelt. In dieser Diplomarbeit stellt der biologische Ansatz die Grundlage für das Werfen und Fangen dar. Dabei wird die Flugbahnprognose durch die bereits gelernte Erfahrung aus vergangenen Würfen durchgeführt [PKFB10]. Dazu werden viele Flugbahnen von Würfen aufgezeichnet und in einer Datenbank gespeichert. Wird ein Objekt zum Roboterarm geworfen, wird seine Flugbahn mit den bereits aufgezeichneten Flugbahnen verglichen, um eine Vorhersage treffen zu können, wie sich das Objekt weiter in der Luft bewegen wird. Mit dieser Information kann der Roboterarm zur optimalen Position für das Auffangen des Objektes bewegt werden. Eine andere Möglichkeit ist die Flugbahn basierend auf physikalischen Modellen zu berechnen [HS91], [FBH+01], [HS95], [ZXZC12].
Bei einem Fangvorgang ist neben dem richtigen Zeitpunkt und der korrekten Position (dem korrekten Ort) weiters ein langsames Abbremsen des Objekts wichtig. Der Roboterarm soll nach dem Fangen des Objektes für kurze Zeit die Flugbahn des Objektes verfolgen und so die kinetische Energie des Objektes absorbieren. Auf diese Weise kann das Objekt sanft aufgefangen werden [PMB13], [PPS12, 138].
Um das Objekt sicher und präzise fangen zu können muss das System schnell genug sein [Pon09, 52]. Die Bilderfassung, die Bildverarbeitung, die Objekterkennung und die Flugbahnprognose erfordern viele Rechenschritte und benötigen somit eine leistungsfähige Plattform um diese Berechnungen in Echtzeit durchführen zu können. Wenn diese Berechnungen zu lange dauern würden, wird der Roboterarm nicht im richtigen Zeitpunkt zur optimalen Auffangstelle bewegt und das Objekt wird nicht aufgefangen. Götzinger implementiert Berechnungen auf einer Graphics Processing Unit (GPU) [Göt15]. Das System konnte mit einer Bildwiederholungsrate von 130 Bilder pro Sekunde arbeiten und erreichte eine Verbesserung der Ausführugsgeschwindigkeit im Vergleich zu einer Central Processing Unit (CPU) um den Faktor 3,46 - 7,17 [Göt15, 87].
Eine weitere Plattform, auf der solche rechenintensive Aufgaben durchgeführt werden können, sind die Field Programmable Gate Arrays (FPGAs). Dabei handelt es sich um re-programmierbare digitale integrierte Schaltkreise. Diese können vom Benutzer entsprechend programmiert und zusammengeschaltet werden und erzeugen so eine anwendungsspezifische Schaltung. Da es sich dabei um einen Schaltkreis handelt, ist die Ausführung extrem schnell [26]. FPGAs eignen sich besonders für Aufgaben, die parallelisierbar sind. Das sind solche, bei denen die gleichen Aufgaben und Berechnungen auf vielen verschiedenen Daten angewendet werden müssen. Die Logikblöcke, die die Daten verarbeiten, können mehrfach implementiert werden, wodurch die Geschwindigkeit der Verarbeitung erhöht wird. Dadurch, dass die Logikblöcke in den FPGAs parallel arbeiten, müssen verschiedene Verarbeitungsvorgänge nicht um die gleichen Ressourcen konkurrieren. Jede unabhängige Verarbeitungsaufgabe wird einem speziellen Bereich des Chips zugeordnet und kann autonom arbeiten, ohne jeglichen Einfluss von anderen Logikblöcken [16].
Im Gegensatz zu den parallelisierbaren Aufgaben, existieren auch solche, welche nicht parallelisierbar sind, bzw. es keinen Sinn macht, sie parallel zu bearbeiten. Sie müssen also sequentiell

abgearbeitet werden. Und dafür ist die CPU am besten geeignet. Deshalb wurde für die Implementierung der Aufgaben eine System-on-Chip (SoC) Entwicklungsplattform, bestehend aus CPUs und einem FPGA, verwendet.

Im Rahmen dieser Diplomarbeit wird auf folgende Fragen eingegangen:

- Bietet die SoC Plattform genug FPGA Ressourcen um die Bilderfassung, die Bildverarbeitung, die Objekterkennung und die Flugbahnprognose umzusetzen?
- Wie sollen die Aufgaben auf die einzelnen Teile des SoCs aufgeteilt werden? Welche Aufgaben werden in Hardware, welche in Software implementiert?
- Erreicht die SoC Plattform die geforderte Bildrate von 100 Bilder pro Sekunde?

2 Stand der Technik

Die Aufgabe, ein geworfenes Objekt zu fangen, fällt dem Menschen nicht schwer. Bei einem Roboter, ist dies nicht so trivial. Wissenschaftler und Forscher beschäftigen sich mit diesem Thema nämlich bereits seit mehr als 25 Jahren [HS91]. Der Grund war meistens die Demonstration des Fortschritts der entwickelten Technologie, welche notwendig ist um so ein Objekt erfolgreich zu fangen. Zusätzlich gewann jedoch dieses Thema in der letzten Zeit an Aufmerksamkeit, als es auch der Grundgedanke bei dem Thema von Materialtransport wurde. Also Transport von Objekten durch Werfen und Fangen [FWWH+06].

2.1 Transport durch Werfen und Fangen

Grundsätzlich kann der Vorgang des Werfens und Fangens von Objekten, um diese zu transportieren, in vier Aktivitäten aufgeteilt werden. Werfen, Objektverfolgung, Flugbahnprognose und Fangen. [PPS12, 138]. Diese Aktivitäten müssen mehr oder weniger nacheinander durchgeführt werden. Die Abbildung 2.1 veranschaulicht diesen Vorgang.

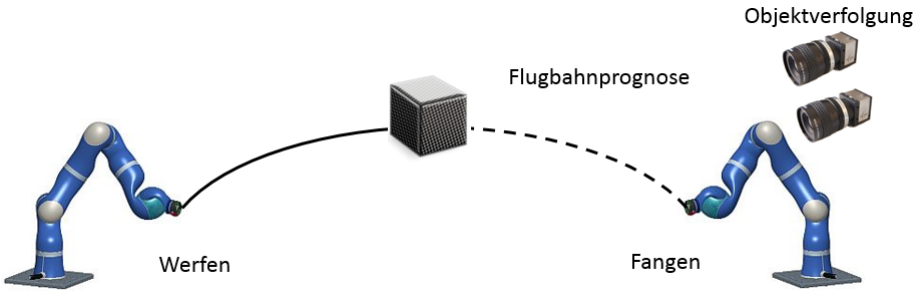

Abbildung 2.1: Überblick über den Ansatz des Transports durch Werfen und Fangen mit den vier Aktivitäten. Geändert von [PPS12, 137].

Im Prinzip muss der Roboter, welcher das geworfene Objekt auffangen soll, wissen, wo sich das Objekt im Raum befindet und wie sich dieses Objekt in der Zukunft bewegen wird. Nur so ist es möglich, den Roboterarm rechtzeitig an die Stelle zu bewegen, wo das Objekt aufgefangen werden soll.

Ähnliche Arbeiten

Wie bereits in Kapitel 1 erwähnt, ist der Fortschritt im Bereich des Materialtransports auch für die Industrie wichtig, jedoch geschieht die Forschung aktuell nur im akademischen Umfeld. Hier sind die Arbeiten [PKFB10], [PPS12], [PMB13], [MVP15], [Göt15], [Pon16], [Mir16] besonders zu erwähnen, da sich diese mit den unterschiedlichen Teilbereichen des Materialtransports beschäftigen und den Fortschritt der Forschung am Institut für Computertechnik der Technischen Universität Wien dokumentieren.

Bereits im Jahre 1995 wurden Versuche mit Werfen und Fangen inklusive der Flugbahneinschätzung und Flugbahnprognose durch Hong und Slotine auf dem Institut of Technology in Massachusetts durchgeführt [HS95]. Ihr Versuchsaufbau mit einem Roboterarm sollte frei geworfene Bälle entgegen dem Roboterarm, auffangen. Die Flugbahn des Balls wurde von zwei Kameras beobachtet. Das System, welches sie entwickelten, basierte auf einer einfachen parabolischen Funktion, an welche die Flugbahn des geworfenen Balls „angepasst“ wurde. Es sind also mindestens zwei Punkte von der Flugbahn notwendig gewesen, um die zukünftige Bewegung des Balls vorhersagen zu können. Mit diesem System waren sie in der Lage 70 bis 80 Prozent aller Würfe erfolgreich zu fangen.
Ein weiterer Versuch wurde im Jahre 2001 durch Frese und seine Kollegen auf dem Institut of Robotics and Mechatronics im German Aerospace Center, durchgeführt [FBH+01]. Sie haben gezeigt, wie ein Roboterarm mit gängigen Komponenten (auf einem Personal Computer (PC) basierendes System), einen Ball fangen kann. Die Flugbahn des Balls wurde auch in diesem Fall mit einem stereo Kamerasystem beobachtet und für die Flugbahnprognose wurde ein erweitertes Kalman Filter verwendet. Von etwa 100 Würfen wurden zwei drittel erfolgreich gefangen. Die nicht erfolgreichen Fangversuche waren dem begrenzten Sichtfeld der Kameras zuzuschreiben. Wenn der Ball kurzzeitig vom Sichtfeld der Kameras verschwunden war, dann konnte die Flugbahnprognose erst spät die Fangposition des Balls ermitteln und somit kam es zu einem fehlerhaften Fangversuch.
Weitere Forschungen am selben Institut von Bäuml im Jahre 2010 [BWH10] basierten auf einem ähnlichen System, wie die Forschung von Frese im Jahre 2001 [FBH+01]. Der Unterschied lag in der Verwendung eines neuen Roboterarms und der Benutzung einer Quad-Core Central Processing Unit (CPU) für die Verarbeitung der aufgezeichneten Daten des stereo Kamerasystems. Für die Berechnungen der Flugbahnprognose war ein 32-Kern Cluster zuständig. Mit dem gleichen erweiterten Kalman Filter wie im Jahre 2001, wurde eine Fangrate von über 80 Prozent aller Fangversuche erreicht.

Als letzter Punkt sei hier die Forschung der KOROS Initiative an der Technischen Universität Wien beschrieben [PPS12]. In dieser Arbeit wurde mit einem KUKA LWR 4 Roboterarm gearbeitet. Zum Werfen wurde eine automatische Vorrichtung benutzt, die den Ball ungefähr immer mit der gleichen Geschwindigkeit von 5 m/s, entgegen dem Roboterarm beschleunigte. Der Abstand zum Roboterarm betrug ungefähr 2,5 m. Für die Beobachung der Flugbahn des Balls wurde ein stereo Kamerasystem verwendet. Die Flugbahnprognose basierte auf einem biologischen Ansatz, bei dem die Flugbahn von dem aktuellen Wurf, mit einer Datenbank an Referenzwürfen verglichen wurde. Näher wird dieser Ansatz in Abschnitt 2.4, unter *Biologischer Ansatz*, beschrieben. Zusätzlich stand die „weiche“ Auffangstrategie im Mittelpunkt, bei der sichergestellt wurde, dass die Kräfte, die beim Fangen auf das Objekt einwirken, minimiert werden [PMB13]. Der biologische Ansatz der Flugbahnprognose stellt die Grundlage für diese Diplomarbeit dar.

2.2 XILINX ZYNQ System-on-Chip Plattform

Diese Diplomarbeit beschäftigt sich mit dem Thema der Implementierung eines Systems für Werfen und Fangen auf einer System-on-Chip (SoC) Plattform, bestehend aus CPUs und einem Field Programmable Gate Array (FPGA) auf einem Chip. Um eine möglichst genaue Prognose der Flugbahn zu erlauben, wird außer einer hohen Auflösung der aufgenommenen Bilder auch eine hohe Zahl an Bildern pro Sekunde benötigt [PKFB10, 688]. Näheres zu der Flugbahnprognose wird in Abschnitt 2.4 beschrieben.

Die hohe Datenmenge an Bildern muss effizient und mit einer Mindestgeschwindigkeit verarbeitet werden. Nur so kann der Roboterarm rechtzeitig mit Informationen über den Ort, wo er das Objekt fangen soll, informiert werden. Diese Geschwindigkeit ist durch die verwendete Bildrate der Kameras definiert. Wenn mit einer Bildrate von 50 Bilder pro Sekunde aufgenommen wird, bedeutet dies eine Zeit von nur 20 *ms* zwischen zwei aufeinander folgenden Bildern. Für diese Berechnung wird die folgende Formel 2.1 benutzt.

$$Zeit = \frac{1}{BilderProSekunde} \tag{2.1}$$

Bei 100 Bildern pro Sekunde ist die Zeit zwischen den Bildern nur mehr halb so lang, nämlich 10 *ms* usw. Je höher also die Bildrate, umso schneller muss das System die Daten verarbeiten können. Denn bereits nach 10 *ms* wird von der Kamera das nächste Bild geliefert und die Ressourcen für die Verarbeitung müssen frei sein, um dieses neue Bild verarbeiten zu können.

Wie in der Diplomarbeit von Pongratz angesprochen wurde, kann die Ausführungszeit der Algorithmen für die Bildverarbeitung, mittels eines FPGAs gesteigert werden [Pon09, 67]. Eine weitere effiziente und leistungsfähige Plattform für diese Algorithmen ist die Graphics Processing Unit (GPU). Ein Ansatz, welcher diese Plattform benutzt, wurde in der Diplomarbeit vom Götzinger [Göt15] beschrieben.
Eine SoC Plattform bietet sich also als eine weitere mögliche und leistungsfähige Plattform an, welche die Deadline von 10 *ms* unterbieten kann und somit eine Bildrate von 100 Bilder pro Sekunde erreichen kann. Der Grund, warum eine SoC Plattform (CPUs und FPGA), statt einer, die nur auf einem FPGA basiert, benötigt wird, liegt in der notwendigen Kombination von den CPUs und dem FPGA. Denn, die Algorithmen, welche die Deadline von 10 *ms* erreichen müssen, werden im FPGA implementiert und auf den CPUs wird eine embedded Linux Version mit den benötigten Treibern und der Steuerungssoftware für die Kameras ausgeführt.

Als Entwicklungsplattform wurde somit das Mini-ITX Board der Firma AVNET ausgewählt [29]. Dieses basiert auf der ZYNQ-7000 SoC Serie der Firma XILINX. Die genaue Chipbezeichnung des verbauten SoC Chips lautet XC7Z100. Dieser Chip integriert zwei ARM CPUs, welche jeweils mit einer Frequenz von 800 *MHz* getaktet sind. Als FPGA ist die Kintex-7 FPGA Serie verbaut. Diese bietet 444 Tausend Logikzellen, 277400 Look-Up Tables (LUTs), 554800 Flip-Flops, 3020 KB Block Random Access Memory (RAM) (bzw. 755 36-kbit Blöcke) und 2020 programmierbare Digital Signal Processor (DSP) Zellen [19]. Auf der Entwicklungsplattform steht dem SoC Chip insgesamt 2 GB an RAM, eine Peripheral Component Interconnect Express 2.0 (PCIe) Schnittstelle, diverse Debugschnittstellen, vier Universal Serial Bus (USB) 2.0 Schnittstellen, eine Ethernet Schnittstelle, eine Universal Asynchronous Receiver/Transmitter (UART) Schnittstelle und weitere nützliche Peripherie zur Verfügung [28].

Die Abbildung 2.2 zeigt das Blockdiagramm der ZYNQ Architektur.

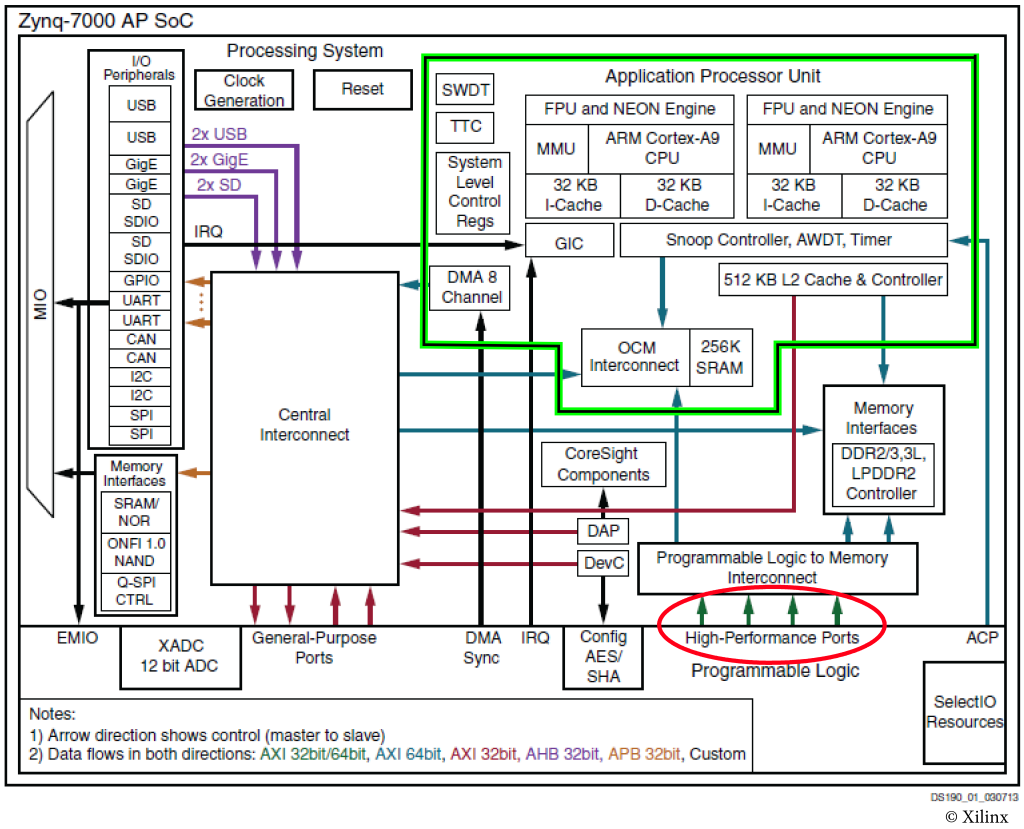

Abbildung 2.2: Die ZYNQ Architektur. Geändert von [CEES14, 17].

2.3 Objekterkennung

Die Objekterkennung ist ein essentieller Teil des Systems für Werfen und Fangen. Das geworfene Objekt wird verfolgt und anhand dieser Informationen kann im späteren Verlauf die Flugbahn ermittelt werden, damit das Objekt erfolgreich gefangen werden kann. Die Verfolgung des Objektes wird aufgrund der hohen Dynamik der Bewegung hauptsächlich mittels Video Kameras durchgeführt. Es gibt verschiedene Versuche, welche die auf einer Kamera aufbauen [BFK08], welche die mittels zwei Kameras arbeiten [PKFB10], [INI96], als auch welche die mehr als zwei Kameras benutzen [BWH10], [BSW+11].
Alle Arbeiten bisher haben sich mit der Erkennung von punktsymmetrischen oder axialsymmentrischen Objekten beschäftigt [HS91], [HS95], [FBM+08], [BFPK09], [PMB13]. Symmetrische Objekte, wie Bälle oder Zylinder sind aufgrund ihrer Form deutlich einfacher zu erkennen. Mit steigender Komplexität der Form des Objektes, steigt auch die Komplexität der Objekterkennung [PPS12, 139]. Deshalb wird in dieser Diplomarbeit die Objekterkennung auf das Erkennen eines geworfenen Balls konzipiert.

Das Prinzip, um ein sich bewegendes Objekt in aufeinander folgenden Bildern zu erkennen, ist einfach und kann leicht erzielt werden. Eine wichtige Voraussetzung ist, dass die Kameras fest sind und sich zwischen den Aufnahmen nicht bewegen.
Es existieren zwei Methoden. Die erste Methode benutzt zwei unmittelbar aufeinander folgende Bilder und berechnet die Differenz dieser. Der Ball wird dann an der Stelle erkannt, wo die Pixeländerung höher ist, als eine vorher definierte Schwelle [TKBM99], [GW07]. Diese Methode hat einen Nachteil falls sich das bewegende Objekt langsamer bewegt, als die Bilder aufgenommen werden. In diesem Fall würde sich der Ball im aktuell aufgenommenen Bild mit dem Ball im vorher aufgenommenen Bild überlappen. Diesen Effekt, welcher „ghosting“ genannt wird, zeigt die Abbildung 2.3. Dadurch würde die neue Position des Balls falsch ausgewertet werden. Der Vorteil, welchen diese Methode bietet, ist die Resistenz gegen den sich ändernden Hintergrund und sie benötigt nicht viel Rechenleistung [Pon09, 6].

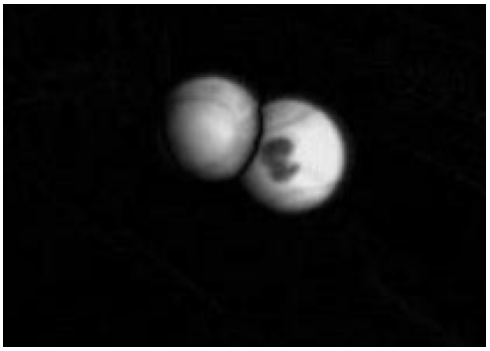

Abbildung 2.3: Der „ghosting“ Effekt bei der Differenz zweier aufeinander folgenden Bilder. Links der Ball in der alten Position, rechts der Ball in der neuen Position [Pon09, 5].

Die zweite Methode, um ein sich bewegendes Objekt in den Bildern zu erkennen, ist die Benutzung von einem statischen Hintergrundbild. Dieses soll keine der zu erkennenden Objekte beinhalten, damit diese nicht die Objekterkennung verfälschen. Dabei wird von dem jeweils aktuell aufgenommenen Bild das Hintergrundbild subtrahiert. In der Theorie, falls der Hintergrund in dem aktuell aufgenommenen Bild identisch mit dem vorher aufgenommenen statischen Hintergrund ist, dann bleiben nur die Pixel von dem Ball übrig. Der Hintergrund würde sich durch die Subtraktion zu Null ergeben. In der Praxis kann es jedoch vorkommen, dass sich der Hintergrund aufgrund von äußeren Einflüssen geringfügig verändert. Dazu kann auch die Änderung der Beleuchtung der Szene, oder das Flackern der Lichter zählen. Eine Möglichkeit um dieser Fehlerquelle entgegen zu wirken, ist die Pixel vom Hintergrundbild nicht als fixe Werte anzunehmen, sondern als Intervall von Werten, die die Pixel an der jeweiligen Stelle annehmen können. Dadurch werden Pixel, die sich in dem Intervall befinden als Hintergrundpixel gewertet und Pixel, welche höher, oder niedriger als das Intervall sind, als Ball gewertet [FBH+01, 1625]. Der große Vorteil im Gegensatz zu der ersten Methode, mit der Subtraktion der aufeinander folgenden Bilder, ist der nicht vorhandene „ghosting“ Effekt. Die Abbildung 2.4 zeigt das Ergebnis der beschriebenen Methode der Hintergrundsubtraktion.

Mit dieser Methode ist es möglich den Ballmittelpunkt ungefähr zu bestimmen. Der Ballmittelpunkt wird durch den Mittelwert aller Pixel, welche zum Ball gehören, ermittelt. Falls jedoch Pixel, welche nicht zum Ball gehören, trotzdem als Ball-Pixel gewertet werden, dann ist ein Fehler passiert. Analog verhält es sich wenn Ball-Pixel als nicht zum Ball gehörende Pixel gewertet werden. Durch solche Fehler kann der ermittelte Ballmittelpunkt von dem tatsächlichen Ballmit-

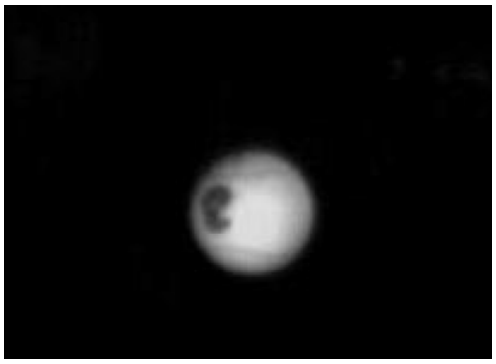

Abbildung 2.4: Hintergrundsubtraktion, es kommt zu keinem „ghosting“ Effekt [Pon09, 6].

telpunkt abweichen. Solche Fehler können durch einen optimierten Objekterkennungsalgorithmus verringert werden, jedoch nicht komplett eliminiert werden [Pon09, 7].
Um eine bestmögliche Objekterkennung mit dem tatsächlichen Ballmittelpunkt zu erreichen ist auf andere Methoden auszuweichen.

Die Erkennung von Kreisen (circle detection) ist eine der am häufigsten benutzten Anwendungen im Rahmen der Computer Vision. Deshalb existieren viele verschiedene Methoden um Kreise zu erkennen [DACN02], zwei davon werden am häufigsten benutzt: der RANSAC Algorithmus und die Hough Transformation. Bevor jeweils eine der Methoden angewendet werden kann, ist ein weiterer Verarbeitungsschritt notwendig - das Erstellen eines Kantenbildes. Dafür wird das Originalbild mittels Filterung in ein Bild umgewandelt, in dem Kanten, der im Bild vorhandenen Objekte, dargestellt werden. An jedem einzelnen Bildpunkt eines Bildes wird durch Untersuchung des umgebenden Bereiches, der Farbwertgradient berechnet. Der Unterschied in dem Farbwertgradient wird dann als eine Kante im Bild dargestellt und trennt somit flächige Bereiche in diesem Bild voneinander. Es entsteht ein Kantenbild, wie es die Abbildung 2.5 veranschaulicht. Einer der bekanntesten Filter, die ein Kantenbild erzeugen, ist der Canny Edge Detector, der bereits 1986 publiziert wurde [Can86].

Abbildung 2.5: Umwandlung des Originalbildes in ein Kantenbild [14].

Die Basis für die Implementierung, unter anderem der Objekterkennung, ist in dieser Diplomarbeit eine SoC Plattform, bestehend aus CPUs und einem FPGA. Die oben genannten Methoden, die Hough Transformation und der RANSAC Algorithmus, sind nur bedingt für die Zwecke dieser

Diplomarbeit einsetzbar. Es gibt zahlreiche Arbeiten zu den Themen der Umsetzung eines Canny Edge Detectors [XVCK14], der Hough Transformation [ZIN13], [TAD00], [ZIN14a], [ZIN14b], als auch des RANSAC Algorithmus [TSHS13], auf einem FPGA. In allen diesen Arbeiten wurde die jeweilige Methode (Canny Edge, RANSAC, Hough Transformation) als Einzige auf dem FPGA realisiert. Sprich, es wurden keine weiteren Ressourcen für andere Teile des Projektes verwendet, welche nicht für die jeweilige Methode notwendig waren. Die verfügbaren Ressourcen auf einem FPGA sind aber begrenzt. In dieser Diplomarbeit wird außer der Objekterkennung, auch der Algorithmus für die Flugbahnprognose und weitere notwendige Logik im FPGA implementiert und somit sind die verfügbaren Ressourcen (siehe Abschnitt 2.2) nicht zu 100 Prozent nur für die Objekterkennung nutzbar.

Der Canny Edge Detector aus der Arbeit [XVCK14] wurde auf einem Virtex-5 FPGA der Firma XILINX realisiert und benötigte ungefähr 24000 Logikzellen, 450 der 36-kbit Blöcken an Block RAM (BRAM) und 224 DSP Zellen. Der Detector arbeitete mit 100 *MHz* und für ein Bild mit 512x512 Pixel benötigte er 0,721 *ms*.

Der Hough Transformation Algorithmus, der 2013 publiziert wurde [ZIN13], wurde auf einem Virtex-6 FPGA ebenfalls von XILINX realisiert. Dieser benötigte ungefähr 40500 Logikzellen, 91 der 36-kbit Blöcken an BRAM und zusätzlich 91 DSP Zellen. Das Design arbeitete mit einer maximalen Frequenz von 247,525 *MHz* und benötigte bei dieser Frequenz für ein Bild mit 300x300 Pixel, eine Zeit von 367 μs.

Ein Jahr später wurde ein verbesserter Algorithmus publiziert [ZIN14a], welcher auf einem Virtex-7 FPGA von XILINX implementiert wurde. Dieser wurde verbessert in der Ausführungszeit und der maximalen Frequenz, die erreicht werden konnte, nämlich 260,016 *MHz*. Bei einem Bild mit 300x300 Pixel benötigte die Ausführung 351 μs. Die Menge an benötigten Ressourcen wurde aber im Gegensatz zu dem vorherigen Design erhöht. Diese Implementierung benötigte ungefähr 80100 Logikzellen und 184 der 36-kbit Blöcken an BRAM. Also beide Ressourcenansprüche wurden etwa verdoppelt. Einzig die Menge an DSP ist mit 13 Stück, verringert worden.

Im gleichen Jahr wurde ein spezieller Ansatz für die Erkennung von Kreisen mittels der Hough Transformation publiziert [ZIN14b]. Umgesetzt wurde dieser auch auf einem Virtex-7 FPGA von XILINX und benötigte etwa 20500 Logikzellen, 155 der 36-kbit Blöcken an BRAM und bis zu 398 DSPs. Bei einer maximalen Frequenz von 181,812 *MHz* benötigte der Algorithmus für ein Bild mit 300x300 Pixel, eine Zeit von 2,51 *ms*.

Von der Geschwindigkeit der Ausführung der einzelnen Algorithmen würde sich die gesetzte Deadline von 10 *ms* (für die Bildrate von 100 Bilder pro Sekunde) ausgehen. Die Bilder, welche von der Kamera geliefert werden, sind nicht verarbeitet. Es müssen also beide Algorithmen, Canny Edge Detector und Hough Transformation, im FPGA implementiert werden. Wenn man die beiden Algorithmen, den Canny Edge Detector aus der Arbeit [XVCK14] und den Hough Transformation Algorithmus aus der Arbeit [ZIN14b], implementiert, dann beträgt die Ausführungszeit weniger als 4 *ms*. Die beiden Implementierungen wurden aber auf unterschiedlichen FPGA Generationen realisiert. Zwischen den Generationen gibt es technologische Unterschiede, deshalb kann hier nur eine ungefähre Schätzung der benötigten Ressourcen angegeben werden. Zusätzlich beziehen sich die Angaben der Ausführungszeiten und der benötigten Ressourcen jeweils auf andere Bildgrößen und Arbeitsfrequenzen. Die folgende Tabelle 2.1 fasst die wichtigsten benötigten Ressourcen der beiden Arbeiten zusammen und stellt sie den verfügbaren Ressourcen, der in dieser Diplomarbeit

verwendeten Entwicklungsplattform, gegenüber.

	Canny Edge Det. [XVCK14] benötigt	Hough Trans. [ZIN14b] benötigt	zusammen benötigt	verfügbar in dieser Arbeit	Ausnutzung in %
Logik-zellen	~24000	~20500	~44500	444000	~10,0
36-kbit BRAM	~450	~155	~605	755	~80,1
DSP	224	398	622	2020	~30,8

Tabelle 2.1: Vergleich der benötigten Ressourcen der Arbeiten [XVCK14] und [ZIN14b] bei einer Implementierung auf der Mini-ITX Plattform.

Wie in Tabelle 2.1 ersichtlich, bereits eine Implementierung dieser zwei Algorithmen im FPGA auf der Entwicklungsplattform führt zu einer BRAM Ausnutzung von 80%. Wenn nur diese beiden Algorithmen im FPGA implementiert werden müssen, dann sind die Ressourcen ausreichend. Jedoch die zusätzlich benötigte Logik und vor allem die Flugbahnprognose benötigt viel lokalen Speicher in Form von BRAM. Dafür wären nur mehr etwa 150 der 36-kbit Blöcke übrig, was nicht ausreichend ist. Zudem sollte eine hohe Ausnutzung (90 %) vermieden werden, da bei der Implementierung Schwierigkeiten mit dem Routing und dem Timing auftreten können.

Fast Circle Detection

Ein weiteres Verfahren, das für die Erkennung von Kreisen in Bildern benutzt werden kann, nutzt statt einem Kantenbild ein Gradientenbild. Genauer, die jeweilige Richtung des Bildgradienten. Die publizierte Arbeit [RFQ03] stellt die Grundlage dieses Verfahrens dar. Ähnlich zu der Hough Transformation, auch bei diesem Verfahren muss vorher das Originalbild einem weiteren Verarbeitungsschritt unterzogen werden - es muss das Gradientenbild erstellt werden. Dazu werden die ersten beiden Schritte vom Canny Algorithmus, nämlich die Glättung und die Berechnung der Gradienten, durchgeführt. Bei dem Fast Circle Detection Algorithmus wird die Hough Transformation für die Erkennung von Kreisen modifiziert und angepasst für die Erkennung von Kreisen, welche deutlich heller oder dunkler als der Hintergrund sind. Die Idee dabei ist, die Symmetrie von den Gradientenvektoren an solchen Kreisen auszunutzen. Entlang des Umfangs eines Kreises sind immer paarweise Gradientenvektoren vorhanden, die entgegengesetzt zueinander angeordnet sind. Dieser Ansatz eignet sich sehr gut für die Erkennung von Kreisen, in unserem Fall von dem geworfenen Ball, da der Ball im Vergleich zum Hintergrund deutlich heller ist (siehe Abbildungen 2.3 und 2.4).
An dieser Stelle wird aber die Methode anhand eines Beispieles mit umgekehrten Farbgradienten beschrieben (anhand der erwähnten Publikation [RFQ03]), das von einem dunklen Kreis an einem hellen Hintergrund besteht (siehe Abbildnung 2.6 links). Die Anwendung für unsere Bedürfnisse (heller Kreis, dunkler Hintergrund) ist analog und an der Funktionalität ändert sich nichts, denn in so einem Fall kann entweder mit einem Negativbild gearbeitet werden, oder die Richtungen der Vektoren werden in die entgegengesetzte Richtung zeigen.

Der erste Schritt ist die Berechnung der Gradientenvektoren. Diese haben im Beispiel, die in Abbildung 2.6 rechts dargestellte Form. Die Richtung ist vom Kreismittelpunkt nach Außen, weil der Kreis dünkler ist als der Hintergrund. Der Gradientenvektor gibt allgemein die Änderung der Pixelintensität an und zeigt in Richtung steigender Intensität. In unserem Fall handelt es sich um

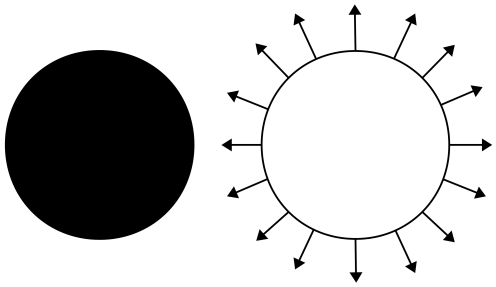

Abbildung 2.6: Links ein dunkler Kreis an einem hellen Hintergrund, rechts die Gradientenvektoren von diesem Kreis. Geändert von [RFQ03, 881].

Bilder in Graustufen, in welchen die Intensitäten der einzelnen Pixel in 8 bit gespeichert werden. Der Wertebereich von 8 bit liegt bei vorzeichenlosen Werten zwischen 0 und 255, wobei 0 eine schwarze Farbe und 255 eine weiße Farbe vom Pixel bedeutet. Die Gradientenvektoren zeigen also von niedriger Intensität in Richtung höherer Intensität, also von schwarzen Pixeln in Richtung von weißen Pixeln.
Aufgrund der Symmetrie von Kreisen, existiert für jeden solchen Vektor ein Vektor in entgegengesetzter Richtung, ein sogenannter Paarvektor. Wie die Abbildung 2.7 links zeigt, hat der Vektor $\overrightarrow{\mathbf{V}_1}$ seinen Paarvektor $\overrightarrow{\mathbf{V}_2}$, der folgende zwei Bedingungen erfüllt:

i. Der Winkel α, definiert als die absolute Differenz zwischen den Richtungen der Vektoren $\overrightarrow{\mathbf{V}_1}$ und $\overrightarrow{\mathbf{V}_2}$, soll annähernd 180 Grad betragen.

ii. Der Winkel β zwischen der Gerade, die P_2 und P_1 verbindet (Basis von $\overrightarrow{\mathbf{V}_2}$ und $\overrightarrow{\mathbf{V}_1}$) und dem Vektor $\overrightarrow{\mathbf{V}_1}$ soll annähernd 0 Grad betragen.[1] $\overrightarrow{P_2P_1}$ soll in die gleiche Richtung wie $\overrightarrow{\mathbf{V}_1}$ zeigen.

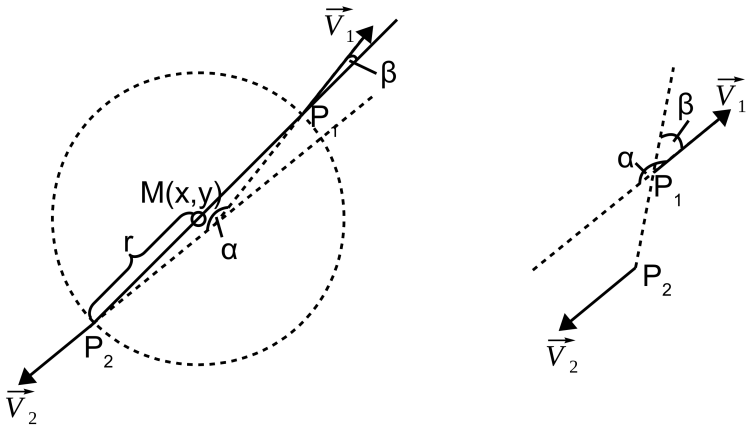

Abbildung 2.7: Links Vektoren $\overrightarrow{\mathbf{V}_1}$ und $\overrightarrow{\mathbf{V}_2}$, die beide Bedingungen i. und ii. erfüllen (Paarvektoren), rechts Vektoren $\overrightarrow{\mathbf{V}_1}$ und $\overrightarrow{\mathbf{V}_2}$, die Bedingung ii. nicht erfüllen. Geändert von [RFQ03, 882].

[1] oder der Vektor $\overrightarrow{\mathbf{V}_1}$ soll in die entgegengesetzte Richtung vom Vektor $\overrightarrow{\mathbf{V}_2}$ zeigen, da sie fast parallel sind aufgrund der Bedingung i.

Im zweiten Schritt werden alle Paarvektoren anhand der obigen Bedingungen gesucht. Unnötige Vektoren werden mittels Bedingung ii. herausgefiltert. Dieser Zusammenhang ist in Abbildung 2.7 rechts ersichtlich. Die Vektoren $\overrightarrow{\mathbf{V}_1}$ und $\overrightarrow{\mathbf{V}_2}$ erfüllen zwar Bedingung i., sind jedoch aufgrund von ii. keine Paarvektoren.
Im dritten Schritt wird jeweils ein Kreis für alle erkannten Paarvektoren definiert, dessen Mittelpunkt in der Mitte zwischen den Punkten P_1 und P_2 liegt und dessen Radius die Hälfte vom Abstand zwischen diesen beiden Punkten ist. In Abbildung 2.7 links ist ein solcher Kreis für die Vektoren $\overrightarrow{\mathbf{V}_1}$ und $\overrightarrow{\mathbf{V}_2}$ eingezeichnet. Falls bekannt ist, in welchem Bereich der Radius von dem gesuchten Kreis liegt, kann eine dritte Bedingung aufgestellt werden, die dafür sorgt, dass die Paarvektoren, die einen Kreis außerhalb von diesem Bereich definieren, herausgefiltert werden.
Im vierten und letzten Schritt wird eine zweidimensionale Akkumulatormatrix, mit jeweils einer Dimension für die Koordinaten x und y vom Kreismittelpunkt, definiert. Die erkannten Kreise aus dem vorherigen Schritt werden in diese Akkumulatormatrix eingetragen und jeweils die Zellen in dieser Matrix für überlappende Kreismittelpunkte hochgezählt. Durch eine Suche nach dem lokalen Maximum kann der tatsächliche Kreis und dessen Mittelpunkt schlussendlich gefunden werden. Dieser Ansatz der Maximumsuche ist identisch mit der klassischen Hough Transformation. Im Gegensatz zu dieser, benötigt der Algorithmus mit den paarweisen Gradientenvektoren, statt einer dreidimensionalen, nur eine zweidimensionale Akkumulatormatrix, wodurch der Speicherbedarf stark verringert werden kann. Der Nachteil ist, dass keine exakt konzentrischen Kreise mit unterschiedlichen Radien gefunden werden können, da alle solche Kreise den exakt gleichen Mittelpunkt haben und somit voneinander nicht unterscheidbar sind.

Laut den Autoren wurde der Algorithmus in Matlab simuliert und erreichte für Bilder mit 256x256 Pixel mehr als eine 700-fache Beschleunigung im Vergleich zur klassischen Hough Transformation. Für Bilder mit 512x512 Pixel erreichte die Beschleunigung sogar mehr als das 1000-fache im Vergleich zur klassischen Hough Transformation. Die Toleranz der Parameter α und β war mit jeweils 5 Grad definiert und die Gradientenvektoren wurden mittels eines Sobelfilters berechnet. Zusätzlich ist diese Methode sehr robust gegen das Rauschen im Bild in Form von *salt-and-pepper noise* (spärlich auftretende weiße und schwarze Pixel im Bild) und die Genauigkeit vom Algorithmus bleibt unverändert solange das Rauschen unter 25% liegt. Bei der klassischen Hough Transformation war die Grenze unter identischen Bedingungen bereits bei 10% erreicht. Über diesen Grenzen stieg die Fehlerrate exponentiell an, jedoch war bei der klassischen Hough Transformation dieser Anstieg steiler und erreichte die maximale Fehlerrate bereits bei 38%. Bei diesem Fast Circle Detection Algorithmus war diese Fehlerrate erst bei 65% erreicht [RFQ03, 884].

Triangulation

Um die tatsächliche Position vom Ball im Raum zu ermitteln ist zumindest ein stereo Kamerasystem notwendig. Es gibt auch Ansätze, welche nur eine Kamera verwenden [BFK08], diese sind aber sehr empfindlich gegenüber diversen Störungen und so ein System kann bereits bei geringen Störungen deutliche Abweichungen der ermittelten Positionen zeigen [FBH+01, 1623].
Es wird somit ein stereo Kamerasystem verwendet. Diese zwei Kameras sind fix in einem definierten Abstand horizontal nebeneinander montiert [Pon09, 19]. Das Prinzip, um aus den zweidimensionalen Bildern von dem selben Objekt im Raum, eine räumliche Position zu bekommen, ist ähnlich dem menschlichen Sehen von Objekten im Raum. Der Mensch benutzt auch zwei Augen und erzeugt aus diesen zwei zwei-dimensionalen Bildern, ein drei-dimensionales Bild [2], [10].

Die Objekterkennung ermittelt den Ballmittelpunkt im linken und rechten Bild. Dieser Ballmittelpunkt ist der Punkt, für den die Position im Raum ermittelt werden soll. Dies geschieht mit Hilfe der Triangulation. Dabei werden die beiden Koordinaten der Ballmittelpunkte vom linken und rechten Bild in die Weltkoordinaten tranformiert. Der Roboterarm kann sich entsprechend dieser Weltkoordinaten im Raum bewegen und den geworfenen Ball auffangen [Pon09, 36]. Die Abbildung 2.8 zeigt diesen Zusammenhang.

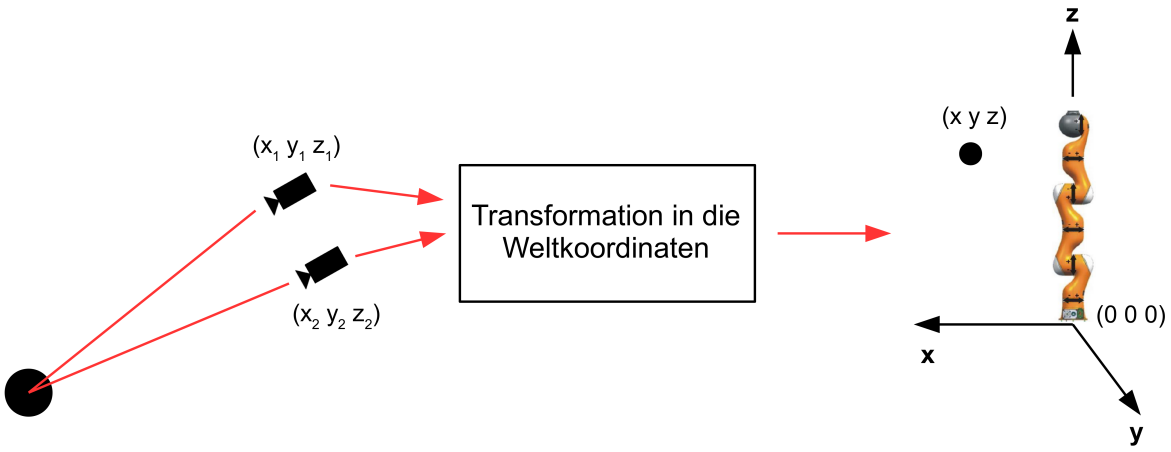

Abbildung 2.8: Transformation der Information über den Ballmittelpunkt in die Weltkoordinaten. Geändert von [Göt15, 19], [PMB13, 29].

Für die Objekterkennung, Lokalisierung im Raum mittels der Triangulation und auch die anschließende Flugbahnprognose ist auch die Position der beiden Kameras wichtig. Je näher die Kameras an dem Objekt positioniert sind, umso bessere Ergebnisse können erzielt werden. Deshalb sind die Kameras hinter der Wurfvorrichtung positioniert und beobachten den Ball, wie er sich in Richtung des Roboterarms bewegt. Auf diese Weise wird die Beobachtung des Balls bereits am Anfang des Wurfs sehr präzise und die Flugbahnprognose kann sehr früh den weiteren Flugverlauf ermitteln [FBH+01, 1624, 1629].

2.4 Flugbahnprognose

Ein weiterer essientieller Teil, ohne dem das Werfen und Fangen von Objekt nicht funktionieren könnte, ist die Flugbahnprognose. Sie arbeitet mit der Information über die aktuelle Position des Balls im Raum. Aber diese Position ist noch nicht ausreichend, um den Ball auch zu fangen. Denn der Ball kann vom Roboter erst dann gefangen werden, wenn er die Information über den Ort, wo er den Ball fangen soll, früh genug bekommt. Es dauert nämlich die Zeit $t_{roboter}$, bis der Roboterarm sich in Richtung der entsprechenden Stelle bewegt. Die Flugbahn vom Ball muss also vorhergesagt werden, wie es die Abbildung 2.9 veranschaulicht.

In den meisten Arbeiten ist die Grundidee für die Flugbahnprognose ähnlich. Das Objekt wird mittels eines stereo Kamerasystems beobachtet, damit die Position im Raum ermittelt werden kann. Nach der Zeit $t_{vorhersage}$ ist die Information über die Ballbewegung vorhanden und die erste Vorhersage über den weiteren Verlauf des Balls kann getroffen werden [HS91, 383]. Die

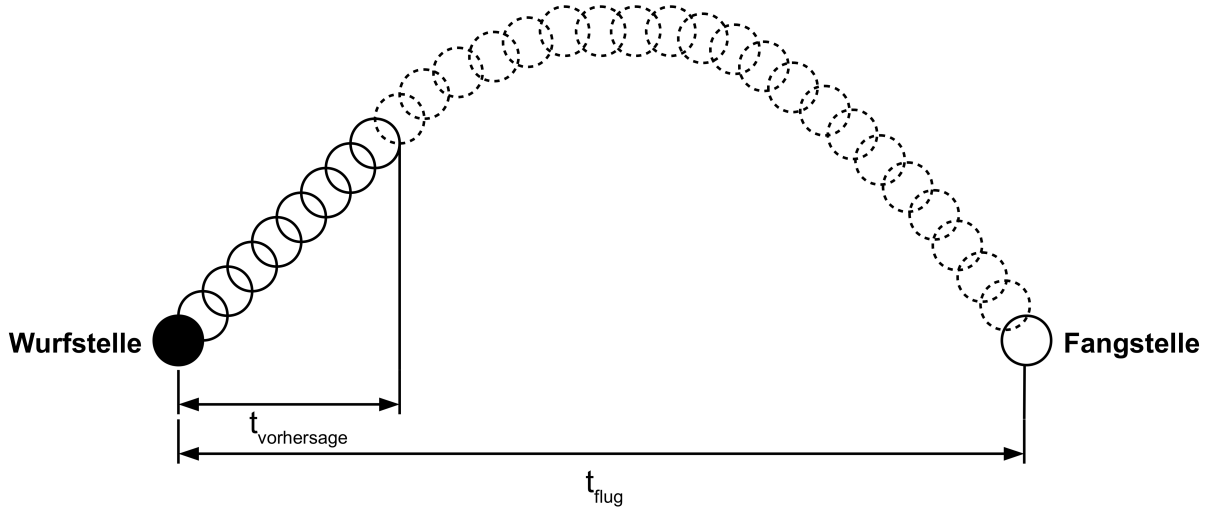

Abbildung 2.9: Der Ball wird von links nach rechts geworfen, nach der Zeit $t_{vorhersage}$ kann der weitere Flug vorhergesagt werden. Geändert von [Göt15, 9].

Zeit $t_{vorhersage}$ ist abhängig einerseits von der verwendeten Methode der Flugbahnprognose, andererseits aber auch von der Aufstellung der Kameras und wie diese die Szene mit dem Wurf beobachten.

Die hier eingeführten Zeiten t_{flug}, $t_{vorhersage}$ und $t_{roboter}$ beeinflussen das Werfen und Fangen von Objekten und bestimmen, ob der Fangvorgang erfolgreich wird, oder nicht. Der Zusammenhang zwischen diesen Zeiten ist in der folgenden Gleichung 2.2 gegeben.

$$t_{vorhersage} + t_{roboter} \leq t_{flug} \tag{2.2}$$

Die Zeit für die Vorhersage der Flugbahn plus die Zeit, welche der Roboter benötigt um sich in die Richtung der Fangstelle zu bewegen muss kleiner gleich der Zeit sein, die der Ball benötigt, um diese Fangstelle zu erreichen. Wenn diese Gleichung nicht erfüllt wird, wird der Roboter die Fangstelle zu spät erreichen und der Ball wird nicht erfolgreich gefangen.
Sobald die ersten Positionen vom Ball im Raum erkannt wurden, können verschiedene Methoden angewendet werden, um den weiteren Flugverlauf zu berechnen. Es muss ein passendes Modell gefunden werden, das den Flugverlauf am besten beschreibt. Falls das Modell einen Flugverlauf beschreibt, der von der Ballbewegung deutlich abweicht, muss dieses angepasst werden, oder komplett durch ein anderes Modell ersetzt werden [Pon09, 11].

Das einfachste Modell, um die Flugbahn zu berechnen, ist das physikalische Modell. In der einfachsten Variante wird nur die Erdanziehungskraft, welche auf den Ball während des Flugs einwirkt, berücksichtigt. In dieser Form benötigt es wenig Rechenleistung, aber es werden viele der anderen Effekte vernachlässigt, welche während des Fluges auf den Ball einwirken [Pon09, 11].

Ein weiters Modell, ist das auf Polynomfunktionen basierende. Dabei werden die beobachteten Positionen des Balls entlang des Flugs beobachtet und es wird versucht diese in eine Polynomfunktion „anzupassen“ [Pon09, 13]. Die Ordnung der Polynomfunktion gibt an, wie genau die

Funktion und die Flugbahn des Objektes übereinstimmen. Messungen haben ergeben, dass die höheren Ordnungen nur gering bessere Ergebnisse liefern und dass je höher die Ordnung ist, umso mehr sind die Funktionen auf Fehler empfindlich [PKFB10, 686]. Die Funktionen zweiter Ordnung gelten als ein Kompromiss zwischen der Empfindlichkeit und einem stabilem Verhalten [PKFB10, 688].

Beim Vergleich der beiden Modelle, hat das Modell basierend auf den Polynomfunktionen schlechter abgeschnitten. Wenn im physikalischen Modell zusätzliche Effekte auf den Ball berücksichtigt werden, ist dieser Ansatz der bessere [PKFB10, 689]. Andere Arbeiten benutzen das erweiterte Kalman Filter mit einem dynamischen Modell, bei dem auch nicht-lineare Effekte, wie der Luftwiderstand, berücksichtigt werden [FBH+01], [BWH10]. Dieses Filter ist jedoch schwer zu implementieren und benötigt viel Rechenleistung aufgrund der Benutzung von Jacobi Matrizen, welche teilweise schwer zu berechnen sind [3], [4].

Die höchste erreichte Fangrate war bei diesen und den ähnlichen Arbeiten beschrieben in Abschnitt 2.1 etwa 80 Prozent. Diese ist bereits ziemlich hoch, jedoch nicht hoch genug um einen zuverlässigen Transport durch Werfen und Fangen zu ermöglichen. Es muss ein Modell herangezogen werden, welches die meisten Faktoren berücksichtigt, die auf den Ball im Flug einwirken.

Biologischer Ansatz

Dieser Ansatz ist neu im Bereich des Transports mittels Werfen und Fangen. Er basiert auf den Arbeiten [PKFB10] und [PPS12]. Biologischer Ansatz bedeutet, dass die Vorhersage der Flugbahn auf biologischem Wege gelöst wird. Sprich, es wird wie beim Menschen, die Lernfähigkeit benutzt sich Erfahrung anzueignen und später wird auf dieses gelernte Wissen wieder zurückgegriffen. Im technischen Aspekt bedeutet dies, dass viele Trajektorien des geworfenen Balls aufgezeichnet und in einer Datenbank gespeichert werden. Beim aktuellen Wurf wird dann die Flugbahn vom Ball mit der Datenbank von gespeicherten Trajektorien verglichen. Dies ermöglicht anschließend die Vorhersage des weiteren Ballfluges und auch der Stelle, wo der Ball vom Roboterarm gefangen werden muss [PPS12, 141].
Je umfangreicher die Datenbank, also je mehr Trajektorien aufgenommen wurden, umso besser und genauer kann die Flugbahn vorhergesagt werden. Mit diesem Ansatz werden theoretisch alle Einflüsse auf den Ball während des Flugs miteinbezogen. Um den aktuellen Flug mit der Menge an Trajektorien zu vergleichen, wird der k-NN Algorithmus (k-Nearest Neighbors) verwendet [MVP15, 34]. Die Gleichung 2.3 zeigt die einfache Form der Suche mit dem k-NN Algorithmus.

$$y = \frac{1}{k} \sum_{n=1}^{k} x_{\mathrm{n}} \tag{2.3}$$

Es wird der Mittelwert der k am besten passenden Trajektorien, zu der aktuellen Trajektorie vom geworfenen Ball, berechnet. Dabei ist x_n eine der k besten Trajektorien aus der Datenbank und y die resultierende Trajektorie. In der Arbeit [MVP15] wird dieser Algorithmus für die Flugbahnprognose simuliert und erreicht eine Fangrate zwischen 85 und 92 Prozent. Somit ist dieser Ansatz besser für die Vorhersage der Trajektorie geeignet, als andere Ansätze basierend auf physikalischen Modellen, Modellen mit Polynomfunktionen, oder Modellen die auf Kalman Filter basieren [MVP15, 35].

2.5 System-on-Chip Partitionierung Hardware/Software

Wie bereits in Abschnitt 2.2 erwähnt wurde, beschäftigt sich diese Diplomarbeit mit der Machbarkeit der Implementierung des Systems für Werfen und Fangen auf einer SoC Plattform, bestehend aus CPUs und einem FPGA auf einem Chip. Der Vorteil so eines Systems, ist die Möglichkeit parallelisierbare Aufgaben als Logik in den FPGA auszulagern und somit die Ausführungszeit von Programmen, im Vergleich zu der Ausführung auf einer CPU, deutlich zu beschleunigen. Zusätzlich hat man die Möglichkeit die CPU für alle anderen Aufgaben zu benutzen, die keine Beschleunigung benötigen, bzw. die keine Beschleunigung erlauben. Früher konnte so eine Auslagerung von parallelisierbaren Aufgaben in den FPGA auch durchgeführt werden, jedoch waren der FPGA Chip und die CPU bzw. der Microcontroller als separate Bauteile verbaut. Der große Nachteil so einer Architektur war die geringe Bandbreite zwischen diesen beiden Chips, höhere Leistungsaufnahme, komplexes Printed Circuit Board (PCB) Layout und limitierte Anschlussmöglichkeiten der beiden Chips [11, 1].
Die Integration von ARM CPUs und einem FPGA auf einem Chip behebt alle genannten Nachteile, vor allem bietet diese Architektur eine sehr schnelle Verbindung mit hoher Bandbreite zwischen den CPUs und dem FPGA. Realisiert wird diese mittels des Advanced eXtensively Interface (AXI) Bus [11, 2].

Es existieren viele verschiedene Arbeiten, welche sich mit dem Thema der Bildverarbeitung und der Hardware (HW)/Software (SW) Partitionierung beschäftigen. HW/SW Partitionierung ist in der Literatur auch unter dem Begriff HW/SW Co-Design zu finden. Bereits im Jahre 2006 wurde eine Arbeit publiziert [KLL+06], welche eine ARM CPU und einen nicht näher beschriebenen Hardware-Beschleuniger Chip für die Bildverarbeitung benutzt hat. Um Echtzeitfähigkeit zu erreichen wurde die Aufgabe entsprechend auf den Beschleuniger und die CPU aufgeteilt.

Im Jahre 2008 wurde eine Plattform für die Erkennung von Fahrzeugen vorgestellt [ACS08], welche einen Video-stream mit 25 Bildern pro Sekunde verarbeiten konnte. Die Bilder von der Kamera waren in Graustufen und hatten eine Auflösung von 384x288 Pixel. Implementiert wurde diese Plattform auf dem FPGA Virtex-2 PRO, welcher auch eine zwei Kern CPU mit 300 *MHz* integriert hatte. Diese CPU war damals noch keine ARM CPU, sondern basierte auf einer PowerPC Architektur. Als Betriebssystem wurde Linux verwendet.

Von den neueren Arbeiten ist die vom Jahre 2012 zu erwähnen [XCZB12], in dieser wird das HW/SW Co-Design nicht zwischen FPGA und CPU durchgeführt, sondern zwischen FPGA und DSP. Da diese Implementierung auf zwei separaten Chips aufgebaut wurde und nicht auf einem SoC, wurde hier teilweise ein anderer Ansatz zur HW/SW Partitionierung gewählt [XCZB12, 3381]. Außer dem Ansatz, dass die parallelisierbaren Aufgaben in HW (FPGA) implementiert sein sollen, wurde hier auch auf die Reduktion der Daten, die zwischen dem FPGA und dem DSP kommuniziert werden müssen, geachtet. Die Reduktion der Daten zwischen FPGA und CPU ist auf modernen SoCs mit FPGA und CPU auf einem Chip nur mit geringerer Priorität zu sehen, da diese SoCs hochperformante Datenleitungen für diesen Datenaustausch integrieren [15], [GCAMJ15].

Als letzte Arbeit sei hier eine vom Jahre 2015 erwähnt [CMCGSS15], die eine ähnliche Plattform als auch eine Kameraanbindung mittels USB, wie in dieser Diplomarbeit, benutzte. Das Ziel war, den Video-stream von einer USB High-Definition (HD) Kamera mit einer maximalen Auflösung von 1280x720, mittels drei unterschiedlichen Filteralgorithmen, welche im FPGA implementiert wurden, zu verarbeiten. Diese Filteralgorithmen wurden als separate Intellectual

Property (IP) Cores realisiert. Zur Implementierung wurde die Zedboard Plattform herangezogen (SoC XC7Z020-CLG-484-1) und als Betriebssystem wurde Linux verwendet. Es wurden nicht näher erläuterte Änderungen in den Treibern durchgeführt, um die Daten von der Kamera direkt in die Puffer der IP Cores zu laden [CMCGSS15, 1696]. Der Treiber für die Kamera war im Linux Kernel vorhanden und musste nicht vom Hersteller geliefert werden. Bei einer Auflösung von 640x480 erreichte die Plattform etwa 55 Bilder pro Sekunde, bei der vollen Auflösung der Kamera 1280x720 nur mehr etwa die Hälfte, 25 Bilder pro Sekunde.

Die zuletzt genannte Arbeit war die Einzige, die eine USB Kamera als Quelle für die Bilddaten verwendet hat. Alle anderen gefunden Arbeiten benutzten entweder die High-Definition Multimedia Interface (HDMI) Schnittstelle, oder eine andere Anbindung als Quelle für die Bilddaten. Alle diese Arbeiten haben aber eines gemeinsam - die HW/SW Partitionierung wurde dazu benutzt, um zu entscheiden, welche Aufgaben im FPGA und welche auf der CPU ausgeführt werden. Die Entscheidung war eindeutig. Aufgaben welche viele Daten mit gleichen Instruktionen verarbeiten, wie z.B. diverse Bildverarbeitungsalgorithmen, sind parallel ausführbar und sollen aufgrund der höheren Effizienz im FPGA implementiert werden. Im Gegensatz dazu, Aufgaben welche seriell abgearbeitet werden müssen, beispielsweise weil Abhängigkeiten in den Daten bestehen, sollen besser auf der CPU implementiert werden. Zusätzlich ist die CPU auch für komplexere Programmabläufe, welche viele if/else Anweisungen beinhalten, besser geeignet [7].

FPGA

Ein FPGA ist zwischen einer CPU und einem Application Specific Integrated Circuit (ASIC) angesiedelt. Es bietet die Möglichkeit der Programmierung, wie eine CPU und erreicht Geschwindigkeiten nahe der ASICs. Ein FPGA besteht aus endlich vielen programmierbaren Logikzellen, welche immer wieder rekonfiguriert werden können. Diese Programmierung geschieht durch ein sogenannten **bitstream**, welcher in den FPGA eingespielt wird. Dieser bitstream besteht, im Gegensatz zu einem kompilierten Programm, der maschinelle Instruktionen für die CPU beinhaltet, aus einer Sequenz von Bits, welche die integrierten Schaltungen und Logikzellen entsprechend konfigurieren. Grob betrachtet integriert ein FPGA Hunderttausende Logikzellen, LUTs und Flip-Flops, Tausende DSPs, Hunderte BRAMs, Blöcke für Eingänge und Ausgänge (IO) usw. Natürlich ändert sich die Menge an Ressourcen mit der verwendeten Technologie und der Ausführung des FPGAs. Die hier genannten Ressourcen sind in dem, in Abschnitt 2.2 erwähnten SoC, integriert. Die Abbildung 2.10 veranschaulicht einige der Bestandteile in einem FPGA. Nähere Beschreibung dieser und der weiteren Bestandteile kann dem Buch [CEES14, 23 ff.] entnommen werden.

Um den FPGA zu programmieren ist eine Hardware Description Language (HDL) notwendig. Mit dieser beschreibt man die Struktur, das Design und die komplette Funktionalität der Schaltungen. Die beiden meist genutzten Sprachen sind Verilog [5] und Very High Speed Integrated Circuit (VHSIC) HDL, oder kurz VHDL [6]. Beide dieser Sprachen erfordern das Erlernen neuer Konzepte und neuer Regeln. Die Beschreibung großer und komplexer Aufgaben, welche im FPGA implementiert werden sollen, kann schnell unübersichtlich werden. Dadurch kann für das Erstellen eines Designs viel Zeit benötigt werden [22, 2]. Um diese Zeit zu verkürzen und den Designvorgang zu vereinfachen, wurde eine Methode entwickelt, die eine Beschreibung der Aufgaben und Algorithmen in der C, C++, oder SystemC Sprache erlaubt. Diese Beschreibung wird dann anschließend in die Verilog, oder VHDL Sprache umwandelt. Die Methode wird High Level Synthese, oder kurz HLS genannt. Es existieren verschiedene Softwareprodukte, welche eine HLS unterstützen. Da im Rahmen dieser Diplomarbeit auf der Entwicklungsplattform von XILINX

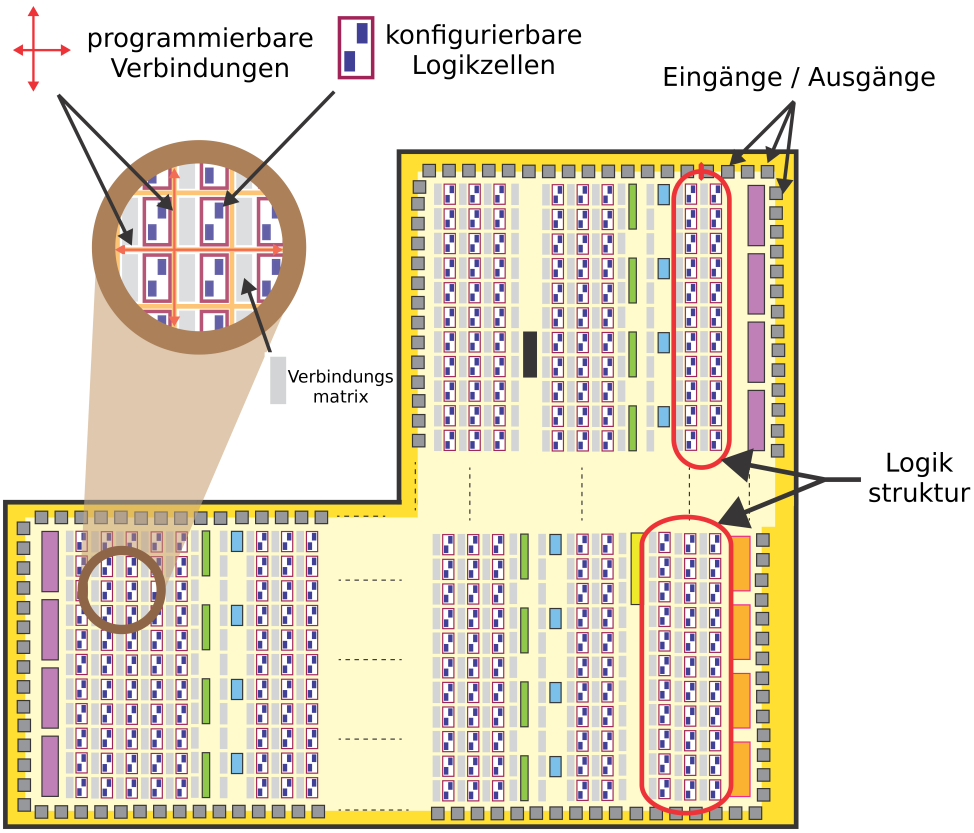

Abbildung 2.10: Einige der Bestandteile eines FPGAs. Geändert von [CEES14, 23].

gearbeitet wird, liegt es nahe, die HLS Lösung von XILINX zu verwenden. XILINX bietet für diese Zwecke die Vivado HLS Programmierumgebung [27] an. Die Benutzung von HLS statt der Hardware Beschreibungssprachen Verilog/VHDL erleichtert und beschleunigt deutlich das Designen von komplexen Algorithmen.

In den Anfängen von FPGAs war die Programmierung auf der register-transfer Ebene (register-transfer level (RTL)) mittels HDL Sprachen, der Stand der Technik. Heute tendieren viele Entwickler hin zu höheren Programmiersprachen wie C und C++. Die Programmierung mittels HDL, statt C/C++ kann in der Informatik mit der Programmierung mittels Assembler, statt den höheren Programmiersprachen verglichen werden [20, 7]. Die Abbildung 2.11 zeigt die vergleichsweise lange Zeit für die Entwicklung eines Designs für FPGA mittels der HDL Programmiersprache. Zusätzlich sind im Diagramm die Entwicklungszeiten und die erreichbare Leistung für andere Plattformen eingetragen.

Es ist ersichtlich, dass die erste funktionierende Version der Software bei der CPU, GPU und DSP bereits früh in der Entwicklung vorhanden ist. Bei jeder Plattform ist ein gewisser zusätzlicher Aufwand notwendig, um die maximale Leistung zu erreichen.
Zusätzlich wird die benötigte Entwicklungszeit für die gleiche Software auf einer FPGA Plattform angegeben. Die optimierte Version für die FPGA Plattform, programmiert mittels HDL, erreicht die höchste Leistung. Aber die benötigte Zeit, um auf diese Leistung zu gelangen, ist mehr als in

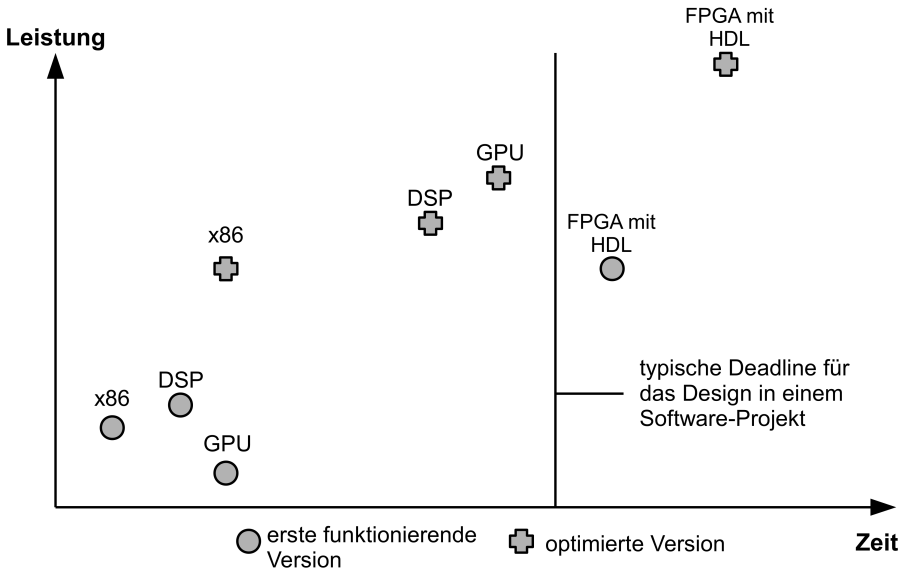

Abbildung 2.11: Entwicklungszeit vs. Leistung für unterschiedliche Plattformen, FPGA wird mittels der Hardware Description Language (HDL) programmiert. Geändert von [20, 7].

einem typischen Software-Projekt den Entwicklern zur Verfügung steht. Deshalb wurden in der Vergangenheit FPGAs nur für solche Aufgaben verwendet, die eine Leistung benötigten, welche nicht durch andere Methoden erreicht werden konnte [20, 7].

Die Abbildung 2.12 vergleicht HLS, für die Programmierung von FPGA, mit anderen Plattformen hinsichtlich der Leistung und den Entwicklungszeiten.

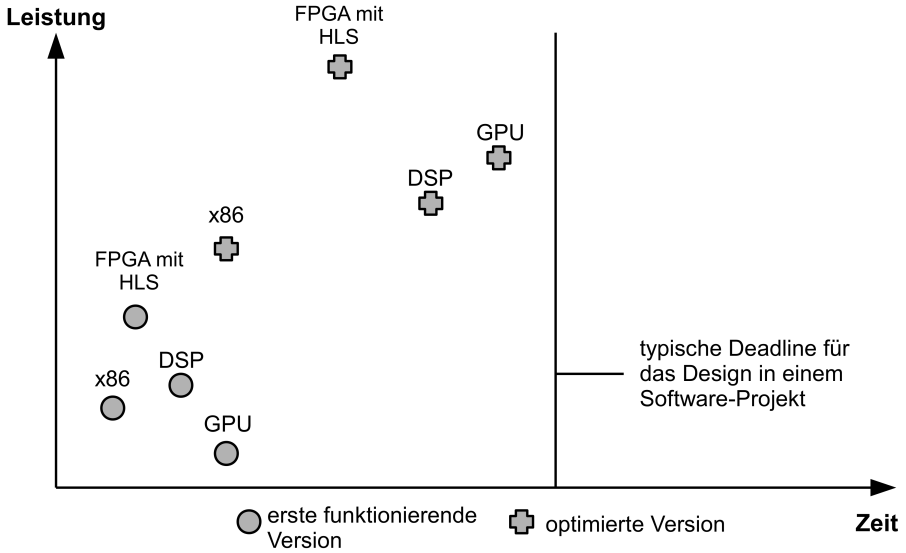

Abbildung 2.12: Entwicklungszeit vs. Leistung für unterschiedliche Plattformen, Design für FPGA wird mittels der High Level Synthese (HLS) entworfen. Geändert von [20, 8].

Die mittels HLS erstellte Designs sind typischerweise nicht so effizient in der Nutzung der Ressourcen, wie „von Hand“ erstellte Designs [TSS15, 20]. In dieser Diplomarbeit wird aufgrund der Einfachheit und des schnelleren Designs trotzdem die HLS Methode verwendet und die Algorithmen in der C Programmiersprache geschrieben.

Da aber eine HW beschrieben wird, muss auch die HW Beschreibung mittels der C Programmiersprache entsprechend „angepasst“ werden. Es können nicht alle Möglichkeiten, die die C Programmiersprache bietet, in HW umgewandelt werden. Zum Beispiel können rekursive Funktionen, dynamische Speicher Allokation und Systemaufrufe, wie beispielsweise Dateimanipulation usw. nicht umgesetzt werden. Zusätzlich muss bedacht werden, dass eine HW „programmiert“ wird und der Algorithmus muss in vielen Fällen verändert werden, damit die von HLS entworfene HW effizient ist. Es passiert selten, dass ein Algorithmus sofort funktioniert und die erwarteten Resultate erzielt. In HLS gibt es spezielle Anweisungen, sogenannte **directives**, mittels welcher dem Compiler zusätzliche Informationen für die Umsetzung bekanntgegeben werden. Durch die Benutzung dieser kann man die Erzeugung der HW steuern und kann dem Compiler angeben, wie er die HW schlussendlich umsetzen soll.

Ein gutes Beispiel dafür, was unter „Anpassung“ gemeint ist, ist die Verwendung der selben Unterfunktion an mehreren Stellen im Programmablauf. Für die Effizienz der HW ist es besser, wenn diese Unterfunktion mehrmals implementiert wird (identische Kopien) und an den Stellen, wo sie aufgerufen wird jeweils auf eine andere Kopie verwiesen wird. Da die Daten durch die Logik bei der Ausführung „fließen“, erzeugt die Verwendung der selben Unterfunktion einen Flaschenhals, weil auf die selbe Logik von mehreren Stellen aus zugegriffen wird. Solche Optimierungen müssen vom Entwickler manuell durchgeführt werden und werden nicht automatisch von HLS erkannt.

3 Projektbeschreibung

Dieses Kapitel beschreibt die notwendigen Teile des Projektes für die Implementierung des Systems für Werfen und Fangen von Objekten. Das Objekt das geworfen wird ist ein Tennisball, der aufgrund der punktsymetrichen Form, eine deutlich einfachere Erkennung erlaubt. Bereits in Abschnitt 2.1 wurde näher auf die einzelnen Teile des Systems eingegangen. Zu diesen gehören grob das Werfen, die Objektverfolgung, die Flugbahnprognose und das Fangen. Da jeder dieser Teile an sich eine Menge an Forschung und Entwicklung benötigt, wird auf das System für Werfen und Fangen nicht näher eingegangen und es wird auf bereits vorhandene Systeme aus früheren Arbeiten zurückgegriffen. Der Fokus dieser Diplomarbeit liegt darin, die Möglichkeit der Implementierung der Objektverfolgung und der Flugbahnprognose auf einer System-on-Chip (SoC) Entwicklungsplattform, bestehend aus Central Processing Units (CPUs) und Field Programmable Gate Array (FPGA) auf einem Chip, zu evaluieren.

3.1 Aufbau der Umgebung zum Werfen und Fangen

Die bereits vorhandene Vorrichtung zum Werfen wird in der Arbeit [MVP15, 30] gezeigt und beschrieben und wird auch in dieser Diplomarbeit verwendet. Zum Fangen wird der Roboterarm KUKA LWR 4+ verwendet [MVP15, 33]. Der Abstand zwischen der Wurfvorrichung und dem Roboter beträgt etwa 2,5 m. Der Ball wurde gleichzeitig mittels zwei IDS uEye UI337xCP-M Kameras beobachtet, welche konvergent mit einem Basisabstand von etwa 1 m hinter der Wurfvorrichung montiert waren. Die Szene wurde mittels einer Kombination aus 500 W halogen und Light-Emitting Diode (LED) Scheinwerfer ausgeleuchtet. Die beiden Kameras unterstützen eine Auflösung von 2048x2048 Pixel bei einer Bildrate von 80 Bilder pro Sekunde. Zusätzlich kann die Auflösung in mehreren Schritten in den beiden Richtungen verringert werden und dadurch werden höhere Bildraten ermöglicht. Es wurde die Auflösung von 2048x768 gewählt, bei welcher die Bildrate von über 110 Bilder pro Sekunde erreichbar ist. Das Ziel in dieser Diplomarbeit ist die Bildrate von 100 Bilder pro Sekunde zu erreichen.
Die Entwicklungsplattform wurde bereits detailiert in Abschnitt 2.2 beschrieben. Es handelt sich um das Mini-ITX der Firma AVNET, bestückt mit dem SoC XC7Z100. Die detaillierte Chipbezeichnung lautet xc7z100ffg900-2. Dieser SoC ist der aktuell leistungsstärkste Chip der ZYNQ-7000 Serie. Die Plattform wurde vom Institut für Computertechnik der Technischen Universität Wien zur Verfügung gestellt.
Zur Entwicklung wurde ein Personal Computer (PC) mit einer Intel Core i5-4460 CPU @ 3,20 GHz mit 16 GB Random Access Memory (RAM) verwendet. Als Betriebssystem kam die 64-bit

Ubuntu 14.04 LTS Linux Distribution zum Einsatz. Die Entwicklungsumgebung Vivado, SDK und Vivado HLS von XILINX wurde in den Versionen 2015.4 und 2016.2 verwendet.

3.2 Hardware/Software Partitionierung

In folgender Abbildung 3.1 ist grob das Gesamtsystem bestehend aus den einzelnen Teilsystemen dargestellt.

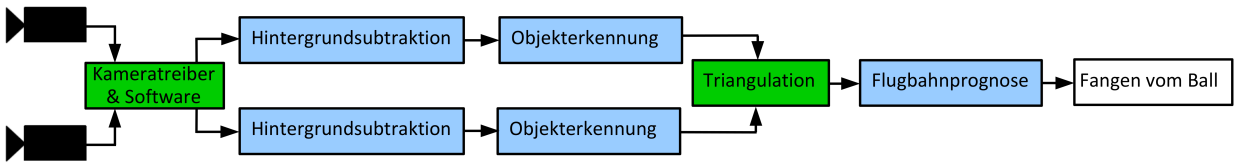

Abbildung 3.1: Übersicht über das Gesamtsystem. Die farblich markierten Teilsysteme werden im Rahmen dieser Diplomarbeit entworfen und implementiert. Die blauen Teile werden im FPGA implementiert, die grünen Teile werden auf der CPU ausgeführt.

Das System, das implementiert werden soll, kann weiter in kleine Teilaufgaben zerlegt werden. Einzig das System für das eigentliche Fangen von dem geworfenen Ball ist nicht mehr Teil dieser Diplomarbeit. Die Information von der letzten Verarbeitungsstufe wird dem Fangsystem über ein User Datagram Protocol (UDP) Packet übermittelt. Das Gesamtsystem wurde daher in diese Aufgaben unterteilt:

- Laden der Bilder von den zwei Kameras in der Auflösung von 2048x768 Pixel
- Subtraktion des vorher aufgenommenen Hintergrundbildes von diesen Bildern
- Ausschneiden der Area-of-Interest (AoI) mit dem Ball um ein 300x300 Pixel Bild zu erstellen
- Erkennen vom Ballmittelpunkt in den 300x300 Pixel Bildern
- Triangulation der ermittelten Ballmittelpunkte um die räumliche Position zu bestimmen
- Flugbahnprognose durchführen

Wie bereits in Abschnitt 1.2 und 2.5 erwähnt wurde, sollen parallelisierbare Aufgaben, wie beispielsweise die Objekterkennung, im FPGA implementiert werden, um die parallelisierbarkeit solcher Aufgaben auszunutzen und die Ausführung mittels der parallelen Architektur vom FPGA zu beschleunigen. Als solche Aufgaben wurden folgende Teile identifiziert: die Subtraktion vom Hintergrundbild, das Erstellen eines Ausschnitts aus einem Bild, die Objekterkennung des Balls mit der Bestimmung des Ballmittelpunktes und die Flugbahnprognose. Alle diese Aufgaben behandeln viele unterschiedliche Daten mit den gleichen Instruktionen und können somit parallel ausgeführt werden. Deshalb ist die Entscheidung vernünftig, diese im FPGA zu implementieren. Im Gegensatz steht die Aufgabe der Triangulation, die Umwandlung der zweidimensionalen Koordinaten des erkannten Ballmittelpunktes in den zweidimensionalen Bildern, in die dreidimensionale Position des Ballmittelpunktes im Raum. Dieses Verfahren behandelt keine unterschiedlichen

Daten mit gleichen Instruktionen, deshalb ist dieses Verfahren nicht parallelisierbar. Bei der Hardware (HW)/Software (SW) Partitionierung geht es in erster Linie darum, die benötigten Aufgaben und Algorithmen für die Ausführung optimal in der Architektur zu platzieren. Sprich, Aufgaben die Bilder verarbeiten laufen schneller auf einem FPGA und Aufgaben, wie die Triangulation, bei denen Anweisungen seriell ausgeführt werden, laufen schneller auf einer CPU.
Deshalb wird die Triangulation, als Teil einer Software, auf der CPU ausgeführt. Der Kameratreiber, der für die Kommunikation mit den Kameras zuständig ist, wird auch auf der CPU ausgeführt. Alle anderen Aufgaben werden im FPGA als Intellectual Property (IP) Cores implementiert. In Abbildung 3.1 ist diese Unterscheidung farblich gekennzeichnet.

3.3 Systemkomponenten

In diesem Abschnitt werden die einzelnen Aufgaben beschrieben, die entworfen wurden, damit das System für Werfen und Fangen realisiert werden konnte.

3.3.1 Anbindung der Kameras

Die Daten von den zwei Kameras werden jeweils über eine Universal Serial Bus (USB) 3.0 Schnittstelle bereitgestellt. Die Mini-ITX Plattform verfügt aber nur über USB 2.0 Schnittstellen, welche nicht die benötigte Datenrate erlauben. Die Entwicklungsplattform bietet aber eine Peripheral Component Interconnect Express 2.0 (PCIe) Schnittstelle mit vier Lanes (x4) an, welche an einer Lane (x1) eine Datenrate von 5 *Gbit/s* erlaubt. Insgesamt erlaubt die PCIe Schnittstelle bei vier Lanes also eine Datenrate von 20 *Gbit/s*. In diese PCIe Schnittstelle wurde eine USB 3.0 Erweiterungskarte installiert, um das Mini-ITX Board mit USB 3.0 Schnittstellen nachzurüsten. Dabei musste auf die Ausführung der Erweiterungskarte geachtet werden. Die üblichen USB 3.0 Erweiterungskarten für PCIe sind nur für eine Lane (x1) konzipiert und bieten gleich mindestens zwei USB 3.0 Schnittstellen an. Die maximale Datenrate von PCIe 2.0 in der Ausführung mit einer Lane (x1) ist eben 5 *Gbit/s*. Jedoch die maximale Datenrate von einer USB 3.0 Schnittstelle ist auch 5 *Gbit/s*. Im Falle, wenn zwei USB 3.0 Schnittstellen benutzt werden, also zwei Geräte an diesen Schnittstellen angeschlossen werden und beide Geräte gleichzeitig Daten übertragen, dann kann dies nur Abwechselnd passieren und die resultierende Datenrate wird theoretisch halbiert. Somit erreicht jedes der angeschlossenen Geräte nur eine Datenrate von 2,5 *Gbit/s*.
Um diesen Flaschenhals zu beheben, wurde eine Erweiterungskarte für PCIe 2.0 mit vier USB 3.0 Schnittstellen in der Ausführung für vier Lanes (x4) benutzt [18]. Diese Erweiterungskarte von StarTech mit der Modellbezeichnung PEXUSB3S44V bietet vier USB 3.0 Schnittstellen an, von denen jede einen separaten Kanal mit einer maximalen Datenrate von 5 *Gbit/s* nutzen kann. Jede der vier USB 3.0 Schnittstellen benutzt somit eine der vier verfügbaren Lanes und dadurch wird keine der Lanes mit anderen USB 3.0 Schnittstellen geteilt. Bei einem gleichzeitigen Betrieb von den zwei USB 3.0 Kameras, kann jede die maximale Datenrate von 5 *Gbit/s* erreichen, ohne dass sich die Kameras beim Datentransfer abwechseln müssen.

3.3.2 Betriebssystem, Treiber, Cross-Compiler

Als Betriebssystem für die Entwicklungsplattform wurde die Linux Kernel Version 4.4, welche von XILINX zur Verfügung gestellt wird, verwendet. Im Laufe der Entwicklung wurden etliche

andere Versionen getestet, 3.17, 3.19, 4.0 und 4.4. Die zuletzt genannte ist die, welche mit allen benötigten Treibern, am stabilsten funktioniert. Der Kernel benötigt, um korrekt zu funktionieren, auch ein root filesystem, auf welchem die Bibliotheken, Treiber und Programme abgespeichert sind. Als root filesystem wurde die Linaro Ubuntu Vivid Developer Distribution in der Version 15.12 verwendet [12]. Als Treiber für die Kameras wurde der embedded Linux Treiber vom Hersteller IDS in der Version 4.80 zur Verfügung gestellt und verwendet [8]. Für die benötigten Direct Memory Access (DMA) IP Cores von XILINX, welche in späterem Abschnitt 3.5 näher beschrieben werden, wurde auf einen Treiber ausgewichen, welcher von anderen Entwicklern unter der Projektbezeichnung „xilinx_axidma“ zur Verfügung gestellt wurde [1]. XILINX bietet zwar Linux Treiber für die meisten seiner IP Cores, inklusive dem Treiber für die DMA, an, aber diese können nicht in selbst geschriebenen Programmen benutzt werden.
In Linux unterscheidet man zwischen dem sogenannten Kernelspace und Userspace. Diese strikte Trennung trennt den Speicherbereich und dient hauptsächlich dem Schutz des Speichers vor absichtlichen und unabsichtlichen Fehlern, die beim Zugriff auf den Speicher auftreten können. So ein Fehler könnte im schlimmsten Fall den Speicherbereich vom Kernel überschreiben und somit das Betriebssystem zum Absturz bringen. Im Kernelspace befinden sich außer dem Kernel selbst, noch weitere Kernelerweiterungen und die meisten Treiber. Im Gegensatz dazu, befinden sich im Userspace die Programme und nur einige wenige Treiber. Die Treiber von XILINX sind im Kernelspace angesiedelt, also nur der Kernel und andere Module können diese Treiber nutzen. Da die Programme wiederum im Userspace angesiedelt sind, kann auf diese Treiber nicht direkt zugegriffen werden. Der „xilinx_axidma“ Treiber [1] für die DMA bildet eine Brücke zwischen dem Userspace und dem Kernelspace und somit konnte die DMA auch vom selbst geschriebenen Programm benutzt werden.
Für die Kompilierung der selbst geschriebenen Programme wurde die Methode der Cross-Compilation benutzt, wobei auf einem Entwicklungsrechner basierend auf der x86/x64 Architektur, das Programm für die ARM Architektur geschrieben und mit dem Cross-Compiler kompiliert wurde. Hierzu wurde die Linaro Toolchain mit Hard Float (HF) der Version 15.02 verwendet [13]. So ein Cross-Kompiliertes Programm kann nur auf der Zielarchitektur, in diesem Fall auf einer ARM Architektur, ausgeführt werden. Eine andere Möglichkeit, um die Programme für die ARM Architektur zu kompilieren, wäre die Programme direkt auf der Zielplattform mit dem dort verfügbaren Compiler zu kompilieren. Diese Möglichkeit ist aber für die häufige Kompilierung umständlich und wurde somit nicht benutzt.

3.3.3 Hintergrundsubtraktion

Wie in Abbildung 3.1 dargestellt wurde, ist die erste Verarbeitungsstufe im FPGA, die ausgeführt wird sobald die ersten Daten von der Kamera zur Verfügung stehen, die *Hintergrundsubtraktion*. Die Aufgabe dieser Stufe ist sehr wichtig, da sie die ungefähre Position vom Ball im Bild berechnet. Dazu wurde von beiden Kameras ein Hintergrundbild aufgenommen, ich welchem sich kein Ball befindet. Die Auflösung des Hintergrundbildes beträgt 2048x768 Pixel. Wenn ein Bild von der Kamera geliefert wird, das einen Ball im Flug beinhaltet, dann wird von diesem aktuellen Bild mit dem Ball, das früher aufgenommene Hintergrundbild subtrahiert. Das Bild von der Kamera mit dem Ball hat ebenfalls eine Auflösung von 2048x768 Pixel. Dieser Vorgang wird in der folgenden Gleichung 3.1 gezeigt.

$$P_{\text{subtraktion}}(x, y) = P_{\text{aktuell}}(x, y) - P_{\text{hintergrund}}(x, y) \tag{3.1}$$

Die Intensität vom Pixel des subtrahierten Bildes an der Stelle (x,y) ist somit die Intensität des aktuellen Bildes an der Stelle (x,y) minus der Intensität des Hintergrundbildes an der Stelle (x,y). Von der Kamera werden Bilder mit Graustufen Werten geliefert, somit wird die Intensität der Pixel im Bereich von 0 bis 255 gespeichert und für die Speicherung ist der Datentyp *unsigned char* am Besten geeignet. Wie bereits in Abschnitt 2.3 erwähnt wurde, kann sich das Hintergrundbild aufgrund von äußeren Einflüssen, beispielsweise durch die sich ändernde Intensität vom Licht, ändern. Deshalb kann es vorkommen, dass in der Gleichung 3.1 ein negativer Wert berechnet wird, falls die Intensität vom Hintergrundbild größer ist, als die vom aktuell aufgenommenen Bild an der gleichen Stelle (x,y). Bei der Speicherung mittels des Datentypes *unsigned char*, kommt es bei einem negativen Wert zu einem Überlauf und als Resultat wird ein Pixel einer hohen Intensität gespeichert. Um diesem Problem entgegenzuwirken, wird die Gleichung 3.1 mit einer Betragsfunktion erweitert, wie die folgende Gleichung 3.2 veranschaulicht.

$$P_{\text{subtraktion}}(x, y) = |P_{\text{aktuell}}(x, y) - P_{\text{hintergrund}}(x, y)| \tag{3.2}$$

Durch die Betragsfunktion kann es im subtrahierten Bild zu geringen Fehlern in den Intensitäten des Hintergrundes kommen (aus negativen Intensitäten werden positive erzeugt). Die Intensitäten der Pixel vom Ball sind davon jedoch nicht betroffen, da diese immer eine höhere Intensität aufweisen als der Hintergrund an der gleichen Stelle (x,y) und werden durch die Betragsfunktion nicht verändert. Die Abbildung 3.2 zeigt den Vorgang der Hintergrundsubtraktion.

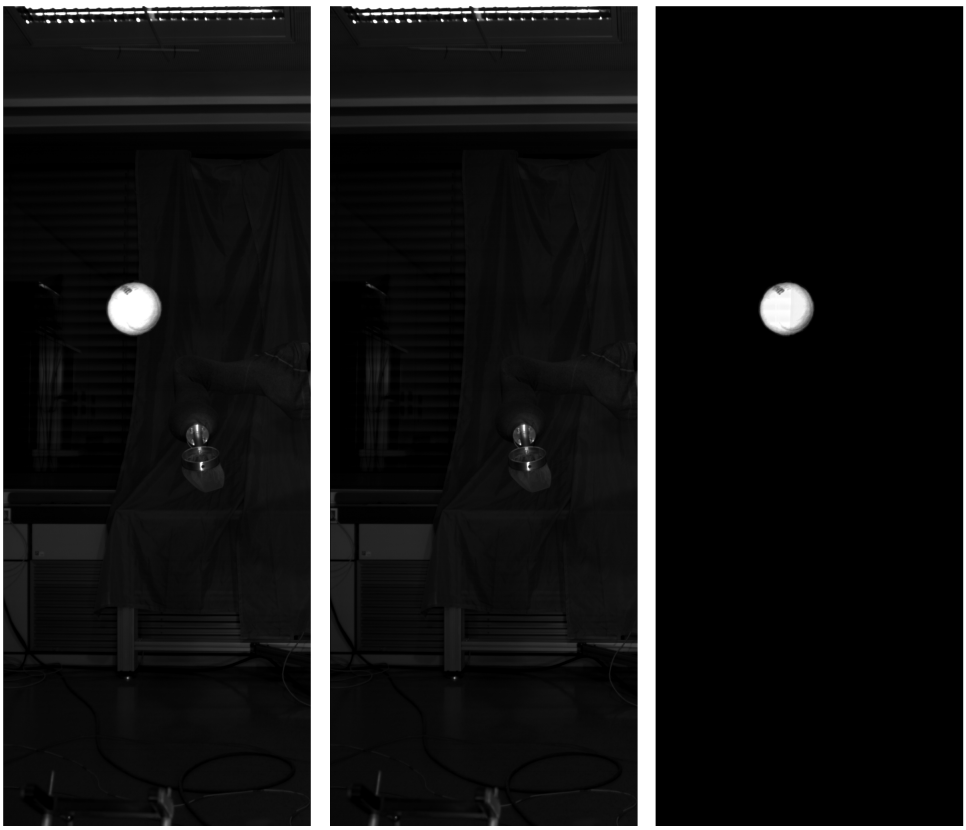

Abbildung 3.2: Hintergrundsubtraktion. Links das aktuell aufgenommene Bild mit dem Ball im Flug, in der Mitte das vorher aufgenommene Hintergrundbild ohne Ball, rechts das subtrahierte Bild.

Die ungefähre Bestimmung der Position vom Ballmittelpunkt wird, wie in Abschnitt 2.3 beschrieben, durch den Mittelwert aller Pixel welche zum Ball gehören berechnet. Dazu zählen alle Pixel mit einer Intensität höher als eine gewisse Schwelle. Dadurch werden etwaige Fehler in den Intensitäten ausgeschlossen und es werden nur Ballpixel gewertet. Ein auf diese Weise ermittelter Ballmittelpunkt kann trotzdem vom tatsächlichen Mittelpunkt abweichen und kann bei der Flugbahnprognose zu Abweichungen führen. Deshalb ist eine genauere Ballmittelpunktbestimmung notwendig. Um den ermittelten Ballmittelpunkt wird ein 300x300 Pixel großer Bereich definiert, in dem sich der Ball befindet. Dieser Bereich wird in der nachfolgenden Verarbeitungsstufe der *Objekterkennung zur Ballmittelpunktbestimmung* gebraucht, die sich aus dem subtrahierten Bild diesen 300x300 Pixel großen Bereich mit dem Ball herausschneidet.

3.3.4 Objekterkennung zur Ballmittelpunktbestimmung

Der Grund für die Benutzung von nur 300x300 Pixel großen Bildausschnitten, statt den Bildern in der großen Auflösung von 2048x768 Pixel, ist die benötigte Rechenleistung, die für das große Bild notwendig wäre um den exakten Ballmittelpunkt zu berechnen. Die hohe Auflösung bietet jedoch den Vorteil der besseren Ergebnisse (siehe Abschnitt 2.2). Durch das Ausschneiden des relevanten Bereiches, der sogenannten Area-of-Interest (AoI), aus dem großen Bild und die Erzeugung eines 300x300 Pixel Ausschnittes mit dem Ball, bleibt die Information von der großen Bildauflösung erhalten und gleichzeitig benötigt die Objekterkennung wesentlich weniger Rechenleistung, um den exakten Ballmittelpunkt zu berechnen. Abbildung 3.3 zeigt, dass es durch die x_{offset} und y_{offset} Werte möglich ist, wenn der Ballmittelpunkt im 300x300 Pixel Bild ermittelt wurde, den Ballmittelpunkt im großen 2048x768 Bild zu berechnen.

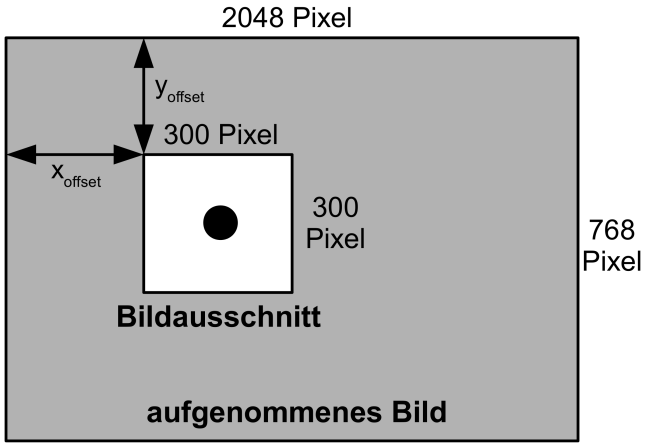

Abbildung 3.3: Ausschneidung des 300x300 Pixel Bildes mit der AoI mit dem Ball vom großen 2048x768 Pixel Bild. Die x_{offset} und y_{offset} Werte erlauben die Ballmittelpunktberechnung im großen Bild. Geändert von [Göt15, 37].

Optimal wäre, wenn die 300x300 Pixel Bilder mit dem Ball (AoI) direkt von der Kamera geliefert werden. Dies ist aber aufgrund von einem (dokumentierten) Treiber/Hardware-Problem der Kameras nicht optimal und es kommt zu Fehlern bei der Aufnahme. Die Verschiebung der AoI-Position zwischen zwei Bildaufnahmen ist nicht zuverlässig. Ein direktes Auslesen und

Verändern der AoI-Position wird dadurch verhindert. Deshalb wurde, als Umweg, das Ausschneiden des 300x300 Pixel Bildes aus dem 2048x768 Pixel Bild auf diesem Weg, per Software gelöst [Pon16, 103].

Diese Verarbeitungsstufe wird ausgeführt, sobald die vorherige Stufe mit der *Hintergrundsubtraktion* beendet ist. Mittels der x_{offset} und y_{offset} Werte wird nur der relevante Bereich (AoI) aus dem subtrahierten Bild der vorherigen Stufe ausgeschnitten und in diesem nach dem Ballmittelpunkt gesucht (siehe Abbildung 3.3).

Die Stufe mit der Ballmittelpunktermittlung wird hier nur grob beschrieben, da sie von einem Studenten am Institut für Computertechnik der Technischen Universität Wien, im Rahmen einer Master-Fachvertiefung, entworfen wurde [Lec16]. Der Algorithmus wurde speziell für die Implementierung auf einem FPGA ausgewählt und basiert auf der Arbeit, die bereits in Abschnitt 2.3, unter *Fast Circle Detection*, näher erläutert wurde. In den zu verarbeitenden Originalbildern ist immer nur ein Kreis vorhanden, deshalb reicht es aus, das Maximum, also die wahrscheinlichste Position vom Ball, zu ermitteln. In der Arbeit wurde der Ansatz mit den paarweisen Gradientenvektoren und die Hough Transformation miteinander verglichen und anhand der geforderten maximal möglichen Ressourcenausnutzung und der maximal möglichen Ausführungszeit entschieden, den Algorithmus der paarweisen Gradientenvektoren zu implementieren [Lec16, 13 f.].

Als erster Schritt wird auf das 300x300 Pixel Bild mit dem Ball der Mittelwertfilter für die Glättung angewendet und die Richtung der Gradientenvektoren mittels des Sobel Operators berechnet. Das Ergebnis wird in Abbildung 3.4 gezeigt. Dabei wird die Richtung durch die Helligkeit kodiert. Die schwarze Farbe entspricht 0 Grad und die weiße Farbe 359 Grad.

Abbildung 3.4: Links das Originalbild, rechts das Bild nach der Anwendung des Mittelwertfilters und der Berechnung der Richtung der Gradientenvektoren.

Der nächste Schritt ist die Suche nach den Paaren der Gradientenvektoren und die „Abstimmung“ über die möglichen Kandidaten der Ballmittelpunkte. Die Abbildung 3.5 zeigt das Ergebnis dieser Stufe im Vergleich mit dem Originalbild. Dabei gibt die Intensität des Pixels die Menge an „Stimmen“ an, die dieser Pixel bekommen hat. Je höher die Intensität, umso wahrscheinlicher ist dieser Pixel der gewählte Ballmittelpunkt.

Im letzten Schritt wird der Pixel mit der höchsten Intensität gesucht und für diesen die x und y Koordinaten in dem ursprünglichen 300x300 Pixel Bild angegeben. Diese Koordinaten gelten als der ermittelte Ballmittelpunkt. Laut der Simulation an mehreren Testbildern beträgt die Abweichung bei den ermittelten Ballmittelpunkten maximal fünf Pixel [Lec16, 16].

Abbildung 3.5: Links das Originalbild, rechts das Bild mit den „Stimmen“ für die möglichen Ballmittelpunkte. Je höher die Intensität vom Pixel, umso wahrscheinlicher ist dieser Pixel der Ballmittelpunkt.

3.3.5 Triangulation

Die dritte Verarbeitungsstufe dient der Berechnung der Position vom Ballmittelpunkt im dreidimensionalen Raum. Dazu werden die ermittelten Ballmittelpunkte von der vorherigen Verarbeitungsstufe, wie in Abbildung 3.3 dargestellt, in Koordinaten im ursprünglichen 2048x768 Pixel Bild von der Kamera zurück berechnet (jeweils für das linke und das rechte Bild). Damit diese Berechnung der räumlichen Position möglichst präzise ermittelt werden kann, müssen die Kameras zuvor kalibriert werden. Dieser Vorgang ist jedoch nicht Teil dieser Diplomarbeit und im Projektverlauf wurde mit bereits kalibriertem Kamerasystem gearbeitet. Ebenso sind alle benötigten intrinsischen Parameter und Matrizen, wie die Rodrigues Rotationsmatrix, bekannt.

Der erste Schritt ist die Normalisierung der Bilder in Abhängigkeit der intrinsischen Parameter. Dieses Verfahren eliminiert die Verzerrungen in den Bildern der Kameras und muss separat für das linke Bild und das rechte Bild durchgeführt werden. Durch die unterschiedliche Position der linken und der rechten Kamera sind die Parameter für das linke und das rechte Bild unterschiedlich und müssen auf das jeweils richtige Bild angewendet werden. Das Ergebnis dieses Prozederes sind die verzerrungsfreien Koordinaten der Bilder.
Mittels der Rodrigues Rotationsmatrix werden nachfolgend diese Koordinaten in Koordinaten im dreidimensionalen Raum umgerechnet. Man erhält sie jeweils für die Sicht von der linken Kamera und für die Sicht von der rechten Kamera.
Diese Koordinaten müssen, im letzten Schritt, in das Weltkoordinatensystem umgerechnet werden, in dem sich der Roboterarm befindet. Hierzu ist eine weitere Matrix notwendig, die sogenannte Transformationsmatrix, die ebenfalls bei der Kalibrierung definiert wird. Es werden nur die Koordinaten einer der Kameras in die Weltkoordinaten umgerechnet. Da sich die Kameras an unterschiedlichen Position befinden, müsste man für beide Kameras unterschiedliche Matrizen für die Umrechnung in die Weltkoordinaten aufstellen. Und da sich der Roboterarm immer nur an der selben Stelle im Raum befindet, würde das Ergebnis von beiden Kameras gleich aussehen, nur die Transformationsmatrix würde sich ändern. In der Praxis ist es üblich, immer die linke Kamera für diese Umrechnung zu benutzten. Daher werden die Koordinaten der rechten Kamera nicht weiter benötigt. Sobald sich aber die Positionen von den Kameras verändern, muss erneut eine Kalibrierung durchgeführt werden und alle Parameter erneut bestimmt werden, inklusive der Transformationsmatrix.

3.3.6 Flugbahnprognose

Im Anschluss an die Triangulation, kann die letzte Verarbeitungsstufe mit den Berechnungen beginnen. Hierbei handelt es sich um die *Flugbahnprognose*. In dieser Diplomarbeit wird die Prognose anhand des biologischen Ansatzes durchgeführt. Das bedeutet, dass auf eine Datenbank mit bereits gelerntem „Wissen“ zurückgegriffen wird und der aktuelle Flug des Balls wird mit den gelernten Flügen (Wissen) verglichen, um eine Vorhersage treffen zu können, wie sich der Ball im weiteren Flugverlauf bewegen wird. Wenn der weitere Flugverlauf bekannt ist, dann kann auf einfache Weise die Position ermittelt werden, an welcher der Roboterarm den Ball auffangen soll (siehe Abbildung 3.6).

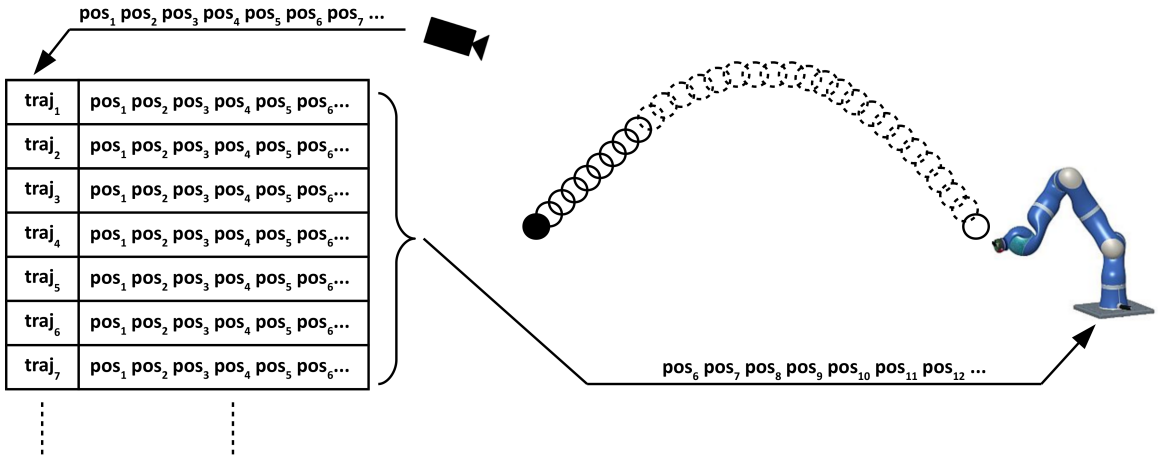

Abbildung 3.6: Der aktuelle Flug vom Ball wird mit bereits gelernten Flügen aus einer Datenbank verglichen. Anhand dieser Information kann der weitere Flugverlauf vorhergesagt und die Information dem Roboterarm übermittelt werden. Geändert von [Göt15, 63], [PPS12, 137].

Die Suche in dieser Datenbank mit gespeicherten Trajektorien wird mittels k-NN Algorithmus (k-Nearest Neighbors) durchgeführt, womit die k Trajektorien gefunden werden, die der aktuellen Trajektorie am ähnlichsten sind. Wenn k mit eins definiert wird, wird nur die eine Trajektorie ausgesucht, die der aktuellen Trajektorie am ähnlichsten ist. Je umfangreicher die Datenbank an Trajektorien ist, also je mehr Trajektorien abgespeichert sind, umso genauer wird die Vorhersage und die erzielten Ergebnisse [Göt15, 63].

Bei dieser Aufgabe, also der Suche nach ähnlichen Trajektorien aus einer großen Datenbank, handelt es sich wiederum um die Verarbeitung verschiedener Daten mit gleichen Instruktionen. Somit ist es Vernünftig, diese Aufgabe in dem FPGA zu implementieren.
Die Datenbank besteht aus 1024 Trajektorien. Zu jeder dieser Trajektorie sind die Koordinaten x,y,z und die Nummer vom zugehörigen Bild, zu welchem die Koordinaten gehören, abgespeichert. Es ist also in der Form eines dreidimensionalen Feldes aufgebaut. Den Ablauf der Flugbahnprognose kann man sich somit wie folgt vorstellen: das erste Bild wird aufgenommen, mittels der Triangulation wird die Position vom Ballmittelpunkt im Raum ermittelt (x,y,z Koordinaten) und diese Koordinaten werden mit den x,y,z Koordinaten vom ersten Bild mit den 1024 Trajektorien aus der Datenbank verglichen. Wird das zweite Bild aufgenommen, geschieht der Vergleich mit den Koordinaten vom zweiten und ersten Bild in der Datenbank usw. Insgesamt sind die Koordinaten x,y,z in der Datenbank für 100 Bilder abgespeichert. Das ist auch ausreichend, da der Ball

durchschnittlich nach etwa 85 Bildern bereits vom Roboterarm gefangen wurde. Da mit einer Bildrate von 100 Bilder pro Sekunde gearbeitet werden soll, resultiert daraus eine Flugzeit von etwa 850 *ms* (alle 10 *ms* wird ein Bild aufgenommen), gemessen vom Zeitpunkt an dem der Ball geworfen wurde, bis er vom Roboterarm gefangen wurde.

Es wird also die Trajektorie aus der Datenbank gesucht, die am ähnlichsten zu der aktuellen Trajektorie ist. Dazu wird der Algorithmus vom Euklidischer Abstand für dreidimensionale Koordinaten herangezogen (siehe Gleichung 3.3). Hier wird der Euklidischer Abstand zweier Punkte A(x,y,z) und B(x,y,z) mit jeweils den Koordinaten x,y,z berechnet.

$$d(A,B) = \sqrt{(x_\mathrm{A} - x_\mathrm{B})^2 + (y_\mathrm{A} - y_\mathrm{B})^2 + (z_\mathrm{A} - z_\mathrm{B})^2} \tag{3.3}$$

Angepasst an unsere Bedingungen, mit den Koordinaten vom aktuellen Flug x_{flug}, y_{flug} und z_{flug} und den zugehörigen Koordinaten aus der Datenbank x_{db}, y_{db} und z_{db}, resultiert die Gleichung 3.3 in die angepasste Gleichung 3.4.

$$d(flug, db) = \sqrt{(x_{\mathrm{flug}} - x_{\mathrm{db}})^2 + (y_{\mathrm{flug}} - y_{\mathrm{db}})^2 + (z_{\mathrm{flug}} - z_{\mathrm{db}})^2} \tag{3.4}$$

Die Gleichung 3.4 gibt also den Euklidischen Abstand der Koordinaten x,y,z vom aktuellen Flug, zu den Koordinaten x,y,z in der Datenbank für ein Bild (aus insgesamt 100) und für eine Trajektorie (aus insgesamt 1024), an.
Dieser Euklidischer Abstand muss natürlich noch für alle Bilder berechnet und zusammenaddiert werden. Sprich, beim ersten Bild wird der Abstand der Koordinaten vom aktuellen Flug und den Koordinaten in der Datenbank vom ersten Bild berechnet. Beim zweiten Bild wird der Abstand der Koordinaten vom aktuellen Flug und den Koordinaten in der Datenbank vom zweiten Bild berechnet und zum berechneten Abstand vom ersten Bild addiert. Beim dritten Bild wird der Abstand der Koordinaten vom aktuellen Flug und den Koordinaten in der Datenbank vom dritten Bild berechnet und zum berechneten Abstand vom ersten und zweiten Bild addiert usw. Diese Addition wird für jedes Bild durchgeführt, bei Bild 100 erfolgt also die Addition aller vorherigen 99 berechneten Abstände.

In Abbildung 3.7 ist schematisch der Vorgang der Addition der Euklidischer Abstände gezeigt. Zum Beispiel beim 4. Bild muss der berechnete Abstand für die Bilder 1 bis 4 berücksichtigt werden, damit die gesamte Trajektorie von der Position im 1. Bild bis zu dem aktuellen Bild berücksichtigt wird.

Die resultierende Gleichung für die Berechnung der Euklidischen Abstände für eine Trajektorie ist in der Gleichung 3.5 angegeben. Dabei gibt k das aktuelle Bild an, die Summe wird also von 1 bis k definiert, d. h. vom 1. Bild bis zum k-ten Bild.

$$d(flug, db) = \sum_{n=1}^{k} \sqrt{(x_{\mathrm{flug,n}} - x_{\mathrm{db,n}})^2 + (y_{\mathrm{flug,n}} - y_{\mathrm{db,n}})^2 + (z_{\mathrm{flug,n}} - z_{\mathrm{db,n}})^2} \tag{3.5}$$

Um die ähnlichste Trajektorie aus der Datenbank zu bestimmen, muss dieser Euklidischer Abstand mit allen Trajektorien in der Datenbank verglichen werden. Die Trajektorie, welche den geringsten Abstand aufweist, ist die ähnlichste zu dem aktuellen Flug und kann für die Vorhersage des weiteren Flugverlaufs herangezogen werden.

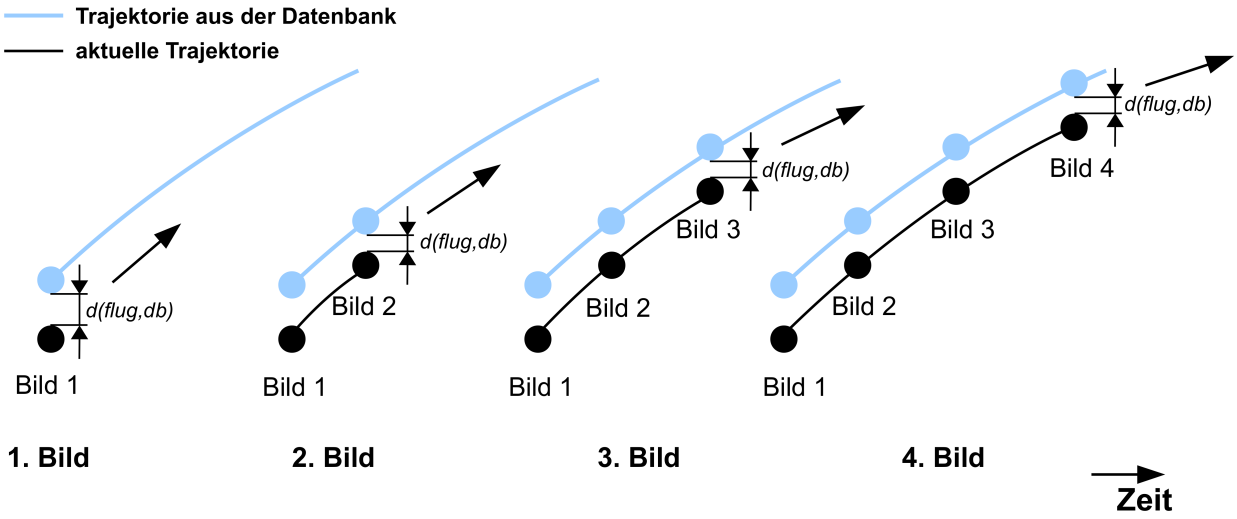

Abbildung 3.7: Schematische Darstellung der Addition der Euklidischer Abstände. Bereits beim 2. Bild muss zum Euklidischen Abstand vom 2. Bild auch der vom 1. Bild addiert werden. Beim 4. Bild müssen die Abstände der Bilder 1 bis 4 addiert werden usw.

3.4 Design der IP Cores

In diesem Abschnitt wird auf die einzelnen IP Cores eingegangen und die Designentscheidungen, welche getroffen wurden, näher erläutert. Alle diese IP Cores wurden mittels der High Level Synthese (HLS) entworfen, bei der der Algorithmus in der C Programmiersprache geschrieben wurde. Mittels der Programmierumgebung XILINX Vivado HLS wurde dieser anschließend in die Verilog bzw. Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (HDL), kurz VHDL, umgewandelt und synthetisiert. Der Begriff der Synthese beschreibt die Umwandlung der mittels HDL beschriebenen Schaltungen bzw. des beschriebenen Verhaltens der Schaltungen, in die Form, die auf einem bestimmten FPGA implementiert werden kann. Vivado HLS generiert für die erstellten IP Cores auch Treiber, welche unter Linux den Zugriff auch vom Userspace erlauben (siehe Abschnitt 3.3.2). Mit diesen Treibern ist es möglich, die IP Cores auf einfache Weise, direkt vom Hauptprogramm aus, zu steuern und Informationen auszulesen, sowie den IP Cores Daten zu übermitteln und Daten von diesen auszulesen.
Im Laufe dieses Abschnitts werden verschiedene Schnittstellen angesprochen. Deshalb wird hier kurz auf diese eingegangen und ihre wichtigsten Eigenschaften beschrieben. Während des Designs der IP Cores können die Schnittstellen zu und von einem IP Core, also die Eingänge und die Ausgänge, in verschiedenen Formen realisiert werden. Zum Beispiel als einfache Datenleitungen für Signale, die je nach dem verwendeten Datentyp von 1 bis 1024 bits breit sein können, oder als Schnittstelle zu RAM, oder Block RAM (BRAM), oder als eine der drei Ausführungen des Advanced eXtensively Interface (AXI) Bus. Der in dieser ZYNQ Plattform unterstützte AXI Bus ist von der Version 4. Man unterscheidet zwischen der AXI4 Full, AXI4 Lite und AXI4 Stream Schnittstelle [21, 24 ff.].

AXI4 Full

- hohe Leistung
- unterstützt Adressierung
- hoher Bedarf an Ressourcen
- kann Daten mittels Burst übertragen
- unterstützt Datenbreiten bis zu 1024 bits

In Vivado HLS wird diese Schnittstelle im sogenannten Master Modus unterstützt. Das bedeutet, der IP Core ist der Master und initiiert und kümmert sich um den Datentransfer von selbst, ohne, dass die CPU benötigt wird. Diese Schnittstelle kann entweder für individuellen Datentransfer verwendet werden, bei welchem für jedes Datenpaket eine Adresse angegeben werden muss und somit der Ablauf wie folgt aussieht: Adresse, Daten, Adresse, Daten usw. Oder es kann der Burst Modus für den Datentransfer verwendet werden, bei welchem nach einer Adresse direkt mehrere Datenpakete übertragen werden und somit höhere Datenraten erreichbar sind.
Diese Schnittstelle soll überall dort benutzt werden, wo viele und große Datenmengen über eine adressierbare Schnittstelle übertragen werden sollen.

AXI4 Lite

- geringe Leistung
- unterstützt Adressierung
- geringer Bedarf an Ressourcen
- keine Burst Übertragung
- unterstützt Datenbreiten bis zu 32 bits

In Vivado HLS wird diese Schnittstelle im sogenannten Slave Modus unterstützt. Das bedeutet, der IP Core ist der Slave und kann selbst keinen Datentransfer initiieren und überträgt Daten nur wenn der Master (CPU, oder Microkontroller) diese anfordert. Mittels dieser Schnittstelle kann der IP Core gesteuert werden und gibt Informationen über seinen Zustand an (idle, done, ready). Zusätzlich können dem IP Core über diese Schnittstelle Parameter übergegeben werden, die vom Algorithmus während der Ausführung berücksichtigt werden können.
Diese Schnittstelle soll überall dort benutzt werden, wo wenige und kleine Datenmengen über eine adressierbare Schnittstelle übertragen werden sollen.

AXI4 Stream

- hohe Leistung
- unterstützt keine Adressierung
- geringer Bedarf an Ressourcen
- unlimitierte Burst Übertragung
- Verbindung und Datentransfer vom Master zum Slave

Diese Schnittstelle ist nur für direkte Verbindungen zwischen zwei IP Cores geeignet. Die Daten werden sequentiell im Stream übertragen. Der Vorteil dieser Schnittstelle liegt im erreichbaren Datendurchsatz, da im Vergleich zu den adressierbaren Schnittstellen AXI4 Full und AXI4 Lite, diese Schnittstelle keinen Adressierungsoverhead aufweist. Daten werden direkt vom Master zum Slave übertragen und die Aushandlung der Übertragung erfolgt auf einfache Weise mittels zwei Datenleitungen *TREADY* und *TVALID*. Mit *TVALID* signalisiert der Master, dass die Daten auf der Datenleitung gültig sind, mit *TREADY* signalisiert der Slave, dass er bereit ist Daten zu empfangen. Für die Datenübertragung müssen also beide Leitungen auf logisch Eins sein. Der Nachteil dieser Schnittstelle liegt in der Notwendigkeit, dass für eine Datenübertragung, der Master als auch der Slave bereit sein müssen. Das ist natürlich auch bei anderen Formen der Datenübertragung notwendig, wie zum Beispiel beim Speichern in den RAM. Für die Daten muss immer eine Senke vorhanden sein. Der Unterschied ist aber, dass normalerweise IP Cores unabhängig voneinander arbeiten sollen. Wenn eine AXI4 Stream Schnittstelle zwischen den IP Cores verwendet wird, sind die IP Cores voneinander abhängig und können nicht jeder für sich selbst arbeiten.

3.4.1 Hintergrundsubtraktion

Die folgende Abbildung 3.8 zeigt den Block mit den Schnittstellen des IP Cores für die *Hintergrundsubtraktion*, wie er mittels HLS erstellt wurde.

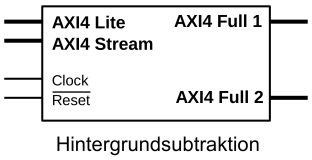

Abbildung 3.8: Block des erstellten IP Cores für die Hintergrundsubtraktion.

Die Funktion dieses IP Cores wurde bereits in Abschnitt 3.3.3 beschrieben. Der IP Core subtrahiert das empfange Bild von der Kamera mit einer Auflösung von 2048x768 Pixel (Bild mit Ball) und ein vorher aufgenommenes Bild vom Hintergrund ebenfalls mit einer Auflösung von 2048x768 Pixel (Bild ohne Ball). Diese beiden Bilder sind die Eingangsdaten für den IP Core. Das Ergebnis der Subtraktion (Bild mit 2048x768 Pixel) sind die Ausgangsdaten des IP Cores. Zusätzlich wird die ungefähre Position vom Ballmittelpunkt ermittelt und ein 300x300 Pixel großer Bereich, wo sich der Ball befindet, definiert. Die x und y Koordinaten dieses Bereichs (siehe Abbildung 3.3) sind die weiteren Ausgangsdaten des IP Cores.

Da das Hintergrundbild statisch ist und es nur am Anfang des Hauptprogramms einmal aufgenommen wird und für jedes aufgenommene Bild von der Kamera das gleiche Hintergrundbild für die Subtraktion verwendet wird, wäre es am effektivsten, das Hintergrundbild im BRAM vom FPGA zu speichern. Das BRAM ist nämlich direkt im FPGA integriert und stellt einen sehr schnellen Speicher für die implementierte Logik dar. Leider ist dieser BRAM, auch in dem leistungsstärkstem ZYNQ, der in dieser Diplomarbeit verwendet wurde, nicht groß genug, um das Hintergrundbild von der linken und der rechten Kamera auf diese Weise zu speichern. Die Größe vom verfügbaren BRAM liegt bei etwa 3 MB (siehe auch Abschnitt 2.2). Ein Bild mit 2048x768

Pixel, bei 8 bit, hat die Größe von 1,5 MB. Zwei solcher Bilder, für die linke und die rechte Kamera, haben somit eine Größe von 3 MB. Das würde sich wahrscheinlich mit dem verfügbaren BRAM ausgehen, aber nur, falls die andere integrierte Logik keine BRAMs benötigt. Das ist aber nicht der Fall, da BRAM sehr häufig von unterschiedlichen IP Cores für die Speicherung von Daten benutzt wird, auf die ein schneller Zugriff benötigt wird. Zusätzlich wird der BRAM auch von weiteren, in dieser Diplomarbeit benutzten IP Cores, benötigt. Es muss also auf eine andere Möglichkeit ausgewichen werden, wo die Hintergrundbilder gespeichert werden können.

Eine weitere Möglichkeit ist die Benutzung von einem DMA, der die Daten vom Hintergrundbild aus dem RAM lädt und über eine AXI4 Stream Schnittstelle dem IP Core per Stream sendet. Dafür muss ein DMA im Design implementiert werden. XILINX bietet verschiedene hoch konfigurierbare IP Cores an, die im Design implementiert werden können, unter anderen auch einen DMA IP Core.
Es stellt sich noch die Frage, wie werden die Bilder von der Kamera (mit dem Ball) dem IP Core übergegeben? Da diese vom Treiber für die Kameras im Speicher (RAM) bereitgestellt werden, liegt dementsprechend nahe, auch in diesem Fall einen DMA zu verwenden. Somit würden sowohl das Hintergrundbild als auch das aktuell aufgenommene Bild mittels DMA aus dem RAM geladen und über eine AXI4 Stream Schnittstelle dem IP Core übergegeben. Der Nachteil dieser Lösung ist die Implementierung von zwei DMAs, die die Daten vom selben RAM laden. Es muss nicht nur über die Steuerung der beiden DMAs gekümmert werden, sondern auch der Datenzugriff zweier DMAs zum selben RAM gleichzeitig kann zu Problemen führen. Da der IP Core für die *Hintergrundsubtraktion* gleich doppelt im FPGA implementiert wird (siehe Abschnitt 3.5), jeweils einer für jede Kamera, müssten die DMAs gleich vierfach implementiert werden. Das würde die Probleme mit der Steuerung und dem gleichzeitigen Zugriff auf den RAM wahrscheinlich noch verstärken.

Eine weitere Möglichkeit, die auch schlussendlich implementiert wurde, ist die Benutzung der AXI4 Full Schnittstelle für das Laden der Hintergrundbilder. Für den Datentransfer kümmert sich somit der IP Core selbst und benötigt für die Hintergrundbilder keinen DMA. Ein DMA per IP Core wird trotzdem benötigt für den Datentransfer der aktuell aufgenommenen Bilder von der Kamera. Diese werden von dem DMA aus dem RAM geladen und über eine AXI4 Stream Schnittstelle zum IP Core übertragen. Der RAM, in welchem die Hintergrundbilder gespeichert werden, ist ein anderer, als in welchem der Treiber für die Kameras die aktuell aufgenommenen Bilder bereitstellt. Auf der Entwicklungsplattform sind insgesamt 2 GB an RAM verbaut (siehe Abschnitt 2.2), aber diese sind unterschiedlich aufgeteilt und an unterschiedliche Teile des SoCs angebunden. 1 GB an RAM sind direkt an die CPUs angebunden, processing system RAM (PS RAM) und die restlichen 1 GB RAM sind direkt an dem FPGA angebunden, programmable logic RAM (PL RAM). Die Speicheranbindung des PS RAM ist 32-bit breit, die des PL RAM ist 64-bit breit und bietet somit eine höhere Bandbreite.
Die aktuellen Bilder von der Kamera werden aus dem PS RAM und die Hintergrundbilder aus dem PL RAM bereitgestellt. Dadurch eliminiert sich das mögliche Problem mit dem gleichzeitigen Zugriff zum selben RAM. Der PL RAM wird außer für die Speicherung der Hintergrundbilder auch als Zwischenspeicher für die subtrahierten Bilder von beiden IP Cores verwendet.

Das Bild von der Kamera mit dem Ball und das aufgenommene Hintergrundbild werden somit über unterschiedliche Schnittstellen dem IP Core zur Verfügung gestellt. Die *AXI4 Stream Schnittstelle* wird für die aktuell aufgenommenen Bilder von der Kamera verwendet. Ein DMA kümmert sich um die Bereitstellung der Daten als Stream. Die *AXI4 Full 1 Schnittstelle* dient

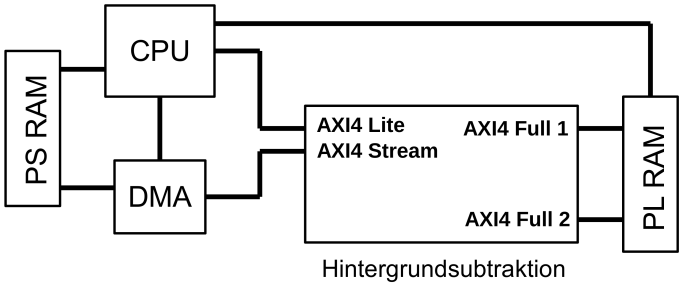

Abbildung 3.9: Vereinfachte Darstellung der Anbindung des IP Cores für die Hintergrundsubtraktion an die CPU, DMA und die beiden RAM Blöcke.

zum Laden des Hintergrundbildes aus dem PL RAM, die *AXI4 Full 2 Schnittstelle* zum Speichern des subtrahierten Bildes im PL RAM. Der Datentransfer über die beiden AXI4 Full Schnittstellen geschieht im Burst Modus um den schnellstmöglichen Transfer zu erreichen. Zur Steuerung und Bereitstellung der Informationen über den IP Core wird die *AXI4 Lite Schnittstelle* verwendet. Über diese werden auch die x und y Koordinaten des relevanten Bereiches mit dem Ball bereitgestellt. Die Steuerung, das Laden der Informationen und der Ergebnisse in Form von x und y Koordinaten geschieht über eine Adresse des IP Cores im erreichbaren Speicherbereich der CPU. Die hier erwähnten Schnittstellen beziehen sich auf die Bezeichnungen in Abbildungen 3.8 und 3.9. In Abbildung 3.9 ist auch die vereinfachte Anbindung des IP Cores an das System dargestellt.

3.4.2 Objekterkennung zur Ballmittelpunktbestimmung

Die folgende Abbildung 3.10 zeigt den Block mit den Schnittstellen des IP Cores für die *Objekterkennung zur Ballmittelpunktbestimmung*, wie er mittels HLS erstellt wurde.

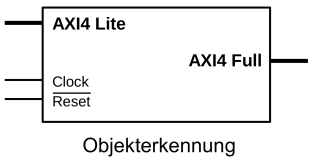

Abbildung 3.10: Block des erstellten IP Cores für die Objekterkennung zur Ballmittelpunktbestimmung.

Die Funktion dieses IP Cores wurde bereits in Abschnitt 3.3.4 beschrieben. Der IP Core berechnet den exakten Ballmittelpunkt in einem 300x300 Pixel Bild. Zuvor muss dieser 300x300 Pixel Bereich aus einem 2048x768 Pixel Bild ausgeschnitten werden. Dazu bekommt der IP Core die x und y Koordinaten dieses 300x300 Pixel Bereiches in dem großen 2048x768 Pixel Bild (siehe Abbildung 3.3). Die Eingangsdaten des IP Cores sind somit das 2048x768 Pixel Bild von der vorherigen Verarbeitungsstufe *Hintergrundsubtraktion* und die x und y Koordinaten des 300x300 Pixel Bereiches. Das Ergebnis der Berechnungen in diesem IP Core ist der Ballmittelpunkt des erkannten Balls in Form von x und y Koordinaten des Pixels, wo sich der Ballmittelpunkt befindet. Diese Koordinaten beziehen sich auf das 300x300 Pixel Bild und nicht auf das ursprüngliche 2048x768 Pixel Bild. Um die Koordinaten im großen 2048x768 Pixel Bild zu bestimmen, müssen

die ermittelten Koordinaten des Ballmittelpunktes im 300x300 Pixel Bild zu den x und y Koordinaten, die in der vorherigen Verarbeitungsstufe ermittelt wurden, addiert werden (siehe auch Abschnitt 3.3.5).

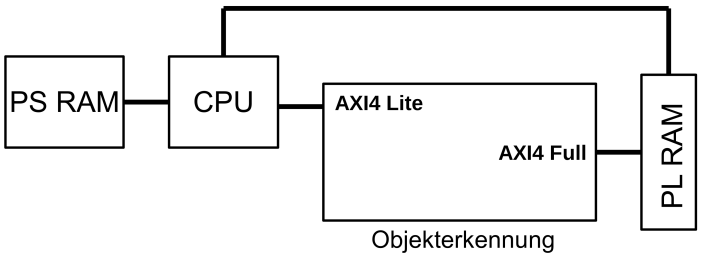

Abbildung 3.11: Vereinfachte Darstellung der Anbindung des IP Cores für die Objekterkennung zur Ballmittelpunktbestimmung an die CPU und den PL RAM Block.

Die *AXI4 Full Schnittstelle* dient zum Laden des subtrahierten Bildes der vorherigen Verarbeitungsstufe aus dem PL RAM. Zur Steuerung und Bereitstellung der Informationen über den IP Core wird die *AXI4 Lite Schnittstelle* verwendet. Über diese Schnittstelle werden sowohl die x und y Koordinaten des 300x300 Pixel Bereiches empfangen, als auch die x und y Koordinaten des ermittelten Ballmittelpunktes bereitgestellt. Dieser IP Core hat wiederum eine eigene Adresse im erreichbaren Speicherbereich der CPU und kann somit über diese gesteuert und mit Daten versorgt bzw. die Daten werden über diese Schnittstelle bereitgestellt.
Die hier erwähnten Schnittstellen beziehen sich auf die Bezeichnungen in Abbildungen 3.10 und 3.11. In Abbildung 3.11 ist auch die vereinfachte Anbindung des IP Cores an das System dargestellt.

Der Vorgang zum Ausschnitt des 300x300 Pixel Bildes aus dem großen 2048x768 Pixel Bild geschieht auf eine einfache Weise. In dem PL RAM ist das große 2048x768 Pixel Bild als Ergebnis der vorherigen Verarbeitungsstufe abgespeichert. Durch die x und y Koordinaten des relevanten Bereiches kann nur das 300x300 Pixel Bild ausgelesen werden. Dies ist möglich, da das Bild im RAM in einem zusammenhängenden Speicherbereich gespeichert ist und auf die einzelnen Pixel separat zugegriffen werden kann. Damit wird sichergestellt, dass nur die Daten übertragen werden, die auch notwendig sind.

3.4.3 Flugbahnprognose

Die folgende Abbildung 3.12 zeigt den Block mit den Schnittstellen des IP Cores für die *Flugbahnprognose*, wie er mittels HLS erstellt wurde.

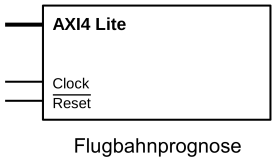

Abbildung 3.12: Block des erstellten IP Cores für die Flugbahnprognose.

Die Funktion dieses IP Cores wurde bereits in Abschnitt 3.3.6 beschrieben. Der IP Core sucht in einer vorher definierten Datenbank an Trajektorien, die eine Trajektorie, welche anhand der Kriterien am ähnlichsten ist. Diese Kriterien sind die aktuellen Koordinaten x,y,z vom Ballmittelpunkt im Raum und die Bildfolgenummer (1-100) des aktuell aufgenommenen Bildes von den Kameras. Für die Suche wird der k-NN Algorithmus benutzt, wobei k = 1 definiert ist. Die Eingangsdaten sind somit die Koordinaten x,y,z und die Bildnummer, zu der diese Koordinaten dazugehören. Das Ergebnis ist die Nummer der Trajektorie, die am ähnlichsten zu diesen (und für Bildfolgenummern ≥ 2 auch der vorherigen) Koordinaten ist, d. h. bei der die Summe über alle bereits aufgenommene Koordinaten den geringsten Euklidischen Abstand aufweist.

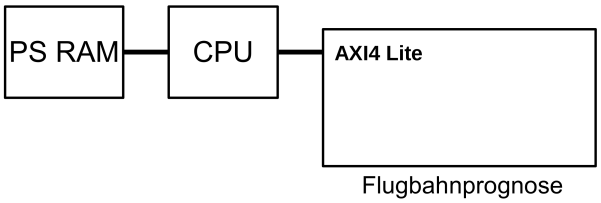

Abbildung 3.13: Vereinfachte Darstellung der Anbindung des IP Cores für die Flugbahnprognose an die CPU.

Bei diesem IP Core entstehen keine großen zu übertragenden Datenmengen und es reicht also vollkommen aus eine *AXI4 Lite Schnittstelle* für alle Parameter zu verwenden. Über diese Schnittstelle wird der IP Core gesteuert und es werden die x,y,z Koordinaten im Raum, als auch die Bildnummer dem IP Core übermittelt. Das Ergebnis in Form von der Nummer der ermittelten Trajektorie wird ebenfalls über diese Schnittstelle bereitgestellt. Wie jeder IP Core, der die *AXI4 Lite Schnittstelle* benutzt, hat auch dieser eine Adresse im erreichbaren Speicherbereich der CPU und kann über diese die Daten empfangen und senden.
Die hier erwähnte Schnittstelle bezieht sich auf die Bezeichnung in Abbildungen 3.12 und 3.13. In Abbildung 3.13 ist auch die vereinfachte Anbindung des IP Cores an das System dargestellt.

Dieser IP Core wurde als letzter entworfen und in das fertige Design implementiert. Da im Design schon zwei IP Cores für die *Hintergrundsubtraktion* und zwei IP Cores für die *Objekterkennung zur Ballmittelpunktbestimmung* implementiert waren (siehe Abschnitt 3.5), gab es einige Schwierigkeiten mit der Umsetzung von diesem IP Core für die *Flugbahnprognose*. Der Grund war, die bereits zu dem Zeitpunkt hohe Ausnutzung der im FPGA verfügbaren Ressourcen aufgrund der im FPGA implementierten Logik. Die Datenbank an Trajektorien ist mit ihren 1024 Einträgen, für 100 Bilder und den x,y,z Koordinaten relativ umfangreich. Es werden somit $1024 \times 100 \times 3 = 307200$ Gleitkommazahlen abgespeichert.

Eine Möglichkeit, wo diese Datenbank gespeichert werden könnte, ist der PL RAM und die Verwendung einer AXI4 Full Schnittstelle, um auf die Einträge der Datenbank zuzugreifen. Der Vorteil dieser Lösung wäre, dass der IP Core nur wenige Ressourcen benötigt, da mit den bereits implementierten IP Cores, nur mehr etwa 40% der Ressourcen im FPGA verfügbar waren. Ein weiterer Vorteil wäre die einfache Erweiterung der Datenbank um zusätzliche Trajektorien, ohne dass der verfügbare Speicherplatz ein limitierender Faktor wäre. Würde man den IP Core so entwerfen, dass auch die Anzahl der Trajektorien mittels eines Parameters konfigurierbar wäre, dann müsste man auch den IP Core nicht jedesmal neu synthetisieren, wenn die Datenbank erweitert wurde. Der Nachteil von nicht festen Parametern, wie zum Beispiel die variable Anzahl an Tra-

jektorien ist, dass HLS während der Synthese nicht die optimale Logik entwerfen kann und somit eventuell der IP Core nicht die benötigte Geschwindigkeit erreicht. Beim Entwurf gilt allgemein, dass je mehr Parameter während der Synthese bekannt sind, umso optimierte und schnellere Logik kann entworfen werden. Parameter, welche dann nachträglich vom Programm aus geändert werden können, können den Entwurf von optimaler und schneller Logik verhindern. Ein Nachteil diese Ansatzes wäre auch der weitere Datenzugriff zusätzlich von diesem IP Core aus. Der PL Speicher und die Datenanbindung wird bereits für häufige Datentransfers von den anderen IP Cores benutzt und es werden dabei große Datenmengen übertragen. Zusätzlicher Datenzugriff auch von diesem IP Core, könnte zu Datenengpässen führen, vor allem, wenn die Datentransfers gleichzeitig stattfinden sollen.

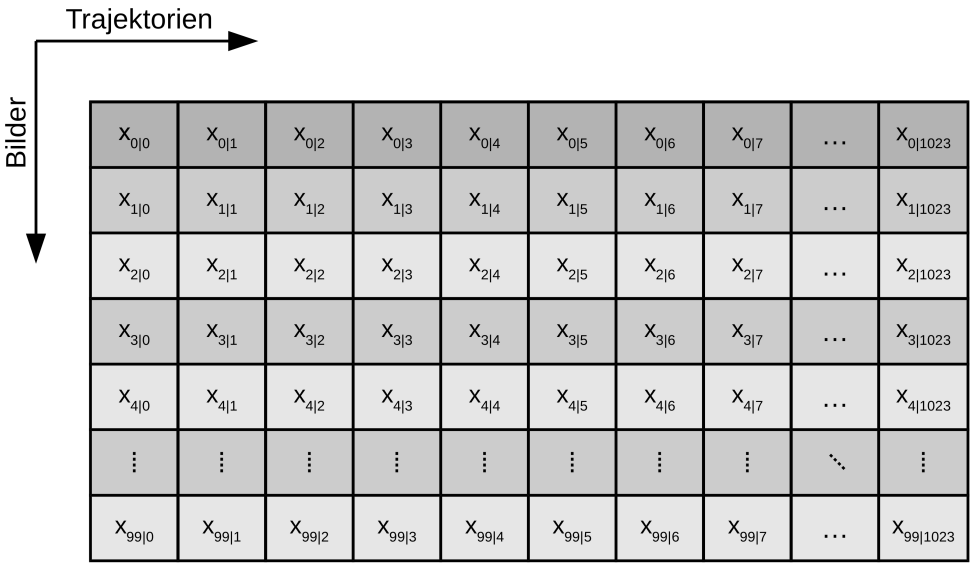

Abbildung 3.14: Darstellung des zweidimensionalen Feldes der Datenbank mit den gespeicherten Trajektorien zu jedem aufgenommenen Bild, hier für die x Koordinate der Ballposition im Raum.

Eine weitere Möglichkeit, die auch schlussendlich implementiert wurde, ist die Speicherung der Datenbank auf dem FPGA mittels BRAM und Look-Up Table (LUT). Die Datenbank wurde anhand der Koordinaten x,y,z in drei Teile aufgeteilt, wobei jeder Teil aus einem zweidimensionalen Feld mit 1024 Trajektorien und 100 Bildern besteht. Die Abbildung 3.14 zeigt den Aufbau des Feldes für die x Koordinate. Die Felder für die y und z Koordinaten sind analog aufgebaut. Es wurde versucht alle drei Teile (Koordinaten x,y,z) mittels BRAM im FPGA zu implementieren, also die Einträge der Datenbank im BRAM zu speichern. Dies war nicht möglich, da die BRAM Ausnutzung im FPGA bereits zu hoch war. Somit wurden zwei der drei Teile (x und y) als BRAM und der letzte dritte Teil (z) der Datenbank als LUT implementiert. Auf diese Weise konnte die gesamte Datenbank an Trajektorien im FPGA implementiert werden. Der Vorteil dieses Ansatzes ist, dass die gesamte Kommunikation nur innerhalb des FPGAs passiert und kein zusätzlicher Datentransfer zum PS oder PL RAM notwendig ist und daher werden auch die Buszugriffe durch diesen IP Core nicht erhöht. Zusätzlich, da die Kommunikation nur innerhalb des FPGAs stattfindet und für die Speicherung der Daten BRAM und LUT verwendet wird, ist der Datentransfer sehr schnell. Ein Nachteil dieser Lösung ist die Notwendigkeit, bei einer Änderung der Datenbank,

den IP Core neu zu synthetisieren, was bei dem sehr großen Design des Gesamtsystems mehrere Stunden in Anspruch nimmt. Zusätzlich ist man durch bereits relativ wenige freie Ressourcen an BRAM und LUT im FPGA, vom Speicherplatz her für neue Datenbankeinträge limitiert. Durch die hohe Ressourcenausnutzung sind auch die Möglichkeiten der Implementierung zusätzlicher Logik, für eventuelle weitere Aufgaben im FPGA, beschränkt.
Da aber dieser IP Core die letzte Aufgabe für dieses Projekt und im Rahmen dieser Diplomarbeit darstellt, wird keine weitere Logik im FPGA implementiert und die Menge an verfügbaren Ressourcen ist somit für die gestellte Aufgabe ausreichend.

3.5 Design des Gesamtsystems

In diesem Abschnitt werden die Designentscheidungen beim Entwurf des Gesamtsystems beschrieben. Die in vorherigem Abschnitt 3.4 gezeigten IP Cores werden in diesem Abschnitt in das Gesamtsystem integriert.

Wie in Abschnitt 2.2 und 3.1 erwähnt wurde, soll das System mit einer Bildrate von 100 Bilder pro Sekunde arbeiten. Das bedeutet, berechnet mit Hilfe der Gleichung 2.1, alle 10 *ms* wird von den beiden Kameras ein neues Bild geliefert.
Die erste Verarbeitungsstufe, bei der das Bild der Kamera ankommt, ist der IP Core für die *Hintergrundsubtraktion* beschrieben in Abschnitt 3.3.3 und 3.4.1. Dieser muss also maximal 10 *ms* für die Verarbeitung der Daten benötigen, weil 10 *ms* nachdem das erste Bild an dem IP Core ankommt, bereits das nächste Bild zum IP Core gesendet wird. Die Logik von dem IP Core muss mit der Verarbeitung fertig sein und bereit sein, neue Daten in Form von einem neuen Bild zu akzeptieren. Wäre das nicht der Fall und der IP Core würde länger als die geforderten 10 *ms* zur Verarbeitung der Daten benötigen, dann könnte das System nicht mit der Bildrate von 100 Bilder pro Sekunde arbeiten, wodurch wiederum beispielsweise die Flugbahnprognose nicht die benötigten Ergebnisse und Genauigkeit für den Transport durch Werfen und Fangen erreichen könnte. Es ist also absolut notwendig, die geforderte Bildrate von 100 Bilder pro Sekunde zu erreichen.
Die nächste Verarbeitungsstufe, *Objekterkennung zur Ballmittelpunktbestimmung* beschrieben in Abschnitt 3.3.4 und 3.4.2, muss ebenfalls mit der Verarbeitung der Daten innerhalb von 10 *ms* fertig sein, genauso die *Flugbahnprognose*, beschrieben in Abschnitt 3.3.6 und 3.4.3. Die Verzögerung dieser Verarbeitungspipeline, bestehend aus diesen drei IP Cores, ist in der Größenordnung von 30 bis 40 *ms*. Darin ist auch die Verarbeitungszeit von der Triangulation, beschrieben in Abschnitt 3.3.5, miteinbezogen worden. Diese wird zwischen dem IP Core für die *Objekterkennung zur Ballmittelpunktbestimmung* und dem IP Core für die *Flugbahnprognose* in der Software ausgeführt (siehe Abbildung 3.15). 30 bis 40 *ms* ist die Zeit, von dem Zeitpunkt an dem das Bild an dem ersten IP Core ankommt, bis zum Zeitpunkt wo die Information über die Trajektorie passend zu dem Bild, geliefert wird. Wichtig in diesem Zusammenhang ist, dass der erste IP Core alle maximal 10 *ms* neue Daten entgegennimmt und der letzte IP Core alle maximal 10 *ms* neue Daten ausgibt.

An dieser Stelle sei erwähnt, dass aus Einfachheits- und auch aus Leistungsgründen entschieden wurde, dass separate IP Cores die Bilder von der linken Kamera und die Bilder von der rechten Kamera verarbeiten werden. Das bedeutet, die IP Cores für die *Hintergrundsubtraktion* und *Objekterkennung zur Ballmittelpunktbestimmung* werden im FPGA zwei Mal implementiert. Würde jeweils nur ein IP Core implementiert, dann müsste die Verarbeitung der Daten seriell stattfinden, also zuerst das linke Bild und anschließend erst das rechte Bild. Bei diesem Ansatz

könnte es zu Leistungsproblemen kommen, da beispielsweise der erste IP Core innerhalb von 5 *ms* mit der Verarbeitung des ersten linken Bildes fertig sein müsste. Denn in den folgenden 5 *ms* würde er schon mit dem ersten rechten Bild beschäftigt sein, damit die Deadline von 10 *ms* eingehalten werden kann.
Da die FPGA Plattform dafür ausgelegt ist, Aufgaben zu parallelisieren, kann für die Bilder von der linken Kamera und für die Bilder von der rechten Kamera jeweils ein eigener IP Core zuständig sein, sofern es die Ressourcen im FPGA zulassen. Die Szene mit dem Wurf wird schließlich auch von zwei Kameras beobachtet, die die Bilder parallel aufnehmen, also liegt die Entscheidung nahe, auch die Daten parallel im FPGA zu verarbeiten.

Die Abbildung 3.15 veranschaulicht schematisch diesen Vorgang der parallelen Verarbeitung der Bilder in den einzelnen IP Cores.

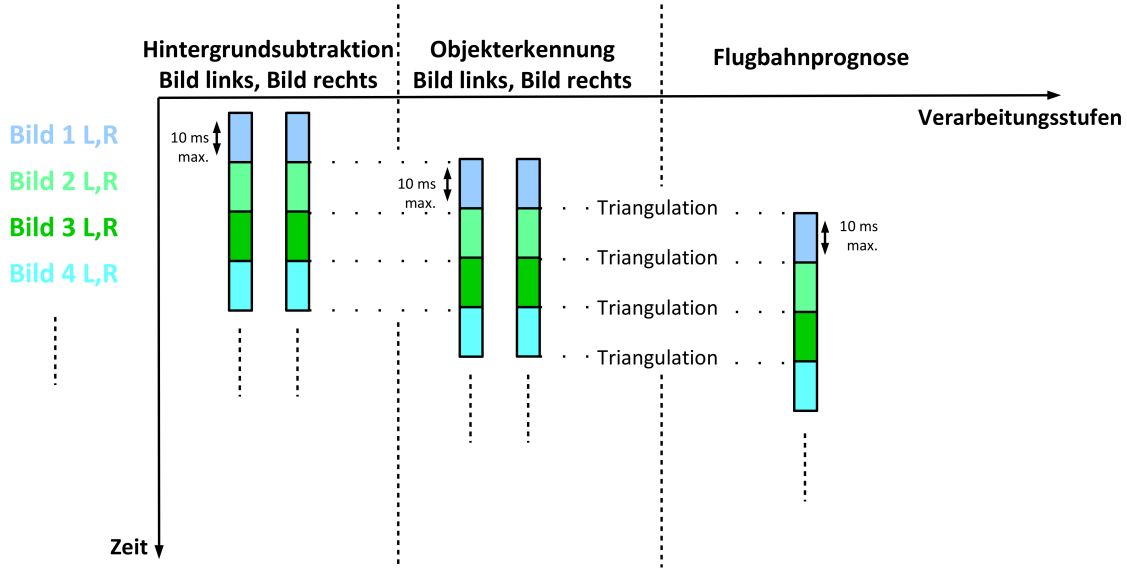

Abbildung 3.15: Schematische Darstellung der parallelen Verarbeitung der Bilder in den IP Cores.

Als erstes wird Bild 1 von der linken und der rechten Kamera in den ersten IP Cores mit der *Hintergrundsubtraktion* verarbeitet. Das Ergebnis der ersten IP Cores wird den zweiten IP Cores mit der *Objekterkennung zur Ballmittelpunktbestimmung* übergegeben und dort verarbeitet. Währen die zweiten IP Cores mit der *Objekterkennung zur Ballmittelpunktbestimmung* von den beiden ersten Bildern beschäftigt sind, verarbeiten die ersten beiden IP Cores mit der *Hintergrundsubtraktion* bereits das zweite Paar an Bildern. Ist die Objekterkennung von den ersten beiden Bildern fertig, wird die Triangulation durchgeführt und anschließend der letzte IP Core mit der *Flugbahnprognose* gestartet. Während die *Flugbahnprognose* die Daten von den ersten beiden Bildern verarbeitet, werden in den zweiten IP Cores mit der *Objekterkennung zur Ballmittelpunktbestimmung* die Daten vom zweiten Paar an Bildern verarbeitet und in den ersten IP Cores mit der *Hintergrundsubtraktion* wird das dritte Paar an Bildern verarbeitet usw. Somit sind die Daten zwischen der ersten und der letzten Verarbeitungsstufe, um die benötigte Verarbeitungszeit in den einzelnen Verarbeitungsstufen, verzögert.

Außer den bisher angesprochenen drei IP Cores sind einige weitere im Gesamtsystem implementiert. Wie bereits in Abschnitt 3.3.1 erwähnt wurde, erfolgt die Anbindung der Kameras über

eine USB 3.0 Erweiterungskarte für die PCIe Schnittstelle. Um diese Schnittstelle an das System anbinden zu können, muss ein dafür konzipierter IP Core im FPGA implementiert werden. Dieser kümmert sich anschließend um den Datentransfer zwischen der PCIe Schnittstelle und dem PS.
Weiters sind die beiden DMA IP Cores, die als Schnittstelle zwischen dem PS und den ersten beiden IP Cores für die *Hintergrundsubtraktion* dienen, implementiert. Mittels der DMAs werden die aufgenommenen Bilder, welche vom Kameratreiber im RAM zur Verfügung gestellt werden, aus dem RAM in einen Stream umgewandelt und zu den beiden IP Cores gesendet.
Zusätzlich ist ein Memory Controller, der als Schnittstelle zwischen der PL und dem PL RAM dient, als IP Core implementiert. Über diesen greifen die IP Cores im FPGA auf PL RAM zu. Dazu gehören einerseits die beiden IP Cores für die *Hintergrundsubtraktion*, die sowohl das Hintergrundbild von der linken und der rechten Kamera aus dem PL RAM laden und auch das Ergebnis in Form von einem subtrahierten Bild in diesem RAM abspeichern. Andererseits wird der Memory Controller auch von den IP Cores für die *Objekterkennung zur Ballmittelpunktbestimmung*, die das 300x300 Pixel Bild aus dem PL RAM laden, benutzt. Weiters wird auch vom Hauptprogramm aus auf den PL RAM zugegriffen um einmalig, am Anfang, die beiden Hintergrundbilder in diesem RAM abzuspeichern. Die restlichen implementierten IP Cores dienen der Steuerung und der Zusammenschaltung aller IP Cores zu einem Gesamtsystem.
Der ZYNQ SoC verfügt über vier High Performance (HP) Schnittstellen, die für den Austausch von Daten zwischen dem PS und der PL verwendet werden können [23, 128 ff.]. Im ZYNQ Blockdiagramm, in Abbildung 2.2, sind diese rot eingekreist. Diese können nur im Slave Modus verwendet werden, somit müssen im PL die IP Cores im Master Modus betrieben werden. Daten, die über diese HP Schnittstellen übertragen werden, haben direkte hochleistungsfähige Verbindung zum PS RAM. Um eventuelle Leistungsengpässe bei der Datenübertragung zu vermeiden, wurden die IP Cores von der PCIe Schnittstelle und die beiden DMAs an separate HP Schnittstellen angeschlossen. Eine andere Möglichkeit wäre die Verwendung eines AXI Interconnects, der quasi als ein Multiplexer funktioniert und mehrere Master mit einer Slave Schnittstelle verbindet. Da aber bis zu vier separate HP Schnittstellen vorhanden sind, kann für jeden Master im FPGA (ein PCIe, zwei DMAs) eine eigene HP Slave Schnittstelle verwendet werden.

Hauptprogramm

Für die Interaktion mit der SoC Plattform und mit dem Linux Betriebssystem ist eine Kommandozeile ausreichend. Die Plattform bietet eine Bildausgabe über die High-Definition Multimedia Interface (HDMI) Schnittstelle an, diese wurde aber nicht benutzt, da dafür weitere Logik im FPGA implementiert werden müsste. Da nur mit einer Kommandozeile gearbeitet wird, wurde die notwendige Software in zwei separate Programme aufgeteilt.
Das erste Programm ist zuständig für die richtige Konfiguration der Kameras, die Aufnahme vom statischen Hintergrundbild von der linken und der rechten Kamera und das Abspeichern von diesen Aufnahmen an einer bestimmten Adresse im PL RAM.
Das zweite Programm, das Hauptprogramm, ist deutlich umfangreicher. Als Erstes wird die richtige Konfiguration der Kameras durchgeführt, der Zugriff auf die DMAs wird eingerichtet und alle IP Cores werden initialisiert. Danach werden separate Threads für die Aufnahme mit der Kamera, für das Starten der DMA Übertragungen und für das Starten der jeweiligen IP Cores eingerichtet. Die separaten Threads sind notwendig, damit eine parallele Ausführung der IP Cores, wie in Abbildung 3.15 dargestellt wurde, erreicht werden kann. Sobald die Daten am Ende der Verarbeitungspipeline ankommen und das Ergebnis des IP Cores für die *Flugbahnprognose* vorhanden ist, wird dieses per UDP Packet an des Fangsystem vom Roboter übermittelt.

4 Ergebnisse und Diskussion

In diesem Kapitel werden detaillierte Messungen und Messmethoden für die einzelnen Teilbereiche des Gesamtsystems beschrieben. Für jeden entworfenen Intellectual Property (IP) Core wurde eine Simulation durchgeführt und die simulierte Ausführungszeit mit der tatsächlichen gemessenen Ausführungszeit in der Hardware (HW) verglichen. Der IP Core wurde im Field Programmable Gate Array (FPGA) implementiert und auf diese Weise von einem Programm, das auf der Central Processing Unit (CPU) ausgeführt wurde, ausgemessen. Damit die Messungen nicht nur mittels Software (SW) durchgeführt werden, wurde zusätzlich ein IP Core von XILINX in das Design integriert, der die Ausführungszeit der IP Cores auch in der HW gemessen hat. Diese gemessenen Zeiten wurden schlussendlich verglichen, um sicherzustellen, dass die Ergebnisse der Messungen nicht durch äußere Einflüsse verfälscht wurden. In einem weiteren Punkt in diesem Kapitel wird auf die Menge an den benötigten FPGA Ressourcen eingegangen, sowohl für die einzelnen IP Cores, als auch für das Gesamtsystem. Im Rahmen dieses Kapitels werden auch die Fragen behandelt, die in Kapitel 1 gestellt wurden.

4.1 Messmethoden

Bei dem Entwurf aller IP Cores wurde als harte Deadline der maximalen Ausführungszeit, die Dauer von 10 *ms* herangezogen (vergleiche Abschnitt 3.5). Um also eine Bildrate von 100 Bilder pro Sekunde zu erreichen, müssen alle IP Cores die Ausführungszeit von maximal 10 *ms* einhalten. Während des Entwurfs der IP Cores, wurde die simulierte Ausführungszeit im Programm Vivado High Level Systhese (HLS) als Referenz herangezogen und es wurde diese als tatsächliche Ausführungszeit in HW angenommen. Mittels zahlreicher Optimierungen an den jeweiligen IP Cores konnte schlussendlich bei allen diesen IP Cores die harte Deadline der maximalen Ausführungszeit von 10 *ms* erreicht werden und sogar um einige Millisekunden unterboten werden. Jedoch, nicht bei allen IP Cores stimmte die simulierte Ausführungszeit mit der tatsächlich gemessenen Ausführungszeit in HW überein. Eine Abweichung konnte bei dem IP Core *Objekterkennung zur Ballmittelpunktbestimmung* beobachtet werden und wird in Abschnitt 4.2.1 näher erläutert.

4.1.1 Messung in Software

Für die Messung in SW wurde die Systemzeit unter Linux herangezogen. Um diese nutzen zu können, musste die Header-Datei *<sys/time.h>* in die Software eingebunden werden. Diese

Header-Datei ermöglicht das Auslesen der aktuellen Systemzeit mit einer Auflösung in Mikrosekunden. Vor dem Start der Berechnungen des IP Cores wurde mittels der Funktion *gettimeofday* die aktuelle Systemzeit gespeichert. Sobald der IP Core das Ergebnis bereitgestellt hat, wurde erneut die aktuelle Systemzeit gespeichert. Aus der Differenz dieser zwei ermittelten Zeitpunkte konnte die Ausführungszeit in Mikrosekunden der jeweiligen IP Cores berechnet werden.

4.1.2 Messung in Hardware

Für die Messung in HW wurde der Advanced eXtensively Interface (AXI) Timer IP Core von XILINX in das Design integriert. Dieser bietet einen 32-bit Timer/Counter in HW an, der über eine eigene Adresse im Speicher erreichbar ist. Über diese lässt er sich umfangreich konfigurieren und auslesen. Betrieben wurde der AXI Timer mit einer Frequenz von 100 *MHz*, wodurch eine Auflösung von 10 *ns* zur Verfügung stand. Da für diesen IP Core kein Linux Treiber vorhanden war, wurde eine einfache SW für die Ansteuerung der Timerregister geschrieben. Mit Hilfe dieser SW konnte, wie bei der Zeitmessung in SW (vergleiche Abschnitt 4.1.1), bevor der zu messende IP Core gestartet wurde, der aktuelle Timerwert vom AXI Timer ausgelesen werden. Anschließend wurde der IP Core gestartet und sobald dieser das Ergebnis geliefert hatte, wurde der AXI Timer gestoppt und der neue Wert ausgelesen. Auch in diesem Fall korrespondiert die Differenz der zwei ermittelten Timerwerte, nach einer Umwandung in Mikrosekunden, der Ausführungszeit der jeweiligen IP Cores.

4.2 Messergebnisse

In diesem Abschnitt werden die Ergebnisse der Messungen der einzelnen IP Cores näher beschrieben. Zu diesen Ergebnissen gehört die simulierte und die in HW gemessene Ausführungszeit (vergleiche Abschnitt 4.1), als auch die geschätzte und die tatsächliche Ressourcenausnutzung der IP Cores im FPGA. Weiters wird auch die geschätzte Leistungsaufnahme der IP Cores angegeben.
In folgenden Tabellen ist die Quelle der geschätzten Ressourcenausnutzung das Programm Vivado HLS, mit dem auch der Entwurf der IP Cores durchgeführt wurde. Die tatsächliche Ressourcenausnutzung und die geschätzte Leistungsaufnahme stammen aus dem Programm Vivado. Diese Informationen sind erst nach dem Entwicklungsschritt der Implementierung aufrufbar. Mittels des Programms Vivado wurde auch das Gesamtsystem entworfen.
Die wichtigsten Ressourcen, die angegeben werden, sind Block Random Access Memory (BRAM), Digital Signal Processor (DSP), Flip-Flop (FF) und Look-Up Table (LUT).

4.2.1 Hintergrundsubtraktion

Für den Entwurf und die Simulation wurde bei diesem IP Core die Vivado HLS Version 2015.4 verwendet. Als Taktfrequenz ist 200 *MHz* gewählt worden, bei der eine Ausführungszeit unter 10 *ms* erreicht werden konnte. Im fertigen Design wurde dieser IP Core mit dieser Frequenz betrieben. Die simulierte, als auch die gemessene Ausführungszeit des im FPGA implementierten IP Cores ist in folgender Tabelle 4.1 angegeben.

Ausführungszeit		
simuliert [ms]	gemessen in SW [ms]	gemessen in HW [ms]
7,865	7,879	7,871

Tabelle 4.1: Simulierte und gemessene Ausführungszeit vom IP Core mit der Hintergrundsubtraktion.

Ein Vergleich zeigt, dass die Ergebnisse der Simulation bezüglich der Ausführungszeit und die der gemessenen Ausführungszeit in HW, bis auf geringe Abweichungen im Mikrosekundenbereich, übereinstimmen. Die hier angegebenen Ausführungszeiten sind die maximal erreichten Zeiten, also worst case. Einige Ergebnisse erzielten auch geringfügig kürzere Zeiten (Unterschied maximal 2 μs), da es sich aber um eine Ausführung in HW handelt, ist die Ausführungszeit deterministisch und benötigt somit immer ungefähr gleich lang. Daher wurde die Menge der Messungen auf zehn beschränkt. Die Abweichungen sind auf Messungenauigkeiten zurückzuführen, die aufgrund der Messung aus einer SW aus, entstehen. Dadurch, dass die SW unter Linux ausgeführt wird, kommt es vor, dass sie zu geringfügig unterschiedlichen Zeiten vom Betriebssystem Scheduler zur Ausführung freigegeben wird. Somit werden auch die Messpunkte zu unterschiedlichen Zeiten gemessen, einmal früher, einmal später, im Vergleich und dadurch entstehen die genannten Abweichungen. Wichtig ist, dass die Ausführungszeit von 10 *ms* nicht überschritten wird und da hat dieser IP Core mehr als 2 *ms* Reserve.

In folgender Tabelle 4.2 ist die geschätzte Ausnutzung der FPGA Ressourcen angegeben und zusätzlich, als Vergleich, die tatsächliche Ausnutzung der Ressourcen nach der Implementierung.

Ressource		BRAM 18K	DSP48E	FF	LUT
verfügbar im FPGA		1510	2020	554800	277400
benötigt vom IP Core	Abschätzung	0	2	6227	6818
	tatsächlich	0	2	5511	4282
Ausnutzung in %	Abschätzung	0	~0	1	2
	tatsächlich	0	0,1	0,99	1,54

Tabelle 4.2: Ressourcenabschätzung und die tatsächliche Ausnutzung nach der Implementierung für den IP Core mit der Hintergrundsubtraktion.

Die hier angegebene Ressourcenausnutzung ist die von einem IP Core. Da aber, wie bereits in Abschnitt 3.5 erwähnt, dieser IP Core im Gesamtsystem doppelt implementiert ist, ein IP Core verarbeitet die Bilder von der linken Kamera, ein IP Core die Bilder von der rechten Kamera, müssen die Ressourcen aus der Tabelle 4.2 ungefähr verdoppelt werden, um die insgesamt benötigten Ressourcen dieser Verarbeitungsstufe zu erhalten. Der Grund, warum hier von ungefähr verdoppeln gesprochen wird, ist der, dass, obwohl die beiden Versionen der IP Cores identisch sind, benötigen sie nach der Implementierung in der HW geringfügig unterschiedlich viele Ressourcen. Die Erklärung für dieses Verhalten wird in späterem Abschnitt 4.3 behandelt. Die hier angegebene Menge an Ressourcen ist die höhere von den zwei IP Cores. Der Unterschied ist aber gering und es ist somit nicht notwendig die genaue Ressourcenausnutzung beider IP Cores hier anzugeben. Dadurch, dass in Tabelle 4.2 der IP Core mit dem höheren Bedarf an Ressourcen angegeben wurde, reicht es diese zu verdoppeln, um einen Überblick über die benötigten Ressourcen beider IP Cores zu bekommen.

Die geschätzte Leistungsaufnahme dieses IP Cores beträgt **0,1 *W***. Dies ist die Leistungsaufnahme von einem IP Core. Um die gesamte Leistungsaufnahme dieser Verarbeitungsstufe zu ermitteln, muss diese ungefähr verdoppelt werden. Analog zu der Ressourcenausnutzung, ist auch die Leistungsaufnahme für die beiden IP Cores unterschiedlich, da sie von der Menge der benötigten Ressourcen abhängig ist. Die hier angegebene Leistungsaufnahme ist die von dem IP Core mit der Ressourcenausnutzung wie in Tabelle 4.2 angegeben und somit die geringfügig höhere von den beiden IP Cores.

4.2.2 Objekterkennung zur Ballmittelpunktbestimmung

Für den Entwurf und die Simulation wurde auch bei diesem IP Core die Vivado HLS Version 2015.4 verwendet. Als Taktfrequenz ist erneut 200 *MHz* gewählt worden, bei der eine Ausführungszeit unter 10 *ms* erreicht werden konnte. Im fertigen Design wurde dieser IP Core mit dieser Frequenz betrieben. Die simulierte, als auch die gemessene Ausführungszeit des im FPGA implementierten IP Cores ist in folgender Tabelle 4.3 angegeben.

Ausführungszeit		
simuliert [ms]	gemessen in SW [ms]	gemessen in HW [ms]
6,886	7,072	7,067

Tabelle 4.3: Simulierte und gemessene Ausführungszeit vom IP Core mit der Objekterkennung zur Ballmittelpunktbestimmung.

Ein Vergleich zeigt, dass die Ergebnisse der Simulation bezüglich der Ausführungszeit und die der gemessenen Ausführungszeit in HW, mehr abweichen, als bei dem vorherigen IP Core mit der *Hintergrundsubtraktion*, beschrieben in Abschnitt 4.2.1. Der Grund für diese Tatsache ist die Methode, wie die Daten aus dem programmable logic random access memory (PL RAM) geladen werden. Bei dem IP Core mit der *Hintergrundsubtraktion* wird der Datentransfer aus dem PL RAM über die AXI4 Full Schnittstelle komplett im Burst Modus durchgeführt. Bei diesem IP Core, *Objekterkennung zur Ballmittelpunktbestimmung*, wird der Datentransfer auch im Burst Modus durchgeführt, jedoch wird immer nur eine komplette Zeile, bestehend aus 300 Pixeln, geladen und nicht das gesamte 300x300 Pixel Bild. Es wird somit nicht ein Burst Transfer mit der Länge von 300x300 Pixeln, also 90000 Pixeln direkt hintereinander durchgeführt. Aufgrund der Tatsache, dass aus dem großen Bild mit 2048x768 Pixel nur ein Ausschnitt von 300x300 Pixeln gebraucht und geladen wird, geschieht der Datentransfer aufgeteilt in 300 Bursts mit jeweils einer Länge von 300 Pixel. Aus dieser Aufteilung in 300 separate Bursts, resultiert die geringfügige Abweichung der simulierten Ausführungszeit von der tatsächlich gemessenen Ausführungszeit in HW. Diese Begründung unterstützt auch die Tatsache, dass in der ursprünglichen Version dieses IP Cores, der Datentransfer des 300x300 Pixel Bildes ohne Burst durchgeführt wurde und die Simulation hat eine Ausführungszeit von **7,437 *ms*** ergeben. Jedoch wenn der IP Core implementiert wurde und eine Messung der Ausführungszeit in HW durchgeführt wurde, dann resultierte daraus eine Ausführungszeit von bis zu etwa **16 *ms***! Dadurch ist begründet, dass die Simulation der Datentransfers über die AXI4 Full Schnittstelle gewisse Abweichungen hat und nicht immer die tatsächliche Ausführungszeit der Datentransfers in HW simulieren kann. Auf diese Weise kann die Abweichung der Ausführungszeit dieses IP Cores, zwischen der Simulation und der HW, begründet werden.

Die hier angegebenen Ausführungszeiten sind erneut die maximal erreichten Zeiten, also worst case. Einige Ergebnisse erzielten auch geringfügig kürzere Zeiten (Unterschied maximal 2 μs), da es sich aber um eine Ausführung in HW handelt, ist die Ausführungszeit deterministisch und benötigt somit immer ungefähr gleich lang. Daher wurde die Menge der Messungen auch bei diesem IP Core auf zehn beschränkt. Die Abweichungen sind erneut auf Messungenauigkeiten zurückzuführen, die bereits bei dem vorherigen IP Core mit der *Hintergrundsubtraktion*, in Abschnitt 4.2.1, beschrieben wurden. Erneut ist es wichtig, dass die Ausführungszeit von 10 ms nicht überschritten wird und da hat dieser IP Core fast 3 ms Reserve.

In folgender Tabelle 4.4 ist die geschätzte Ausnutzung der FPGA Ressourcen angegeben und zusätzlich, als Vergleich, die tatsächliche Ausnutzung der Ressourcen nach der Implementierung.

Ressource		BRAM 18K	DSP48E	FF	LUT
verfügbar im FPGA		1510	2020	554800	277400
benötigt vom IP Core	Abschätzung	289	9	111155	108791
	tatsächlich	273	9	54511	70598
Ausnutzung in %	Abschätzung	19	~0	20	39
	tatsächlich	18,08	0,45	9,83	25,45

Tabelle 4.4: Ressourcenabschätzung und die tatsächliche Ausnutzung nach der Implementierung für den IP Core mit der Objekterkennung zur Ballmittelpunktbestimmung.

Die hier angegebene Ressourcenausnutzung ist die von einem IP Core. Auch dieser IP Core wird im Gesamtsystem doppelt implementiert und die Ressourcen aus der Tabelle 4.4 müssen ungefähr verdoppelt werden, falls man die insgesamt benötigten Ressourcen dieser Verarbeitungsstufe wissen möchte. Der Grund, warum hier von ungefähr verdoppeln gesprochen wird, ist der Selbe, wie beim IP Core *Hintergrundsubtraktion*, beschrieben in Abschnitt 4.2.1. Die Erklärung für dieses Verhalten wird in späterem Abschnitt 4.3 behandelt. Die hier angegebene Menge an Ressourcen ist die höhere von den zwei IP Cores. Der Unterschied ist aber erneut gering und es ist somit nicht notwendig die genaue Ressourcenausnutzung beider IP Cores hier anzugeben. Dadurch, dass in Tabelle 4.4 der IP Core mit dem höheren Bedarf an Ressourcen angegeben wurde, reicht es diese zu verdoppeln, um einen Überblick über die benötigten Ressourcen beider IP Cores zu bekommen.

Die geschätzte Leistungsaufnahme dieses IP Cores beträgt **3,6 *W***. Dies ist die Leistungsaufnahme von einem IP Core. Um die gesamte Leistungsaufnahme dieser Verarbeitungsstufe zu ermitteln, muss diese ungefähr verdoppelt werden. Analog zu der Ressourcenausnutzung, ist auch die Leistungsaufnahme für die beiden IP Cores unterschiedlich, da sie von der Menge der benötigten Ressourcen abhängig ist. Die hier angegebene Leistungsaufnahme ist die von dem IP Core mit der Ressourcenausnutzung wie in Tabelle 4.4 angegeben und somit die geringfügig höhere von den beiden IP Cores.

Triangulation

Nachdem der IP Core *Objekterkennung zur Ballmittelpunktbestimmung* die Ergebnisse in Form vom ermittelten Ballmittelpunkt im linken und im rechten Bild bereitstellt, wird die Triangulation durchgeführt (siehe dazu auch Abschnitt 3.3.5). Die Triangulation wird nicht im FPGA implementiert, sondern direkt im Hauptprogramm auf der CPU ausgeführt (siehe dazu auch

Abschnitt 3.2). Von den Ergebnissen kann an dieser Stelle nur die gemessene Ausführungszeit angegeben werden. Erneut wurde die Messung in SW, als auch in HW mittels des AXI Timers durchgeführt. Die Messungen wurden hundert Mal ausgeführt und anschließend gemittelt. Das Ergebnis der Messung mittels der SW ergibt eine Ausführungszeit von **28,62 μs**. Die Messung mittels des AXI Timers ergibt eine Ausführungszeit von **24,1 μs**. Diese gemessenen Zeiten, im Gegensatz zu den IP Cores, sind nicht deterministisch und können sich in Abhängigkeit der CPU Last ändern. In Abbildung 4.1 ist zusätzlich ein Histogramm der hundert Messungen dargestellt. Wie aber ersichtlich, ist die Ausführungszeit der Triangulation, im Vergleich zu den IP Cores, vernachlässigbar klein.

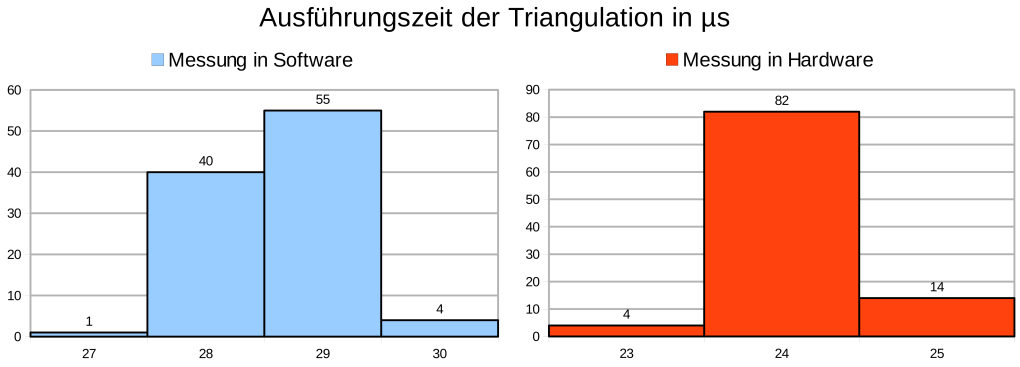

Abbildung 4.1: Histogramm der Ausführungszeiten der Triangulation für die Messung in Software und in Hardware.

4.2.3 Flugbahnprognose

Aufgrund der Verfügbarkeit einer neuen Version von Vivado HLS, Version 2016.2, wurde diese für den Entwurf und die Simulation bei diesem IP Core verwendet. Dieser IP Core wurde entworfen und simuliert für eine Taktfrequenz von 100 *MHz* und wurde mit dieser Frequenz im fertigen Design betrieben. Laut Vivado HLS und der Simulation während des Entwurfs konnte, aufgrund von Timingproblemen, keine höhere Frequenz bei diesem IP Core erreicht werden. Anders als bei den vorherigen IP Cores *Hintergrundsubtraktion* und *Objekterkennung zur Ballmittelpunktbestimmung*, hängt die Zeit, die dieser IP Core benötigt um die Ergebnisse zu liefern, von dem Parameter ab, mit dem der IP Core gestartet wird. Dieser Parameter ist die aktuelle Bildnummer. Die Vorgehensweise, wie die Trajektorie ermittelt wird, wurde bereits ausführlich in Abschnitt 3.3.6 beschrieben. Kurz gesagt, je höher ist die Bildnummer, umso länger benötigt der IP Core für die Ermittlung der Trajektorie und das Liefern von Ergebnissen. Da die Simulation von hundert Durchläufen von dem IP Core zeitintensiv war, wurde sie auf zehn Messungen, alle zehn Bilder, reduziert. Die folgende Tabelle 4.5 zeigt den Vergleich zwischen der simulierten Ausführungszeit und der gemessenen Ausführungszeit des IP Cores, wenn dieser im FPGA implementiert wurde.

	Ausführungszeit		
Bildnummer	simuliert [μs]	gemessen in SW [μs]	gemessen in HW [μs]
1	31,845	113	107
10	308,385	387	381
20	615,585	696	690
30	922,785	1003	997
40	1229,985	1316	1310
50	1537,185	1623	1617
60	1844,385	1927	1921
70	2151,585	2237	2231
80	2458,785	2542	2536
90	2765,985	2888	2883
100	3042,435	3131	3125

Tabelle 4.5: Simulierte und gemessene Ausführungszeit vom IP Core mit der Flugbahnprognose.

Die hier angegebenen Ausführungszeiten sind erneut die maximal erreichten Zeiten, also worst case. Einige Ergebnisse erzielten auch geringfügig kürzere Zeiten (Unterschied maximal 2 μs), da es sich aber um eine Ausführung in HW handelt, ist die Ausführungszeit deterministisch und benötigt somit immer ungefähr gleich lang. Daher wurde die Menge der Messungen auch bei diesem IP Core auf zehn beschränkt. Die Abweichungen sind erneut auf Messungenauigkeiten zurückzuführen, die bereits in Abschnitt 4.2.1 beschrieben wurden. Erneut ist es wichtig, dass die Ausführungszeit von 10 *ms* nicht überschritten wird und da hat dieser IP Core bei der Ausführung für das hundertste Bild fast 7 *ms* Reserve.

Die Messung der Ausführungszeit in HW wurde mit einem einfachen Programm automatisiert und daher konnten die Ergebnisse für alle Bildnummern ermittelt werden. Es wurde gemessen sowohl mittels der SW Methode, als auch mittels der HW Methode. Diese Ergebnisse werden zusammen mit einer gemittelten Kurve der simulierten Ausführungszeiten in Abbildung 4.2 dargestellt.

Wie ersichtlich, steigt die Ausführungszeit mit steigender Bildnummer annähernd linear an und ist um eine konstante Zeit gegenüber der simulierten Ausführungszeit verschoben. Diese Verschiebung hängt mit der zusätzlichen Verarbeitungszeit im Programm zusammen. Die Messungen wurden nämlich gestartet, noch bevor die Eingangsparameter im Programm gesetzt wurden. Würde man die Messung unmittelbar vor dem Start von dem IP Core starten und unmittelbar nachdem das Ergebnis bereitgestellt wurde beenden, dann würde die Differenz zwischen der simulierten Ausführungszeit und der gemessenen Ausführungszeit in SW und HW deutlich geringer ausfallen. Vergleiche dazu auch die Differenz der simulierten und der gemessenen Ausführungszeit der beiden anderen IP Cores in den Abschnitten 4.2.1 und 4.2.2.
Mit dieser Verschiebung ist in Abbildung 4.2 die Kurve der simulierten Ausführungszeit deutlich erkennbar, im Gegensatz zu den Kurven der gemessenen Ausführungszeit, bei denen schwer zwischen der SW und der HW Messung unterschieden werden kann, da diese sich eng beieinander befinden.

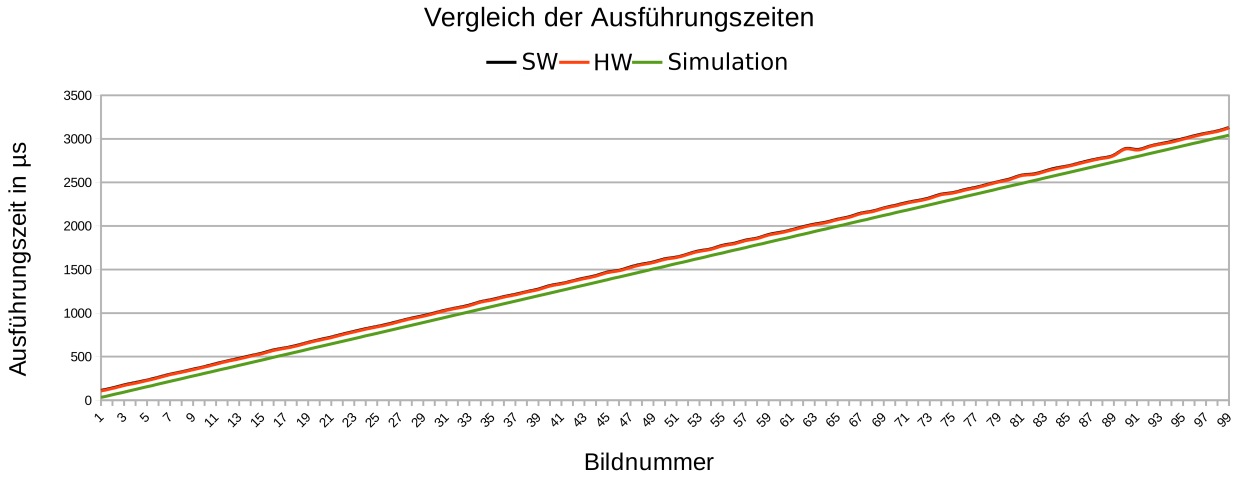

Abbildung 4.2: Darstellung und Vergleich der Ausführungszeiten gemessen in SW und HW. Zusätzlich ist die gemittelte Kurve der simulierten Ausführungszeiten eingetragen.

In folgender Tabelle 4.6 ist die geschätzte Ausnutzung der FPGA Ressourcen angegeben und zusätzlich, als Vergleich, die tatsächliche Ausnutzung der Ressourcen nach der Implementierung.

Ressource		BRAM 18K	DSP48E	FF	LUT
verfügbar im FPGA		1510	2020	554800	277400
benötigt vom IP Core	Abschätzung	515	7	1919	54117
	tatsächlich	511	7	1613	42880
Ausnutzung in %	Abschätzung	34	~0	~0	19
	tatsächlich	33,84	0,35	0,29	15,46

Tabelle 4.6: Ressourcenabschätzung und die tatsächliche Ausnutzung nach der Implementierung für den IP Core mit der Flugbahnprognose.

Die variable Ausführungszeit dieses IP Cores, in Abhängigkeit des Parameters der aktuellen Bildnummer, ist durch das Design des IP Cores bedingt. Dieses Design basiert, unter anderem auf einer Schliefe mit einer variablen Grenze, welche die aktuelle Bildnummer darstellt. Dadurch ist die Ausführungszeit von dieser Grenze der Schleife abhängig. Eine andere Möglichkeit, wie das Design aufgebaut werden konnte war, die Schleife mit einer festen Grenze zu definieren. Diese Grenze wäre die maximal mögliche Bildnummer und der Inhalt der Schleife wäre nur in Abhängigkeit einer Variable (=aktuelle Bildnummer) ausgeführt worden. Auf diese Weise konnte der IP Core mit einer konstanten simulierten Ausführungszeit von **3,073** ***ms*** entworfen werden.
Es gilt, dass Schleifen mit variablen Grenzen, die Erstellung von optimaler HW durch die HLS, beeinflussen können [24, 314 ff.]. Durch solche Schleifen können einige Optimierungen von HLS verhindert werden. Die Ressourcenausnutzung des IP Cores mit der festen Ausführungszeit, im Vergleich zu dem IP Core mit der variablen Ausführungszeit (siehe Tabelle 4.6), wurde aber geringfügig größer. Die Abschätzung der Ressourcen änderte sich bei FFs auf 1945 und LUTs auf 54153. Die tatsächliche Ressourcenausnutzung wurde nicht ermittelt, da schlussendlich nur der IP Core mit der variablen Ausführungszeit im FPGA implementiert wurde. Es wurde entschieden, dass die variable Ausführungszeit für das Gesamtsystem die bessere Wahl ist, da der IP Core mit

der festen Ausführungszeit geringfügig längere Zeit benötigt um das Ergebnis bereitzustellen und auch mehr Ressourcen benötigt. Durch die Schleife mit variabler Grenze wurden daher, in diesem Fall, keine Optimierungen von HLS verhindert.

Die geschätzte Leistungsaufnahme des implementierten IP Cores mit der variablen Ausführungszeit beträgt **0,9 *W***.

4.3 Ergebnisse der Messungen des Gesamtsystems

In diesem Abschnitt werden die Ergebnisse der Messungen des Gesamtsystems mit allen implementierten IP Cores beschrieben. Zu diesen Ergebnissen gehört vor allem die tatsächliche FPGA Ressourcenausnutzung des implementierten Designs und die geschätzte Leistungsaufnahme des Gesamtsystems. Am Ende dieses Abschnitts wird darauf eingegangen, warum das Gesamtsystem die gestellten Anforderungen **nicht** erfüllt.

Die Quelle der Daten in diesem Abschnitt ist das Programm Vivado der Version 2016.2, mit dem auch das Gesamtsystem entworfen wurde. Die Tabelle 4.7 gibt die FPGA Ressourcenausnutzung des Gesamtsystem an.
Von den zusätzlichen Ressourcen werden hier Look-Up Tables, die als Random Access Memory verwendet werden können (LUTRAM), Input/Output (IO), Gigabit Transceiver (GT), Global Buffer (BUFG), Mixed-Mode Clock Manager (MMCM), Phase-Locked Loop (PLL) und Peripheral Component Interconnect Express (PCIe), angegeben.

Ressource	Nutzung	verfügbar	Ausnutzung in %
LUT	240854	277400	**86,83**
LUTRAM	12672	108200	11,71
FF	167453	554800	30,18
BRAM	624	755	**82,65**
DSP	29	2020	1,44
IO	75	362	20,72
GT	4	16	25,00
BUFG	11	32	34,38
MMCM	2	8	25,00
PLL	1	8	12,50
PCIe	1	1	100,00

Tabelle 4.7: FPGA Ressourcenausnutzung des Gesamtsystems.

Die Ausnutzung der FPGA Ressourcen für LUT und BRAM ist sehr hoch und bei dem Schritt der Synthese und der anschließenden Implementierung konnte mit den Standardeinstellungen von Vivado kein funktionierendes Design erstellt werden. Aufgrund der hohen Ausnutzung hatten einige der Zellen im FPGA Schwierigkeiten das Timing einzuhalten. Deshalb wurden mehrere alternative Strategien der Implementierung getestet. Schlussendlich wurde mit der Strategie „Performance_NetDelay_high“ ein funktionierendes Design erstellt, welches das Timing einhalten konnte.

Wenn man die Ressourcenausnutzung in Prozent von den wichtigsten Ressourcen der einzelnen IP Cores addiert (siehe Tabellen 4.2, 4.4 und 4.6), kommt man bei LUT auf etwa 70%, bei FF auf etwa 22% und bei BRAM auf etwa 70%. Die übrigen Ressourcen werden von den anderen notwendigen IP Cores, wie Direct Memory Access (DMA), PCIe, PL RAM Memory Controller, diverse Interconnects und andere, benutzt.

In folgender Abbildung 4.3 ist die FPGA Ressourcenausnutzung visuell dargestellt. Dabei werden in verschiedenen Farben die einzelnen IP Cores voneinander unterschieden. Den größten Teil vom FPGA, etwa die Hälfte, nehmen die beiden IP Cores *Objekterkennung zur Ballmittelpunktbestimmung* ein, dargestellt in der Farbe **blau** und **dunkel grün**. Der nächst größere IP Core ist die *Flugbahnprognose*, dargestellt in der Farbe **hell grün**. Als letzte folgen die beiden IP Cores *Hintergrundsubtraktion*, dargestellt in der Farbe **cyan** und **magenta**. Die restlichen notwendigen Ressourcen werden in der Farbe **rot** dargestellt.

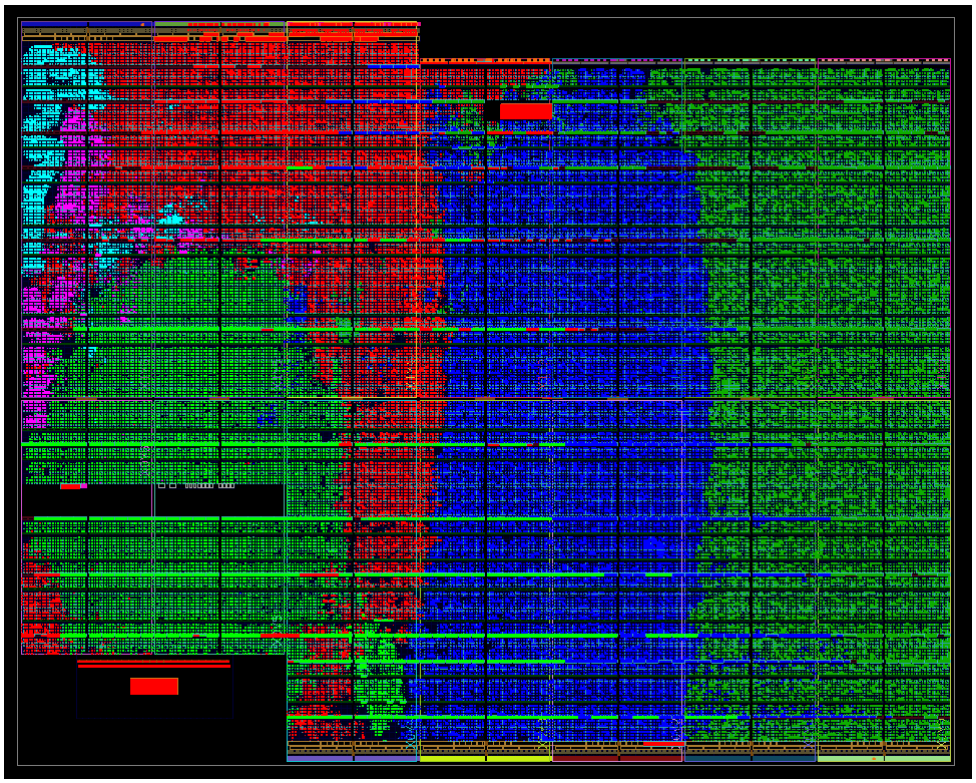

Abbildung 4.3: Visualisierung der Ressourcenausnutzung im FPGA.

Wenn man die Abbildung 4.3 mehrfach vergrößert werden die einzelnen Logikzellen, aus denen der FPGA besteht, sichtbar. Eine davon ist in Abbildung 4.4 dargestellt. Hierbei handelt es sich um eine Logikzelle, die zu einem der IP Cores *Objekterkennung zur Ballmittelpunktbestimmung* in Abbildung 4.3 gehört und mit der blauen Farbe gekennzeichnet ist.
Ganz links in Abbildung 4.4 sind die vier Blöcke, die als LUT dienen, positioniert. Danach sind drei 2-zu-1 Multiplexer vorhanden. Es folgt ein sogenannter CARRY4 Block und ganz rechts sind acht FF Blöcke zu sehen. Die hier nicht mit der blauen Farbe markierten Teile der Logikzelle sind von keinem IP Core belegt und sind somit nicht beschaltet.

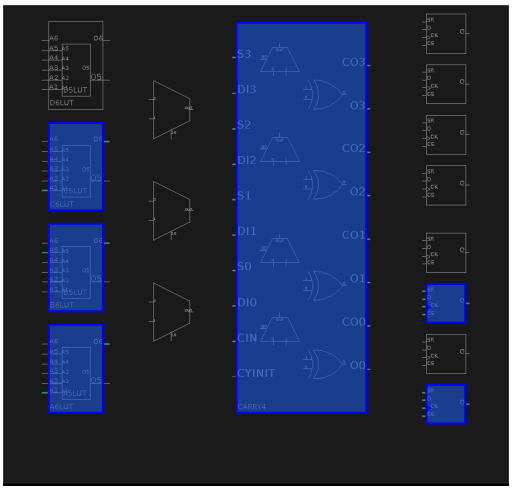

Abbildung 4.4: Eine der Logikzelle des IP Cores für Objekterkennung zur Ballmittelpunktbestimmung.

Die simulierte Leistungsaufnahme des Gesamtsystems, bestehend aus den beiden ARM CPUs und dem FPGA, beträgt **13,9** ***W***. Die beiden ARM CPUs benötigen davon, bei einer CPU Last von 50%, insgesamt eine Leistung von **1,7** ***W***. Die restliche Leistung von **12,2** ***W*** wird vom FPGA benötigt. Von den **12,2** ***W*** benötigen die entworfenen IP Cores (siehe Abschnitte 4.2.1, 4.2.2 und 4.2.3) insgesamt eine Leistung von ungefähr **8,3** ***W***. Die übrige Leistung von ungefähr **3,9** ***W*** wird von den restlichen implementierten IP Cores im FPGA benötigt. Es wird hier von einer ungefähr benötigten Leistung gesprochen, da die IP Cores, welche doppelt im FPGA implementiert sind, nicht beide eine identische Leistungsaufnahme ausweisen. Der Grund dafür ist, dass die IP Cores geringfügig unterschiedlich viele Ressourcen benötigen (vergleiche dazu Abschnitte 4.2.1 und 4.2.2). Es wurde bei den IP Cores jeweils die höhere Leistungsaufnahme angegeben.

Die gesamte Entwicklungsplattform, das Mini-ITX Board, beschrieben in Abschnitt 2.2, wurde mit einem ATX-Netzteil mit einer Maximalleistung von 420 W betrieben. Während das Hauptprogramm ausgeführt wurde, konnte mit einem Leistungsmessgerät eine Leistungsaufnahme von **34,7** ***W*** gemessen werden. Dabei war die CPU Auslastung konstant bei 100%, beide USB 3.0 Kameras waren an der USB 3.0 Erweiterungskarte im PCIe Steckplatz angeschlossen und eine Aufnahme mittels beider Kameras war im Gange.

Der Grund, warum identische IP Cores schlussendlich unterschiedlich viele Ressourcen in der HW benötigen, liegt an dem Designschritt der Implementierung. Bei der Implementierung wird unter anderem versucht die geforderte Funktionalität mit möglichst wenigen Ressourcen zu erzielen. Dabei werden etliche Optimierungen an der Logik durchgeführt, welche einerseits in Abhängigkeit der Platzierung im FPGA, andererseits auch von der umgebenden Logik im FPGA, unterschiedlich sind. Diese Optimierungen umfassen Flächenoptimierung, Energieoptimierung, kombinatorische und sequentielle Logikoptimierung und weitere, damit eine möglichst effiziente Logik für den FPGA entworfen wird [25, 50 ff.]. Dadurch kommt es vor, dass identische IP Cores unterschiedliche Ressourcenausnutzung aufweisen können.
Diese Optimierungen sind auch der Grund, warum die Abschätzung der Ressourcen der IP Cores aus der HLS und die tatsächliche Ausnutzung der Ressourcen nach der Implementierung im FPGA, teilweise um Größenordnungen unterschiedlich sind (siehe Tabellen 4.2, 4.4 und 4.6).

Warum werden die gestellten Anforderungen nicht erfüllt?

Früher in diesem Abschnitt wurde gezeigt, dass die Teile des Gesamtsystems, wie sie entworfen und aufgebaut wurden, die geforderte Bildrate von 100 Bilder pro Sekunde erreichen können. Alle IP Cores sind in der Lage, Ergebnisse mit einer Ausführungszeit von unter 10 *ms*, zu produzieren. Es wurde eine Software entwickelt, welche die Bilder von den beiden Universal Serial Bus (USB) 3.0 Kameras lädt, sie an die ersten IP Cores in der Verarbeitungspipeline übergibt, die einzelnen IP Cores steuert und den Datenfluss zwischen den IP Cores koordiniert (siehe Abschnitt 3.5). Das Problem bereiten aber die USB 3.0 Kameras, die für die Funktionalität des Gesamtsystems eine wesentliche Rolle spielen. Wie bereits in Abschnitt 3.3.1 beschrieben wurde, sind die beiden Kameras über eine USB 3.0 Erweiterungskarte, welche in dem PCIe Steckplatz der Mini-ITX Plattform installiert wurde, an das Gesamtsystem angebunden. In der entwickelten Software wird auf die beiden Kameras, mittels dem vom Hersteller IDS zur Verfügung gestelltem Treiber, zugegriffen. Eine Kommunikation mit den Kameras, ohne den Treiber zu benutzen, ist nicht möglich. An dieser Stelle entsteht der Flaschenhals, welcher die korrekte Funktionalität des Gesamtsystems für eine Bildrate von 100 Bilder pro Sekunde verhindert. In anderen Worten, eine Bildrate von 100 Bilder pro Sekunde wird nicht erreicht. Der Grund dafür sind die beiden ARM CPU Kerne, getaktet mit jeweils 800 *MHz*, welche nicht genügend Leistung haben, um die großen Datenmengen von den USB 3.0 Kameras in einer ausreichenden Geschwindigkeit zu laden. Diese Tatsache konnte auch nicht vorher ermittelt werden, ohne dass die Kameras an dem System tatsächlich getestet wurden, denn der Kamerahersteller bietet keine Informationen über die minimalen HW System Anforderungen an. Es werden nur die minimalen SW Anforderungen für den Treiber angegeben, die erfüllt wurden.
Die Kameras lassen sich über den Treiber umfangreich konfigurieren, unter anderem auch der Pixeltakt, mit dem die Kameras über die USB 3.0 Schnittstelle die Daten übermitteln. Die Einstellungen vom Pixeltakt, der Belichtungszeit und der Bildrate, sowie der Größe des Bildbereichs, hängen voneinander ab. Damit bei einem Bildbereich von 2048x768 Pixel, eine Bildrate von 100 Bilder pro Sekunde erreichbar ist, muss die Belichtungszeit auf 1 *ms* und der Pixeltakt auf 336 *MHz* gesetzt werden. Wird einer dieser Parameter verändert, so ändert sich die maximal erreichbare Bildrate. Der benötigte Pixeltakt von 336 *MHz* ist nahe an dem maximal unterstützten Pixeltakt von den Kameras. Bei diesem können aber keine Bilder von den Kameras empfangen werden, da jede Übertragung mit einer Fehlermeldung abbricht. Der Pixeltakt, bei dem die wenigsten Übertragungsfehler auftreten, ist der minimal unterstützte von den Kameras und zwar 38 *MHz*. Je höher die Einstellung für den Pixeltakt, umso öfter passieren Übertragungsfehler und die aufgenommenen Bilder gehen verloren. Das Problem bei dem niedrigen Pixeltakt von 38 *MHz*, ist die maximal erreichbare Bildrate von etwa 25 Bildern pro Sekunde. Während die Datenübertragung läuft, schwankt die Bildrate zudem stark zwischen 1 und 25 Bilder pro Sekunde. Die Auslastung beider CPU Kerne liegt dabei konstant bei 100%, wofür hauptsächlich der Kameratreiber verantwortlich ist. Die Schwankung der Bildrate lässt sich durch die auftretenden Übertragungsfehler erklären. Je mehr Übertragungsfehler passieren, umso geringer ist die vom Kameratreiber gemeldete Bildrate. Diese starke Schwankung der Bildrate ist in Abbildung 4.5 dargestellt. Die Ausreißer in dieser Abbildung treten also dann auf, wenn die wenigsten Übertragungsfehler passieren. Das ist dann der Fall, wenn die CPUs es ab und zu schaffen, trotz der 100-prozentigen Auslastung, die ankommende Menge an Daten von den Kameras über die USB 3.0 Schnittstelle zu verarbeiten. Die Erklärung für die Schwankung der Bildrate und den entstandenen Flaschenhals ist somit die fehlende Rechenleistung der CPUs, wodurch an die Grenze der verwendeten Technologie gestoßen wurde.

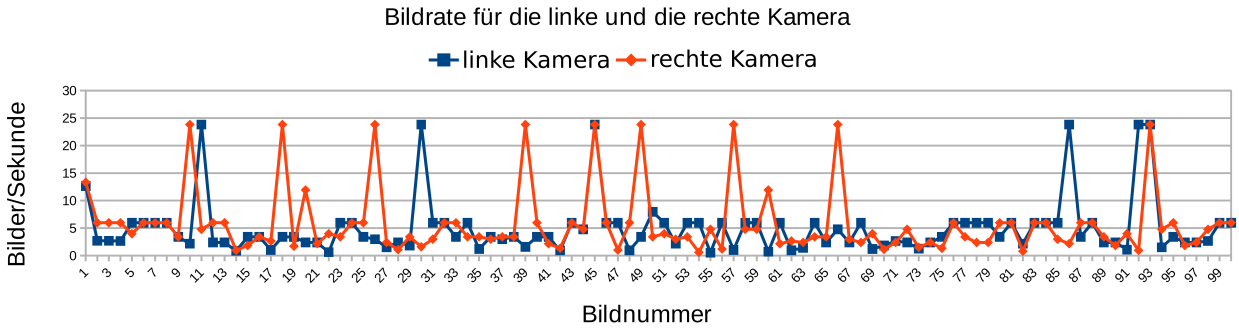

Abbildung 4.5: Darstellung der Schwankung der Bildrate.

Die Kameras wurden mit der selben USB 3.0 Erweiterungskarte auch mit dem Personal Computer (PC), welcher für die Entwicklung benutzt wurde (siehe Abschnitt 3.1), mit den identischen Parametern getestet. Mit dem Pixeltakt von 336 *MHz*, der Belichtungszeit von 1 *ms* und dem Bildbereich von 2048x768 Pixel, konnte eine stabile Bildrate von 100 Bilder pro Sekunde, mittels beider Kameras gleichzeitig, erreicht werden. Dabei wurden keine Übertragungsfehler festgestellt und die Bildaufnahme funktionierte problemlos.

An dieser Stelle soll auch erwähnt werden, dass ein anderer Entwickler ein ähnliches System aufgebaut hat [30]. Dabei wurde versucht zwei USB 3.0 Kameras (nicht näher spezifiziert) an dem Mini-ITX Board mittels einer USB 3.0 Erweiterungskarte für den PCIe Steckplatz, anzuschließen und zu betreiben. Die erreichte Bildrate war geringer, als die in dieser Diplomarbeit. Als ein möglicher Grund dieses Ergebnisses wurde die nicht genug leistungsfähige CPU identifiziert.

Damit können die Fragen, die in Kapitel 1 gestellt wurden, beantwortet werden:

- **Ja**, die einzelnen Teilbereiche wie, die Bilderfassung, die Bildverarbeitung, die Objekterkennung und die Flugbahnprognose können alle auf der gewählten System on Chip (SoC) Plattform implementiert werden. Das FPGA bietet dafür genug Ressourcen an.
- Die Aufteilung der Aufgaben, welche in HW und welche in SW implementiert werden, wurde in Abschnitt 3.2 behandelt.
- **Nein**, die SoC Plattform erreicht nicht die geforderte Bildrate von 100 Bilder pro Sekunde.

5 Zusammenfassung und Ausblick

Die Grundlage für diese Diplomarbeit stellt der „Wurftransport“-Ansatz dar. Dieser besteht aus Robotern, welche sich die Objekte gegenseitig zuwerfen und dadurch eine Basis für ein Transportsystem schaffen, mit dem Objekte vom Punkt A zum Punkt B transportiert werden können. Dies ist ein möglicher Ansatz für flexiblere Produktionslinien.
In dieser Diplomarbeit wurde das System, das für diesen Ansatz notwendig ist, auf einer System-on-Chip (SoC) Plattform, bestehend aus Central Processing Units (CPUs) und Field Programmable Gate Array (FPGA), entworfen. Dabei stand die Hardware/Software Partitionierung im Fokus, sprich, welche Aufgaben werden im FPGA und welche auf der CPU implementiert.

5.1 Zusammenfassung

Eine der wichtigsten Fragen, die am Anfang dieser Diplomarbeit gestellt wurde, ob das entworfene System die Bildrate von 100 Bilder pro Sekunde erreichen kann, muss mit **nein** beantwortet werden. Es wurde im Rahmen dieser Diplomarbeit auf die technologischen Grenzen der verwendeten SoC Plattform gestoßen (fehlende Rechenleistung der CPUs) und somit konnte schlussendlich das System, nicht wie gewünscht, in Betrieb genommen werden. Andererseits wurde aber auch gezeigt, dass die SoC Plattform genug FPGA Ressourcen zur Verfügung stellt, damit das System für den Transport mittels Werfen und Fangen, implementiert werden kann. Weiters wurde gezeigt, wie die einzelnen notwendigen Teilbereiche des Gesamtsystems entworfen und implementiert wurden und dass diese schnell genug Ergebnisse liefern können, damit eine Bildrate von 100 Bilder pro Sekunde erreichbar ist. Die Teilbereiche (Verarbeitungsblöcke im FPGA) sind in der Lage, diese Bildrate zu erreichen, das Gesamtsystem aber nicht. Diese Tatsache wurde in Abschnitt 4.3 beschrieben. Trotz ausgiebiger Optimierungen und Versuche konnte die vom Gesamtsystem erreichbare Bildrate nicht auf die benötigten 100 Bilder pro Sekunde gesteigert werden.

5.2 Ausblick

Diese Diplomarbeit stellt, trotz der nicht erreichten Funktionalität des Gesamtsystems, eine gute Ausgangsbasis für die weitere Entwicklung des Systems für Werfen und Fangen dar, wenn für die Umsetzung eine SoC Plattform gewählt wird. Aktuell kann auch ein funktionierendes System, welches in der Diplomarbeit vom Götzinger entworfen wurde [Göt15], als Basis für das System für Werfen und Fangen herangezogen werden. Dieses existiert in der ergänzten Form,

wo zwei der Graphic Processing Units (GPUs) im System verwendet werden, eine GPU für die Ballmittelpunkt Bestimmung im linken und rechten Bild und eine GPU für die Flugbahnprognose [Göt15, 89]. Dieser Ansatz mit den zwei GPUs wurde gewählt, da im Gegensatz zu dem ursprünglichen Ansatz mit nur einer GPU, die Datenbank mit den gespeicherten Trajektorien für die Flugbahnprognose erweitert wurde. Die Berechnungen mit nur einer GPU würden mit der größeren Datenbank eine längere Zeit in Anspruch nehmen, sodass eine Bildrate von 100 Bilder pro Sekunde nicht mehr erreichbar wäre. Zusätzlich sind die zwei GPUs zum Pipelining, ähnlich wie in dieser Diplomarbeit, gewählt worden (vergleiche Abschnitt 3.5).
Der Vorteil des Systems basierend auf der SoC Plattform, im Gegensatz zu dem System basierend auf GPUs, ist einerseits die deterministische Ausführungszeit, da die Dauer der Ausführung der Berechnungen im FPGA immer ungefähr die gleiche Zeit in Anspruch nimmt und nicht von der aktuellen Auslastung des Systems abhängig ist. Andererseits spielt auch die Leistungsaufnahme des Gesamtsystems eine bedeutende Rolle und auch in diesem Bereich hat die SoC Plattform einen Vorteil im Gegensatz zu dem System basierend auf GPUs. Alleine nur zwei leistungsfähigere GPUs haben eine Leistungsaufnahme von mehreren Hundert Watt. Der SoC hat eine simulierte Leistungsaufnahme von knapp 14 Watt, die gesamte Entwicklungsplattform hat eine gemessene Leistungsaufnahme von 34,7 Watt.
Für das System basierend auf den GPUs sprechen die Anschaffungskosten. Diese sind um einiges niedriger, als die Kosten für die verwendete SoC Plattform. Und natürlich die Funktionalität, die im aktuellen Zustand und mit der aktuellen Generation der SoC Plattform nicht erreichbar ist.

Eine eventuelle Möglichkeit, wie das System für Werfen und Fangen, basierend auf der aktuellen SoC Plattform funktionieren könnte, wäre wenn man die Daten von den Kameras signifikant reduziert. Statt jedesmal die großen 2048x768 Pixel Bilder zu übertragen, könnte die Auflösung der Bilder auf 300x300 Pixel reduziert werden. Im Prinzip würde der linken und rechten Kamera der relevante Bildbereich (AoI) mit dem geworfenen Ball vor der Bildaufnahme mitgeteilt und diese würde nur diesen Bereich über die Universal Serial Bus (USB) 3.0 Schnittstelle senden. Ein Algorithmus würde jeweils die aktuelle AoI in Abhängigkeit von der aktuellen Ballposition im Raum bestimmen. Es würde somit die Suche nach der ungefähren Position des Balls in den aufgenommenen Bildern wegfallen und diese würde nach einem festen Algorithmus bestimmt. Die *Hintergrundsubtraktion* und alle folgenden Verarbeitungsstufen würden wie zuvor vorhanden sein, müssten aber an die neue Bildgröße angepasst werden. Zudem müsste eine neue Software entwickelt werden, die die angepasste AoI den Kameras übermittelt.
An dieser Stelle muss aber erwähnt werden, dass bei dieser Methode der Aufnahme derzeit Probleme auftreten. Aufgrund eines (dokumentierten) Treiber/Hardware-Problems der Kameras kommt es bei der Verschiebung der AoI-Position, zwischen zwei aufeinander folgenden Bildaufnahmen, zu Fehlern. Basierend auf Informationen des Kameraherstellers, kann der Zeitrahmen für das Beheben dieses Problems nicht vorhergesagt werden [Pon16, 103].

Eine weitere Möglichkeit, bei der eventuell das in dieser Diplomarbeit entworfene System funktionieren könnte, wäre die Portierung dieses Systems auf die nachfolgende Generation der SoC Plattform. Diese ist derzeit noch nicht verfügbar, basiert aber auf deutlich leistungsfähigeren CPUs mit vier Kernen und einer Taktfrequenz von jeweils 1,5 *GHz*. Weiters ist auch eine native USB 3.0 Schnittstelle vorhanden, somit ist der Umweg über die USB 3.0 Erweiterungskarte für den Peripheral Component Interconnect Express (PCIe) Steckplatz nicht mehr notwendig. Zudem ist auch die Menge an den zur Verfügung stehenden Ressourcen im FPGA gestiegen, wodurch auch zusätzliche Logik und beispielsweise eine größere Datenbank mit den Trajektorien für die Flugbahnprognose implementiert werden könnte.

Wissenschaftliche Literatur

[ACS08] Alt, Nicolas ; Claus, Christopher ; Stechele, Walter: Hardware/software architecture of an algorithm for vision-based real-time vehicle detection in dark environments, IEEE, 2008, S. 176–181

[BFK08] Barteit, Dennis ; Frank, Heinz ; Kupzog, Friederich: Accurate prediction of interception positions for catching thrown objects in production systems, IEEE, 2008, S. 893–898

[BFPK09] Barteit, Dennis ; Frank, Heinz ; Pongratz, Martin ; Kupzog, Friederich: Measuring the Intersection of a Thrown Object with a Vertical Plane, IEEE, 2009, S. 680–685

[BSW+11] Bäuml, Berthold ; Schmidt, Florian ; Wimböck, Thomas ; Birbach, Oliver ; Dietrich, Alexander ; Fuchs, Matthias ; Friedl, Werner ; Frese, Udo ; Borst, Christoph ; Grebenstein, Markus ; Eiberger, Oliver ; Hirzinger, Gerd: Catching flying balls and preparing coffee: Humanoid Rollin'Justin performs dynamic and sensitive tasks, IEEE, 2011, S. 3443–3444

[BWH10] Bäuml, Berthold ; Wimböck, Thomas ; Hirzinger, Gerd: Kinematically optimal catching a flying ball with a hand-arm-system, IEEE, 2010, S. 2592–2599

[Can86] Canny, John: A Computational Approach to Edge Detection, IEEE, 1986, S. 679–698

[CEES14] Crockett, Louise H. ; Elliot, Ross A. ; Enderwitz, Martin A. ; Stewart, Robert W.: *The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*. 1. Strathclyde Academic Media, 2014

[CMCGSS15] Cerezuela-Mora, Javier ; Calvo-Gallego, Elisa ; Sánchez-Solano, Santiago: Hardware/software co-design of video processing applications on a reconfigurable platform, IEEE, 2015, S. 1694–1699

[DACN02] D'Orazio, Tiziana ; Ancona, Nicola ; Cicirelli, Grazia ; Nitti, Marco: A ball detection algorithm for real soccer image sequences, IEEE, 2002, S. 210–213 vol.1

[FBH+01] Frese, Udo ; Bäuml, Berthold ; Haidacher, S. ; Schreiber, Gerhard ; Schaefer, Ina ; Hahnle, Michael ; Hirzinger, Gerd: Off-the-shelf vision for a robotic ball catcher, IEEE, 2001, S. 1623–1629 vol.3

[FBM+08] Frank, Heinz ; Barteit, Dennis ; Meyer, Marcus ; Mittnacht, Anton ; Novak, Gregor ; Mahlknecht, Stefan: Optimized Control Methods for Capturing Flying Objects with a Cartesian Robot, IEEE, 2008, S. 160–165

[FWWH+06] Frank, Heinz ; Wellerdick-Wojtasik, Norbert ; Hagebeuker, Bianca ; Novak, Gregor ; Mahlknecht, Stefan: Throwing Objects – A bio-inspired Approach for the Transportation of Parts, IEEE, 2006, S. 91–96

[GCAMJ15] Göbel, Matthias ; Chi, Chi C. ; Alvarez-Mesa, Mauricio ; Juurlink, Ben: High Performance Memory Accesses on FPGA-SoCs: A Quantitative Analysis, IEEE, 2015, S. 32

[Göt15] Götzinger, Maximilian: *Object detection and flightpath prediction: a parallelized approach using a graphics processing unit*, Vienna University of Technology, Diplomarbeit, 2015

[GW07] Gonzales, Rafael ; Woods, Richard: *Digital Image Processing*. 3. Prentice Hall, 8 2007. – ISBN 013168728X

[HS91] Hove, Barbara ; Slotine, Jean-Jacques E.: Experiments in Robotic Catching, IEEE, 1991, S. 380–386

[HS95] Hong, Won ; Slotine, Jean-Jacques E.: Experiments in hand-eye coordination using active vision, Springer Berlin Heidelberg, 1995, S. 130–139

[INI96] Ishii, Itaru ; Nakabo, Yoshihiro ; Ishikawa, Mutsuo: Target tracking algorithm for 1 ms visual feedback system using massively parallel processing, IEEE, 1996, S. 2309–2314 vol.3

[KLL+06] Kim, Jisu ; jae Lee, Hyuk ; ho Lee, Tae ; Cho, Myungje ; beom Lee, Jae: Hardware/Software Partitioned Implementation of Real-time Object-oriented Camera for Arbitrary-shaped MPEG-4 Contents, IEEE, 2006, S. 7–12

[Lec16] Lechner, Martin: Bildverarbeitung am FPGA - Realisierung einer Balldetektion mittels High Level Synthese / TU Wien. 2016. – Forschungsbericht

[Mir16] Mironov, Konstantin: *Predicting the Trajectory of the Flying Object with the Use of k-Nearest Neighbors*, Vienna University of Technology, Diss., 2016

[MVP15] Mironov, Konstantin ; Vladimirova, Irina ; Pongratz, Martin: Processing and Forecasting the Trajectory of a Thrown Object Measured by the Stereo Vision System, IFAC-PapersOnLine, 2015, S. 28–35

[PKFB10] Pongratz, Martin ; Kupzog, Friederich ; Frank, Heinz ; Barteit, Dennis: Transport by Throwing - a bio-inspired Approach, IEEE, 2010, S. 685–689

[PMB13] Pongratz, Martin ; Mironov, Konstantin ; Bauer, Friedrich: A Soft-Catching Strategy for Transport by Throwing and Catching, Vestnik UGATU Vol. 17, no. 6 (59), 2013, S. 28–32

[Pon09] Pongratz, Martin: *Object Touchdown Position Prediction: A Stereo Vision Based Approach*, Vienna University of Technology, Diplomarbeit, 2009

[Pon16] Pongratz, Martin: *Bio-inspired Transport by Throwing System*, Vienna University of Technology, Diss., 2016

[PPS12] Pongratz, Martin ; Pollhammer, Klaus ; Szep, Alexander: KOROS Initiative: Automatized Throwing and Catching for Material Transportation, Springer-Verlag Berlin Heidelberg, 2012, S. 136–143

[RFQ03] Rad, Ali A. ; Faez, Karim ; Qaragozlou, Navid: Fast Circle Detection Using Gradient Pair Vectors, Proceedings of 7nd Digital Image Computering: Techniques and Applications., 2003, S. 879–887

[TAD00] Tagzout, Samir ; Achour, Kahina ; Djekoune, Oualid: Hough transform algorithm for FPGA implementation, IEEE, 2000, S. 384–393

[TKBM99] Toyama, Ken ; Krumm, John ; Brumitt, Bern ; Meyers, Bennet: Wallflower: principles and practice of background maintenance, IEEE, 1999, S. 255–261 vol.1

[TSHS13] Tang, Jia W. ; Shaikh-Husin, Nasir ; Sheikh, Usman U.: FPGA implementation of RANSAC algorithm for real-time image geometry estimation, IEEE, 2013, S. 290–294

[TSS15] Tabkhi, Hamed ; Sabbagh, Majid ; Schirner, Gunar: Power-efficient real-time

solution for adaptive vision algorithms, IEEE, 2015, S. 16–26

[XCZB12] HUI XIONG, Zhi ; CHENG, Irene ; JUN ZHANG, Mao ; BASU, Anup: HW/SW co-design of an embedded omni-imaging system, IEEE, 2012, S. 3378–3383

[XVCK14] XU, Qian ; VARADARAJAN, Srenivas ; CHAKRABARTI, Chaitali ; KARAM, Lina: A Distributed Canny Edge Detector: Algorithm and FPGA Implementation, IEEE, 2014, S. 2944–2960

[ZIN13] ZHOU, Xin ; ITO, Yasuaki ; NAKANO, Koji: An Efficient Implementation of the Hough Transform Using DSP Slices and Block RAMs on the FPGA, IEEE, 2013, S. 85–90

[ZIN14a] ZHOU, Xin ; ITO, Yasuaki ; NAKANO, Koji: An Efficient Implementation of the Gradient-based Hough Transform using DSP slices and block RAMs on the FPGA, IEEE, 2014, S. 762–770

[ZIN14b] ZHOU, Xin ; ITO, Yasuaki ; NAKANO, Koji: An Efficient Implementation of the One-Dimensional Hough Transform Algorithm for Circle Detection on the FPGA, IEEE, 2014, S. 447–452

[ZXZC12] ZHANG, Yifeng ; XIONG, Rong ; ZHAO, Yongsheng ; CHU, Jian: An Adaptive Trajectory Prediction Method for Ping-Pong Robots, Springer Berlin Heidelberg, 2012, S. 448–459

Internet Referenzen

[1] Brandon Perez & Jared Choi auf GitHub.com. *Webseite*, Zugriff am 02.August 2016. https://github.com/bperez77/xilinx_axidma.

[2] Department of Computer, Control, and Management Engineering Antonio Ruberti at Sapienza University of Rome. *Webseite*, Zugriff am 27.Juli 2016. http://www.dis.uniroma1.it/~iocchi/stereo/triang.html.

[3] Department of Computer Science and Engineering University at Buffalo. *PDF*, Zugriff am 17.Oktober 2006. http://homes.cs.washington.edu/~todorov/courses/cseP590/readings/tutorialEKF.pdf.

[4] Department of Computer Science University of North Carolina at Chapel Hill. *PDF*, Zugriff am 17.Oktober 2016, Dokument vom 24.Juli 2006. https://www.cs.unc.edu/~welch/media/pdf/kalman_intro.pdf.

[5] Doulos. *Webseite*, Zugriff am 29.Juli 2016. https://www.doulos.com/knowhow/verilog_designers_guide/what_is_verilog/.

[6] Doulos. *Webseite*, Zugriff am 29.Juli 2016. https://www.doulos.com/knowhow/vhdl_designers_guide/what_is_vhdl/.

[7] FPGAcenter.com. *Webseite*, Zugriff am 28.Juli 2016. http://fpgacenter.com/fpga/fpga_or_cpu.php.

[8] IDS Imaging Development Systems GmbH. *Webseite*, Zugriff am 02.August 2016. https://en.ids-imaging.com/download-ueye-emb-hardfloat.html.

[9] Institute of Management Accountants. *Webseite*, Zugriff am 24.Oktober 2016. http://sfmagazine.com/post-entry/december-2015-the-cost-of-manufacturing-disruptions/.

[10] Intorobotics. *Webseite*, Zugriff am 27.Juli 2016. https://www.intorobotics.com/fundamental-guide-for-stereo-vision-cameras-in-robotics-tutorials-and-resources/.

[11] Library at the Universidad Politécnica de Madrid. *PDF*, Zugriff am 31.Juli 2016, Dokument vom Juli 2013. http://oa.upm.es/21488/1/PFC_ALVARO_BUSTOS_BENAYAS.pdf.

[12] Linaro. *Webseite*, Zugriff am 02.August 2016. https://releases.linaro.org/ubuntu/images/developer/15.12/.

[13] Linaro. *Webseite*, Zugriff am 02.August 2016. https://releases.linaro.org/15.02/components/toolchain/binaries/arm-linux-gnueabihf/.

[14] The MathWorks, Inc. *Webseite*, Zugriff am 23.Juli 2016. http://www.mathworks.com/matlabcentral/fileexchange/40737-canny-edge-detector.

[15] Mohammad S. Sadri. *PDF*, Zugriff am 31.Juli 2016, Dokument vom September 2013. http://www.googoolia.com/downloads/papers/sadri_fpgaworld_ver2.pdf.

[16] National Instruments Corporation. *Webseite*, Zugriff am 8.Januar 2016. http://www.ni.com/white-paper/6984/en/.

[17] Quest TechnoMarketing e.K. *Webseite*, Zugriff am 13.Dezember 2015. http://www.quest-trendmagazin.de/artikel-archiv/einsatz-von-robotern-steigt-2011.html.

[18] StarTech.com. *Webseite*, Zugriff am 02.August 2016. https://www.startech.com/Cards-Adapters/USB-3.0/Cards/PCI-Express-USB-3-Card-4-Dedicated-Channels-4-Port~PEXUSB3S44V.

[19] XILINX. *PDF*, DS190 (v1.9), Zugriff am 31.Juli 2016, Dokument vom 20.Januar 2016. http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.

[20] Xilinx Inc. *PDF*, Zugriff am 31.Juli 2016, Dokument vom 2.Juli 2013. http://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf.

[21] Xilinx Inc. *PDF*, Zugriff am 06.August 2016, Dokument vom 24.Juni 2015. http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf.

[22] Xilinx Inc. *PDF*, Zugriff am 29.Juli 2016. http://www.xilinx.com/support/documentation/white_papers/demystifying-accelerated-smart-vision-systems-with-all-programmable-socs.pdf.

[23] Xilinx Inc. *PDF*, Zugriff am 06.Oktober 2016, Dokument vom 27.September 2016. http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

[24] Xilinx Inc. *PDF*, Zugriff am 15.September 2016, Dokument vom 08.Juni 2016. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug902-vivado-high-level-synthesis.pdf.

[25] Xilinx Inc. *PDF*, Zugriff am 17.September 2016, Dokument vom 08.Juni 2016. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug904-vivado-implementation.pdf.

[26] Xilinx Inc. *Webseite*, Zugriff am 6.Januar 2016. http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm.

[27] Xilinx Inc. *Webseite*, Zugriff am 29.Juli 2016. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[28] ZedBoard. *PDF*, Zugriff am 25.Juli 2016. http://zedboard.org/sites/default/files/product_briefs/PB-AES-MINI-ITX-G-v11.pdf.

[29] ZedBoard. *Webseite*, Zugriff am 25.Juli 2016. http://zedboard.org/product/mini-itx.

[30] ZedBoard. *Webseite*, Zugriff am 20.Oktober 2016. http://zedboard.org/content/pcie-problems-when-using-usb-30-cards.

[31] ZEIT ONLINE GmbH. *Webseite*, Zugriff am 12.Dezember 2015. http://www.zeit.de/auto/2013-04/ford-fliessband-massenproduktion.

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Wien, am 11.November 2016 _____

Andrej Hanic BSc.