technische universität
dortmund

Masterarbeit

Online Gauß-Prozesse zur Regression auf FPGAs

Sebastian Buschjäger (B Sc.)

18. Januar 2016

Betreuer: Prof. Dr. Katharina Morik
Dipl.-Inform. Nico Piatowski

Fakultät für Informatik
Lehrstuhl für künstliche Intelligenz (LS8)
TU Dortmund

Zusammenfassung

Mit zunehmender Integration informationsverarbeitender Systeme in alltägliche Bereiche des Lebens wächst die Anzahl erhobener Daten stetig an. Zusätzlich verlangt diese zunehmende Integration eingebetteter Systeme in den Alltag nach immer energiesparenderen Systemen. FPGAs bieten als programmierbare Schaltkreise eine sowohl energiesparende, als auch frei programmiere Systemarchitektur an, sodass ihre Verwendung in eingebetteten Systemen nahe liegt.
Um Informationen aus den erhobenen Daten zu extrahieren, können Verfahren des maschinellen Lernens benutzt werden. Damit ist eine Verbindung von maschinellen Lernverfahren und FPGAs ebenfalls naheliegend.
Die vorliegende Arbeit implementiert beispielhaft einen Gauß-Prozess auf einem FPGA und geht dabei auf verschiedene Aspekte des Systementwurfs ein. Hierzu wird zunächst ein vom Lernalgorithmus unabhängiges System auf Basis eines Soft-Prozessors entwickelt. Dieser Soft-Prozessor führt einen minimalen TCP/IP Protokollstapel aus, um die Daten einer Gegenstellen entgegenzunehmen. Anschließend wird die Implementierung des Lernalgorithmus mit Hilfe der High Level Synthese vorgestellt und einige Optimierungen diskutiert, sodass schließlich eine Verbindung beider Hardwareeinheiten über eine gemeinsame Schnittstelle erfolgen kann.
Die hier vorgestellte Systemarchitektur ist zunächst unabhängig vom Lernalgorithmus, sodass dieser einfach ausgetauscht werden kann. Zusätzlich zeigt sich, dass die hier vorgestellte Hardwareimplementierung eine zu Hardware aus dem Bereich der eingebetteten Systeme vergleichbaren Geschwindigkeit liefert.
Der Energieverbrauch des umgesetzten Systems konnte durch passende Systemkonfiguration und Anpassung des TCP/IP Protokollstapels im Vergleich zu eingebetteter Systemhardware halbiert werden, was insgesamt zu einer erhöhten Effizienz führt.

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Hiermit bestätige ich, die vorliegende Masterarbeit selbständig und nur unter Zuhilfenahme der angegebenen Literatur verfasst zu haben.

Dortmund, den 18. Januar 2016

Sebastian Buschjäger

Inhaltsverzeichnis

1 Einführung **1**

1.1 Verwandte Arbeiten . . . 2

1.2 Ziel dieser Arbeit . . . 2

1.3 Struktur dieser Arbeit . . . 3

2 Systementwurf **5**

3 Field Programmable Gate Arrays **9**

3.1 Aufbau . . . 11

3.1.1 Signalpfadkonfiguration . . . 11

3.1.2 Logikkonfigurierung . . . 11

3.1.3 Peripherie . . . 12

3.2 Programmierung . . . 13

3.2.1 Hardware Description Language (HDL) . . . 13

3.2.2 High Level Synthesis (HLS) . . . 14

3.2.3 Konfiguration . . . 14

3.2.4 Synthese . . . 15

3.2.5 Place & Route . . . 15

3.2.6 Partielle Rekonfiguration . . . 16

3.3 Diskussion . . . 16

4 Gauß-Prozesse **19**

4.1 Grundbegriffe des maschinellen Lernens . . . 19

4.2 Grundbegriffe der Bayes-Statistik . . . 21

4.3 Vollständige Gauß-Prozesse . . . 22

4.3.1 Gauß-Prozesse zur Regression . . . 24

4.3.2 Gauß-Prozesse zur Klassifikation . . . 25

4.4 Anwendung von Gauß-Prozessen . . . 26

4.5 Diskussion . . . 28

5 Online Gauß-Prozesse 29
5.1 Approximative Gauß-Prozesse 29
5.2 Online Gauß-Prozesse 30
5.2.1 Projection GP 31
5.3 Diskussion 37

6 Implementierung 39
6.1 Implementierung in einer Hochsprache 39
6.2 FPGA Systementwurf 42
6.2.1 Verwendete Hard- und Software 43
6.2.2 Systemarchitektur 44
6.2.3 Diskussion 54
6.3 FPGA Implementierung 57
6.3.1 Projection GP Implementierung 57
6.3.2 Optimierung des Durchsatzes 61
6.3.3 Diskussion 70

7 Experimente 75
7.1 Datensätze 75
7.2 Verwendete Architekturen 76
7.3 Vorhersagegenauigkeit 78
7.4 Durchsatz 79
7.5 Energieverbrauch und Energieeffizienz 82
7.6 Diskussion 85

8 Zusammenfassung und Ausblick 91

Anhang A Eigenschaften der Normalverteilung 93

Anhang B Blockschaltbild der FPGA Systemarchitektur 100

Abbildungsverzeichnis 101

Tabellenverzeichnis 102

Literaturverzeichnis 103

1 Einführung

Mit zunehmender Integration informationsverarbeitender Systeme in allen Bereichen des alltägliche Lebens wächst die Menge erhobener Daten stetig an. Lange konnte dem stetigen Wachstum an Daten ein stetiges Wachstum an Rechenkapazität entgegengesetzt werden, sodass eine Verarbeitung größerer Datenmengen mit neueren Prozessorgenerationen möglich wurde. Seit einiger Zeit stößt die Verarbeitungstechnik jedoch zunehmend an physikalische Grenzen. Eine weitere Erhöhung der Rechenkapazität ist kaum noch durch eine höhere Transistordichte zu erreichen [HP11]. Hieran schließt sich ein Bedarf nach immer energieeffizienteren Prozessoren an, um der Vision von ubiquitären Systemen näher zu kommen [Wei91], sodass ein Trend zu heterogenen Mehrprozessorsystemen zu beobachten ist [HP11].
Damit wird zusätzlich zu klassischen CPUs auch zunehmend Hardware aus dem Bereich der eingebetteten Systeme wie Mikrocontroller oder Spezialhardware in Form von Signalprozessoren eingesetzt (vgl. [JR13]). Des Weiteren sind neben General Purpose Computation on Graphics Processing Unit (GPGPU) auch Field Programmable Gate Arrays (FPGAs) als Hardwarearchitekturen im industriellen Markt angekommen [WL08, BRS13].
Um die großen Datenmengen sinnvoll zu nutzen, sollte die Datenanalyse nicht nur im Nachhinein und außerhalb der Systemnutzung erfolgen, sondern vor allem direkt beim Benutzer, während dieser ein System verwendet [RAS08]. Hier haben sich insbesondere Algorithmen des maschinellen Lernens bewährt, die mittels statistischer Methoden Vorhersagen aus den Daten ableiten können [Sad11].
Eingebettete Systeme machen normalerweise Gebrauch von einfachen integrierten Schaltungen, Signalprozessoren und Mikrocontrollern, deren Rechenkapazität nur schwer mit der Menge der erhobenen Daten skalieren. Übliche Standardhardware lässt sich aufgrund ihres vergleichsweise hohen Energieverbrauches nur schwierig in eingebetteten Systemen anwenden, sodass hier für komplexe Aufgaben vor allem anwendungsspezifische integrierte Schaltkreise genutzt werden [Mar10]. Integrierte Schaltkreise haben jedoch neben hohen Produktionskosten eine rein statische Funktionalität, die sich während der Anwendung nicht auf Änderungen in der Umgebung anpassen lässt. Hier bieten FPGAs als programmierbare Schaltkreise eine schnelle, dynamische und gleichzeitig energiesparende Ausführungsplattform.

Betrachtet man Aufgabenstellungen von eingebetteten Systeme, so müssen häufig Problemstellungen der Lokalisierung, sowie der inversen Kinematik gelöst werden. Hier haben sich maschinelle Lernverfahren und insbesondere Gauß-Prozesse bewährt, sodass sie ein entsprechendes Anwendungsfeld aufweisen [GMHP04, FHF06, FFL07]. Eine Verknüpfung von FPGAs mit maschinellen Lernverfahren und insbesondere Gauß-Prozessen scheint also naheliegend.

1.1 Verwandte Arbeiten

Bedingt durch die zunehmende Erhebung großer Datenmengen in Kombination mit komplexen Algorithmen erscheint die Verwendung spezifischer Hardware als sinnvoll. Die Anwendbarkeit von Maschinellen Lernverfahren auf FPGAs wurde bereits in einigen Arbeiten untersucht.

Narayanan et al. stellen in [NHM+07] eine Implementierung von Entscheidungsbäumen auf FPGAs mit einer Beschleunigung um einen Faktor von ca. $1,5$ im Vergleich zu Standardhardware vor. Hussain et al. erreichen in [HBSE11] eine bis zu 51 mal schnellere und gleichzeitig 200 mal energieeffizientere Berechnung eines Clusterings mittels K-Means auf FPGAs. In [SWY+10] stellen die Autoren ein MapReduce Framework für FPGAs vor, welches den PageRang Algorithmus ca. 31 mal schneller als eine klassische Implementierung ausführt.

In [PB08a, PB08b] beschreiben die Autoren eine effiziente Implementierung der SVM auf FPGAs, die vor allem auf einer schnellen Auswertung der Kernmatrix beruht. Dabei gehen die Autoren insbesondere auch auf hardwarespezifische Rundungsfehler durch Festkommazahlen ein. Irick et al. stellen in [IDNG08] eine schnelle Anwendung der SVM zur Bildklassifikation in Filmen vor, die nach Angaben der Autoren erstmals zum damaligen Zeitpunkt eine Klassifikation in Echtzeit erlaubte.

Des Weiteren findet sich eine Fülle verschiedener Arbeiten zur Umsetzung von neuronalen Netzen auf FPGAs (siehe z.B. [WHTS14, MS10, OR06, ZS03]), die zum Teil schon eine gewisse Marktreife erreicht haben [Xilj, Auv]. Diese Arbeiten unterscheiden sich je nach verwendetem FPGA und der betrachteten Anwendung mehr oder weniger stark. Hier werden Geschwindigkeitsverbesserungen mit einem Faktor von bis zu 75 erreicht [MS10].

1.2 Ziel dieser Arbeit

FPGAs können als eine schnelle und energiesparende Ausführungsplattform genutzt werden, welche jedoch keinerlei Laufzeitumgebung für Dateiabstraktionen oder Peripheriezugriffe anbietet. Aus diesem Grund muss neben der eigentlichen Implementierung auch der Entwurf des umliegenden Systems erfolgen. Dieser Systementwurf hat sich mit der dritten Generation der verfügbaren Werkzeuguntersützung für FPGAs stark gewandelt

[CLN+11, BRS13], wodurch sich Unterschiede zu der vorhandenen Literatur ergeben.
Das Entwurfsvorgehen für die aktuelle FPGA- und Werkzeuggeneration soll zunächst vorgestellt werden, um darauf aufbauend eine passende Laufzeitumgebung für maschinelle Lernalgorithmen auf dem FPGA zu entwerfen. Hierbei soll eine möglichst modulare und energiesparende Systemarchitektur entworfen werden, sodass sich die hier vorgestellte Systemarchitektur gut in eingebettete System anwenden lässt und zusätzlich der maschinelle Lernalgorithmus wegen der Modularität des Systems einfach ausgetauscht werden kann.
Anschließend soll eine beispielhafte Umsetzung eines Gauß-Prozesses auf dem FPGA die Einbettung in das Gesamtsystem zeigen, wobei hier Wert auf eine möglichst hohe Geschwindigkeit der Hardwareimplementierung gelegt werden soll.
Die Umsetzung einer energiesparenden Systemarchitektur für verschiedene maschinelle Lernalgorithmen ist nach Wissen des Autors neu, da in der vorhandenen Literatur jeweils ein neues System für einen anderen Algorithmus entworfen wird. Des Weiteren ist Umsetzung von Gauß-Prozessen auf FPGAs ist nach Wissen des Autors ebenfalls neu, sodass ich hier weitere Unterschiede zur vorhanden Literatur ergeben.

1.3 Struktur dieser Arbeit

Die vorliegende Arbeit ist wie folgt organisiert. Kapitel 2 beschriebt ein mögliches Entwurfsvorgehen für technische Systeme, die im Kern einen maschinellen Lernalgorithmus verwenden. Dieses Vorgehen wird im Verlaufe der Arbeit verwendet, um die Einflüsse der gewählten Methode und Systemarchitektur auf die anschließende Implementierung deutlich zu machen.
Kapitel 3 beschreibt die Funktionsweise von FPGAs und geht dabei auf die Programmierung von FPGAs ein. Hier wird insbesondere auf die aktuelle Werkzeugunterstützung und die damit verbundenen Änderungen im Entwurfsprozess eingegangen.
Danach stellt Kapitel 4 die Grundlagen des maschinellen Lernens vor und präsentiert ausführlich Gauß-Prozesse. Anschließend beschreibt Kapitel 5 verschiedene Typen von Gauß-Prozessen, welche kritisch gegeneinander abgewogen werden, um so eine geeignete Auswahl für die anschließende Implementierung auf FPGAs zu treffen. Kapitel 6 geht detailliert auf die Implementierung der FPGA Systemarchitektur ein und beschrieb insbesondere Techniken zum reduzieren des Energieverbrauches und Optimierung des Durchsatzes des maschinellen Lernalgorithmus.
In Kapitel 7 werden Experimente zum Durchsatz, zur Vorhersagegüte und zum Energieverbrauch der vorgestellten Implementierung gezeigt, sodass Kapitel 8 mit einer Diskussion dieser Ergebnisse die vorliegende Arbeit abschließt.

2 | Systementwurf

Wie bereits erwähnt, stehen Entwicklern mit CPUs, GPGPUs, FPGAs sowie weiterer Spezialhardware eine Vielzahl verschiedener Systemarchitekturen zur Verfügung. Bei dieser großen Vielfalt ist es häufig schwierig eine passende Systemarchitektur für ein gegebenes technische Problem zu wählen.
Tatsächlich müssen die Vor- und Nachteile einzelner Plattformen immer auf Basis der gewünschten Anwendung evaluiert werden.
Im Fachgebiet des Systems Engineering wurden deshalb eine Vielzahl verschiedener Vorgehensmodelle präsentiert, die Systementwickler bei ihren Entscheidungen unterstützen sollen. Diese Vorgehensmodelle beinhalten neben dem eigentlichen Systementwurf auch Teilschritte zum Risikomanagement, einer Marktanalyse sowie der Systemüberwachung (vgl. [Bue11]), gehen aber nicht detailliert auf die Beziehungen zwischen Hardware, Software und der zu implementierenden mathematischen Methoden ein.
Das `CRISP-DM` Modell aus dem Bereich des Data Mining [WH00] nimmt hier explizit die mathematischen Methoden in den Analyse- und Entwurfsprozess mit auf, geht jedoch ebenfalls nicht auf die besondere Beziehung zwischen Hard- und Software ein.
Der Teilbereich des Hard- und Software Codesigns hingegen modelliert die Beziehungen zwischen Hard- und Software sehr exakt, abstrahiert jedoch von der zu implementierenden mathematischen Methode (vgl. [Ern98, Tei12]).
Es zeigt sich zunehmend, dass die zur Verfügung stehenden Systemarchitekturen verschiedene Entwurfsphilosophien verfolgen, wodurch sie sich zum Teil besser für bestimmte Aufgabenstellungen und mathematische Methoden eignen (siehe z.B: [THL09, LKC+10]). Zum Entwurf eines effizienten Systems muss also nicht nur die Interaktion von Hard- und Software miteinander betrachtet werden, sondern auch der Einfluss der verwendeten mathematischen Methode modelliert werden.
Um den Rahmen der vorliegenden Arbeit nicht zu sprengen und dennoch einen transparenten Entwurf zu erlauben, der sowohl auf die mathematische Methodik, als auch auf die Implementierung in Hard- und Software eingeht, wird im Laufe dieser Arbeit der in Abbildung 2.1 gezeigte Entwurfsprozess verwendet.

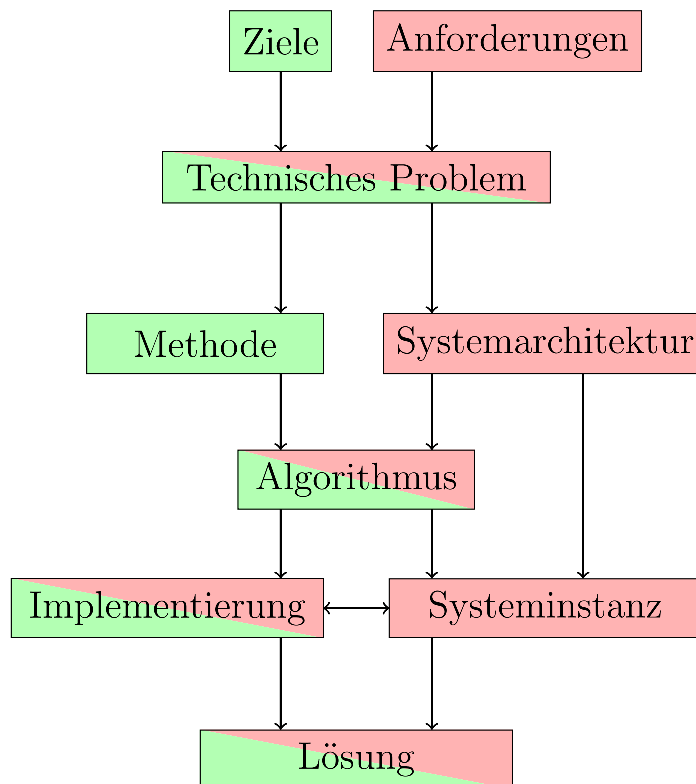


Abbildung 2.1: Schematisches Vorgehen bei der Systementwicklung im Hinblick auf das Zusammenspiel von Systemarchitektur und mathematischer Methode, sowie dem implementierten Algorithmus.

Ziele und Anforderungen

Zunächst müssen sowohl Ziele, als auch Anforderungen an ein technisches System definiert werden. Die Ziele werden im Kontext des maschinellen Lernens als ein maschinelles Lernproblem wie Klassifikation, Regression oder Clustering (vgl. Abschnitt 4.1) formuliert. Dem gegenüber beschreiben die Anforderungen die eigentlichen Systemeigenschaften wie z.B. die zur Verfügung stehende Hardware, die Betriebskosten oder den Energieverbrauch.

Technisches Problem

Ziele und Anforderungen werden zusammen in einem technischem Problem formuliert, sodass eine formale Basis für weitere Entscheidungsprozesse besteht. An dieser Stelle muss insbesondere eine klare Formulierung der Ziele erfolgen und einzelne Anforderungen nach ihrer Wichtigkeit sortiert werden.

Methode

Aus den Zielen leitet sich direkt die Menge an möglichen mathematischen Methoden ab, die das Erreichen dieser Ziele ermöglichen. Möchte man z.B. ein Regressionsproblem lösen, so können entsprechende Methoden wie z.B: Gauß-Prozesse (vgl. Abschnitt 4.3) oder z.B. eine Support-Vector-Regression [SS04] betrachtet werden. Diese Auswahl ist nicht auf eine einzige Methode beschränkt, sondern kann zunächst eine Menge verschiedener Methoden darstellen, die im Verlaufe des Entwurfsprozesses verfeinert wird.

Systemarchitektur

Aus den Anforderungen leitet sich im Wesentlichen die Systemarchitektur ab. Diese kann von eingebetteten Systemen mit FPGAs, über Desktopsysteme mit GPGPUs bis hin zu großen Serversystemen mit einer Lambda Architektur [Boc15] sowohl mit verteilten, als auch mit zentralisierten Systemkomponenten reichen. Hier muss insbesondere neben der eigentlichen Datenanalyse die Integration des Systems in den Produktiveinsatz mit modelliert werden.
Als einfaches Beispiel sei hier die Kaggle ECML/PKDD 2015 Taxi Prediction Challange [Kag] genannt, in der es Ziel ist, die Fahrzeiten und die Endposition von Taxen in Porto vorherzusagen. Die hier benutze Systemarchitektur verwendet eine Kombination von verteilten, eingebetteten Systemen mit GPS Empfängern in den einzelnen Taxen, deren GPS Daten mit einem zentralen Server aggregiert werden. Die anschließende Analyse der Daten erfolgte dann lokal bei den Teilnehmern der Challange mit ihrer eigenen Hardware.

Algorithmus

Aus Sicht der Methode gibt es oft eine Menge verschiedener Algorithmen, die diese Methode umsetzen. Soll z.B: ein Regressionsproblem mit der Methode der Gauß-Prozessen gelöst werden, so gibt es neben vollständigen Gauß-Prozessen auch eine Vielzahl verschiedener Approximationen (siehe Kapitel 5), die sich durch unterschiedliche Vor- und Nachteile jeweils besser oder schlechter für eine gegebene Architektur eignen. Im Falle von GPGPUs z.B. werben Hersteller mit einem hohen Durchsatz bei gegebener Datenparallelität [nvi], weshalb die Wahl des Algorithmus dieses berücksichtigen sollte.
Damit muss der verwendete Algorithmus anhand der Systemarchitektur und der gewünschten Methode ausgewählt werden.

Systeminstanz

Die Systeminstanz beschreibt die tatsächliche Umsetzung der Systemarchitektur mit echter Hardware. Hier müssen konkrete Entwurfsentscheidungen betreffend dem Prozessormodell, der Speichergröße oder dem GPGPU Modell getroffen werden.

Zusätzlich muss hier die Ausführungsumgebung des Algorithmus wie z.B: die verwendete Java Virtual Machine ausgewählt werden und die Entwickler müssen sich auf externe Bibliotheken, Werkzeuge etc. einigen.
Zusätzlich stellt der Algorithmus selbst bestimmte Anforderungen an die Systeminstanz, die sich in der Implementierung entsprechend wiederfinden. Als einfaches Beispiel seien hier Datenstrukturen wie Hash-Maps oder Binärbäume genannt, die zentraler Bestandteil eines Algorithmus sein können und von der Systeminstanz unterstützt werden müssen.

Implementierung

Die eigentliche Implementierung setzt den Algorithmus für die konkrete Systeminstanz um. Die Systeminstanz definiert dabei wesentliche Teile diese Implementierung durch die verwendete Ausführungsumgebung und die verwendeten externen Bibliotheken.
Durch die Implementierung für eine bestimmte Systeminstanz lassen sich sowohl Rückschlüsse zur Optimierung des Algorithmus selbst, als auch auf instanzspezifische Optimierungen, die unabhängig vom Algorithmus sind, durchführen. Als Beispiel sei an dieser Stelle die Verwendung von bestimmten Assembler Instruktionen oder die Anpassung des Speicherlayouts auf Cachegrößen genannt (siehe z.B. [WPB+09, KCS+10]). Dabei kann die Optimierung auf Cachegrößen und bestimmten Assemblerbefehlen direkt in der Implementierung erfolgen und zusätzlich in den Entwurf des Algorithmus einfließen, indem z.B. Datenstrukturen die Cachegrößen berücksichtigen oder die Verwendung bestimmter Assembler Instruktionen erleichtern.

3 | Field Programmable Gate Arrays

Informationsverarbeitende Systeme bestehen aus Hard- und Software, die in einem komplexen Zusammenspiel die Gesamtfunktionalität bereitstellen. Hardware bezeichnet hierbei die physikalische Ausführungsplattform wie z.B: eine handelsübliche CPU, wohingegen Software die eigentliche ausgeführte Funktionalität beschreibt (vgl. [PH05]).
Die Ausführungsplattform ist statisch, d.h. ihre physikalischen Eigenschaften lassen sich nach der Herstellung nicht mehr ändern. Software auf der anderen Seite bildet zunächst eine abstrakte, nicht physikalische Einheit, die unabhängig von der Hardware existiert. Die Systemfunktionalität kann so durch Austausch der Software geändert werden, ohne die eigentlichen physikalischen Bausteine zu modifizieren [PH05].
Je nach Grad des Zusammenspiels von Hard- und Software ergeben sich verschiedene Architekturen mit verschiedenen Eigenschaften (vgl. Abbildung 3.1). Anwendungsspezifische integrierte Schaltungen (ASICs) kommen ohne jegliche Software aus, d.h. die gesamte Funktionalität ist bereits in Hardware kodiert. Das so entstehende System ist für genau eine Funktionalität ausgelegt, wodurch der resultierende Schaltkreis üblicherweise extrem schnell und stromsparend ist [HD08]. Dem gegenüber steht die Tatsache, dass die Produktionskosten für ASICs vergleichsweise hoch sind, weil für eine Änderung der Funktionalität die Schaltung vollständig neu entwickelt werden muss.
General Purpose Central Processing Units (CPUs) und General Purpose Computation on Graphics Processing Units (GPGPUs) hingegen kodieren keinerlei Funktionalität in Hardware, sondern stellen lediglich eine Ausführungsplattform mit einem definierten Befehlssatz und standardisierten Schnittstellen zur Verfügung. Die Software kodiert die eigentliche Funktionalität als eine Folge von Befehlen des Befehlssatzes, welche dann zur Laufzeit des Systems interpretiert und ausgeführt wird. Somit entscheidet die Software erst zur Laufzeit des Systems über die eigentliche Funktionalität. Diese Dynamik birgt einen gewissen Verwaltungs- und Dekodierungsmehraufwand, sodass CPU/GPGPUs langsamer und weniger energieeffizient sind als ASICs. Dem Gegenüber sind die Kosten für CPU/GPGPUs geringer, da hier lediglich die Software zur Änderung der Funktionalität ausgetauscht werden muss.

ASICs und CPU/GPGPUs bilden die beiden Extreme in der aktuellen Hardwarelandschaft: ASICs sind statisch, aber sehr schnell und effizient, wohingegen CPU/GPGPUs dynamisch aber langsamer und weniger energieeffizient sind [HD08].
Field Programmable Gate Arrays (FPGAs) sind nun in der Mitte dieser beiden Extreme anzusiedeln und versuchen das Beste von beiden Seiten zu kombinieren. FPGAs sind programmierbare integrierte Schaltkreise, d.h. ihre Funktionalität wird zwar vor der Benutzung fest in Hardware kodiert, jedoch lässt sich diese Funktionalität zu einem späteren Zeitpunkt und teilweise sogar während der Ausführung wieder ändern. Dadurch sind FPGAs energieeffizienter als CPU/GPGPUs und gleichzeitig dynamischer als ASICs [HD08].
Im Folgenden soll die Arbeitsweise von FPGAs anhand von [HD08] erklärt werden. Zusätzlich bieten Hersteller wie `Xilinx` unter [Xill, Meh, Sun] bzw. `Altera` unter [Alt, Moo07] entsprechende Schriften zu ihren FPGAs an.
Abschnitt 3.1 geht dabei auf die Architektur eines FPGAs ein, wohingegen Abschnitt 3.2 den Entwurf von Systemen mit der aktuellen FPGA Generation vorstellt. Abschließend wird in Abschnitt 3.3 kurz auf die Unterschiede im Entwurfsvorgehen zu früheren FPGA Generationen erläutert.

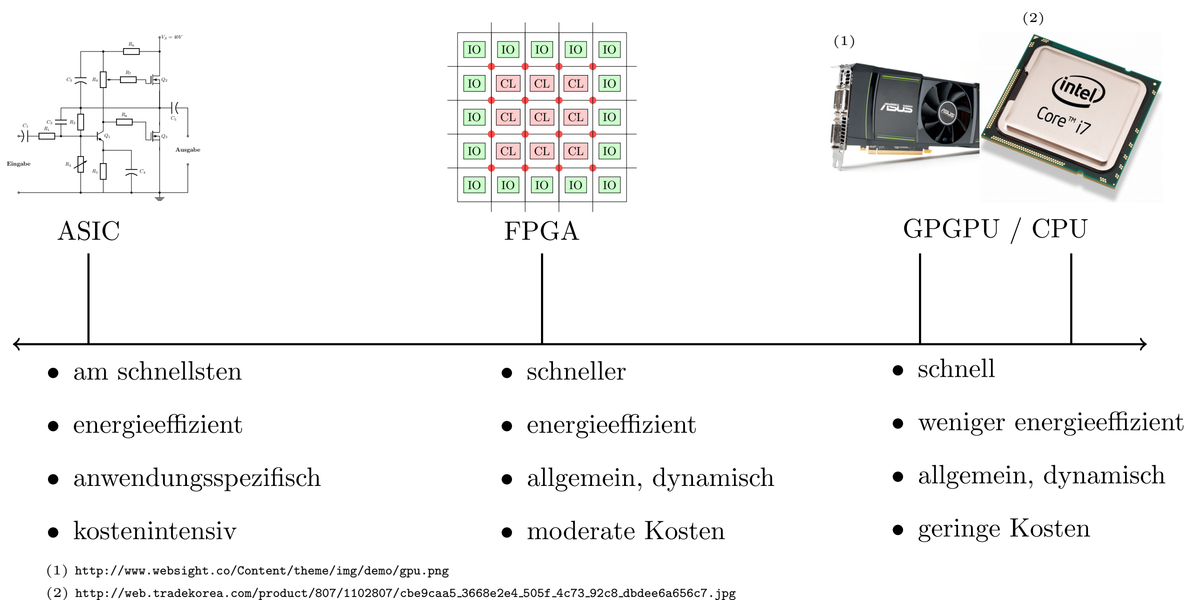


Abbildung 3.1: Hardwareüberblick: Ganz links befinden sich schnelle und energieeffiziente anwendungsspezifische Hardwareeinheiten. Weiter rechts auf der Achse wird die Hardware vergleichsweise langsamer und weniger energieeffizient, jedoch auch für einen allgemeineren Anwendungszweck nutzbar.

3.1 Aufbau

FPGA Chips bestehen in ihrem Kern aus einem zweidimensionalen Gitter von konfigurierbaren Logikblöcken (CL), Ein- und Ausgabeblöcken am Rande des Chips (IO) sowie konfigurierbaren Signalpfaden zwischen den einzelnen Blöcken (siehe Abbildung 3.2 und Abbildung 3.3). Die Ein- und Ausgabeblöcke sind üblicherweise nicht konfigurierbar und variieren je nach Hersteller und umgebender Hardware mehr oder weniger stark. Sie dienen zur Kommunikation mit externen Komponenten wie z.B. zusätzlichem Speicher oder zusätzlichen Schnittstellen wie Ethernet oder PCIe.
Die konfigurierbaren Logikblöcke bilden das Herzstück der Konfigurierbarkeit eines FPGAs. Sie können eine beliebige logische Funktion fester Größe abbilden, welche durch eine vorherige Belegung der Wahrheitstabelle einprogrammiert wird. Diese Logikblöcke werden durch passende Konfiguration der Signalpfade in geeigneter Weise verbunden, um schließlich beliebige logische Funktionen abzubilden. FPGAs sind damit funktional vollständig.

3.1.1 Signalpfadkonfiguration

Abbildung 3.2 zeigt eine schematische Darstellung der Signalpfadkonfiguration. Die Logikblöcke sind mit festen Signalpfaden verbunden, die sich durch die gesamte Gitterstruktur des FPGA Chips erstrecken. An jeder Pfadkreuzung sind Transistoren angebracht, die die entsprechende Route freischalten oder blockieren. Die Konfiguration dieser Transistoren wird über einen Ein-Bit Speicher realisiert, sodass die Freischaltung eines einzelnen Signalpfades dem Setzen des entsprechenden Bits gleichkommt.
In Abbildung 3.2 sind insgesamt Sechs Transistoren (rote Kreise) abgebildet, die alle mögliche Pfade einer Kreuzung konfigurieren. In der aktuellen FPGA Generation gibt es häufig eine höhere Anzahl an Signalpfaden, sodass die Anzahl der Transistoren pro Kreuzung wesentlich höher sein kann. Des Weiteren haben einige Hersteller sogenannte Fastlanes eingeführt, die nur an wenigen Stellen konfigurierbar sind, dafür jedoch ohne Verzögerung an den Kreuzungen auskommen.

3.1.2 Logikkonfigurierung

Abbildung 3.3 zeigt eine schematische Darstellung der konfigurierbaren Logikblöcke. Ein Logikblock kann die Funktion eines Ein-Bit Speichers in Form eines D-Flipflops oder die Funktion einer Logikeinheit mit fester Wahrheitstabelle übernehmen. Ein einfacher Multiplexer, welcher wiederum mittels eines Ein-Bit Speichers gesteuert wird, entscheidet über die gewünschte Programmierung.
Die Wahrheitstabelle wird in Form einer einfachen Look-Up Tabelle gespeichert, sodass einfache logische Funktionen direkt einprogrammiert werden können.

Wird die Logikeinheit als Speichereinheit genutzt, so muss die Wahrheitstabelle nicht beschrieben werden und die Signale werden direkt an das D-Flipflop weiter geleitet.

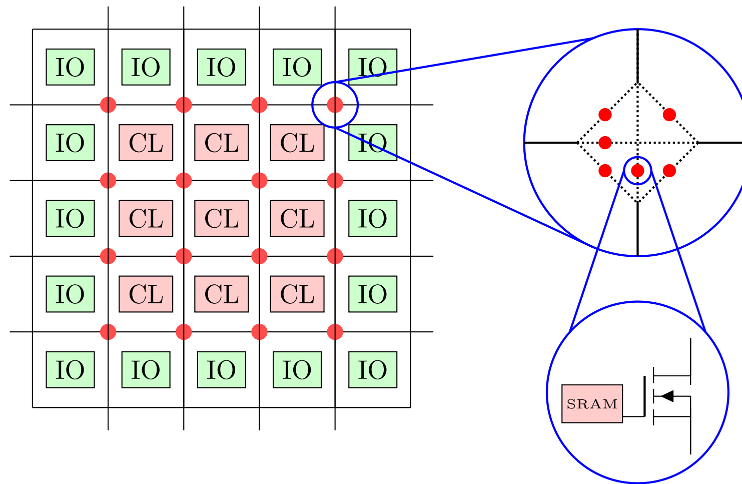


Abbildung 3.2: FPGA: Signalpfadkonfiguration. Signalrouten sind durch feste Pfade vorgegeben, lediglich Pfadkreuzungen sind frei konfigurierbar. Hier ist eine einfache Kreuzung mit 6 verschiedenen Teilpfaden abgebildet.

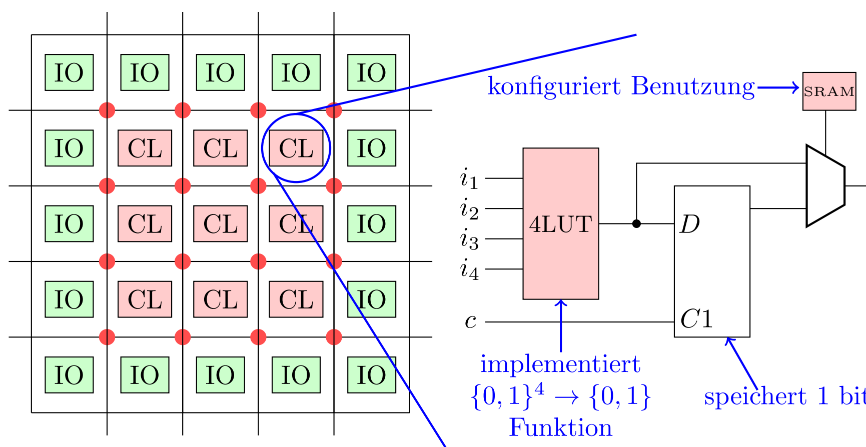


Abbildung 3.3: FPGA: Konfigurierbare Logikblöcke. Jeder Logikblock kann entweder als Ein-Bit Speicher oder als ein Logikblock konfiguriert werden. Der hier abgebildete Block implementiert eine beliebige boolsche Funktion mit 4 Eingangsvariablen $i_1, \ldots, i_4$. Zusätzlich wird ein Taktsignal c zum Ansteuern des Flipflops verwendet.

3.1.3 Peripherie

Prinzipiell lässt sich mit einem FPGA jegliche Schaltung herstellen, die die Anzahl verfügbaren der Logikblöcke nicht überschreitet. Um nun nicht einen Großteil der Logikblöcke für Standardstrukturen wie Speicher oder Arithmetikeinheiten zu benutzen, haben viele

Hersteller zusätzlich diese Strukturen fest als Hardwareschaltungen in ihre Chips integriert.
So bieten aktuelle FPGAs zusätzliche zu den konfigurierbaren Logikblöcken feste Speicherteile in Form von Blockram (BRAM) und Teile digitaler Signalprozessoren als sogenannte DSP-Slices an. Diese zusätzlichen Hardwareeinheiten können beim Entwurf und der Programmierung berücksichtigt werden, um so Logikzellen zu sparen.

3.2 Programmierung

Die Funktionalität eines FPGAs wird durch die Konfiguration der Signalpfade und der verwendeten Wahrheitstabellen in den einzelnen Logikblöcken beschrieben.
Um komplexe Funktionalitäten mit einem FPGA umzusetzen, wird diese Konfiguration nicht direkt durchgeführt, sondern automatisiert aus einer höheren Beschreibungssprache übersetzt. Abbildung 3.4 zeigt schematisch das Vorgehen bei der Programmierung und der anschließenden Generierung des Bitstroms, mit welchem das FPGA konfiguriert wird.

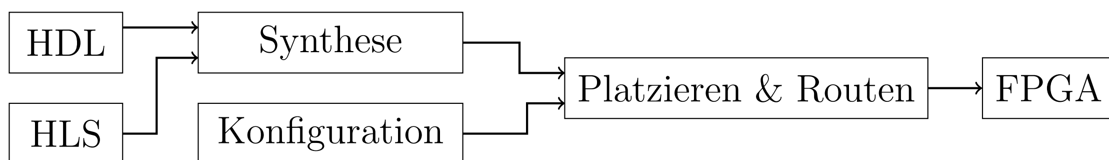


Abbildung 3.4: FPGA: Programmierung. Schematische Darstellung der Programmierung eines FPGAs.

3.2.1 Hardware Description Language (HDL)

Um eine einfache Programmierung von FPGAs zu ermöglichen, werden Hardwarebeschreibungssprachen (HDL) als besondere Form von Programmiersprachen verwendet. Hardwarebeschreibungssprache sind Sprachen, die speziell für den Entwurf von Hardware entwickelt worden sind. Sie enthalten neben klassischen Sprachenkonstrukten wie Variablen oder Schleifen auch besondere Sprachkonstrukte für den Hardwareentwurf. Unter Anderem können Hardwarebeschreibungssprachen echte Nebenläufigkeit ausdrücken, sowie Signallaufzeiten exakt modellieren (siehe z.B. [Kae08] Kapitel 4 oder [MVG+12]). Zusätzlich sei an dieser Stelle angemerkt, dass HDLs keinerlei Formen von Speicherverwaltung kennen. Insbesondere sind Konzepte wie ein Variablenstack oder Heap unbekannt.
Damit sind HDLs vergleichsweise rudimentär und der Zugriff auf Speicherstrukturen wie Felder oder die Benutzung von Gleitkommazahlen muss hersteller- und modellspezifisch selbst implementiert werden.
Die beiden bekanntesten Vertreter von Hardwarebeschreibungssprachen sind VHDL [VHD]

und Verilog [Ver]. FPGA Hersteller unterstützen üblicherweise beide Programmiersprachen und bieten entsprechende Übersetzungswerkzeuge für ihre FPGAs an [Alt, Xilk].

3.2.2 High Level Synthesis (HLS)

High Level Synthese bezeichnet die automatische Übersetzung des Programmcodes einer höheren Programmiersprache wie `C,C++` oder `SystemC` in entsprechenden HDL Code.
Durch die höhere Abstraktionsebene einer Hochsprache soll eine schnellere Implementierung des gewünschten Algorithmus bei gleichzeitiger modellspezifischer Übersetzung und Optimierung des Algorithmus für das konkrete FPGA ermöglicht werden.
Aus praktischer Sicht ist jedoch zunächst anzumerken, dass Hochsprachen und Hardwarebeschreibungssprachen unterschiedliche Konzepte unterstützen. So können Hochsprachen wie `C` oder `C++` keine echte Parallelität[1] oder Signallaufzeiten ausdrücken, wohingegen HDLs keinerlei Heapoperationen oder Funktionsaufrufe modellieren können. Aus diesem Grund bilden HLS Werkzeuge nur eine Teilmenge der Hochsprache in HDL Code ab und führen neue Übersetzungsdirektiven[2] in die Hochsprache ein, um z.B. echte Nebenläufigkeit zu modellieren.
Die ersten Generationen von HLS Tools wurde kaum im Produktiveinsatz verwendet, da der erzeugte Code nicht mit einer manuellen Implementierung wettbewerbsfähig war. Dieser Umstand hat sich in den letzten Jahren zunehmend verändert, da eine automatisch generierte Implementierung immer besser spezielle Eigenschaften der FPGAs wie z.B: DSP oder BRAM-Einheiten ausnutzen kann [MVG+12, BRS13]. Des Weiteren bieten moderne HLS Tools durch die Verwendung verschiedener Übersetzungsdirektiven verschiedene Möglichkeiten zur Optimierung an, sodass der Programmierer einfacher mögliche Optimierungen evaluieren kann (siehe z.B: [Xilh]).

3.2.3 Konfiguration

Die Programmierung mit einer Hardwarebeschreibungssprache kann sehr mühselig sein, da sie auf einem sehr niedrigen Abstraktionsniveau geschieht. Darüber hinaus können viele Standardbauteile, wie z.B. eine arithmetisch-logische Einheit für Gleitkommaoperationen in einer Vielzahl von verschiedenen Implementierungen wiederverwendet werden.
Um eine Wiederverwendung und Weitergabe von Hardwarecode zu erlauben, bieten FPGA Hersteller und Drittanbieter daher ihre Intellectual Property (IP) zur Benutzung an. Diese IP ist vergleichbar mit vorkompilierten Bibliotheken aus der Softwareentwicklung, die entsprechend in ein Projekt eingebunden werden können.
Für Standardoperationen und Standardprotokolle existieren bereits fertige HDL Implementierungen, sodass FPGA-Entwickler diese nutzen können. Um die Konfigurierbarkeit

[1] Nebenläufigkeit durch Threads wird hier von der Ausführungsumgebung bereitgestellt.
[2] Diese sind ähnlich zu dem in `C,C++` bekanntem `pragma.` Schlüsselwort

des FPGAs nicht einzuschränken, unterstützen viele IP Entwicklungen zusätzliche verschiedene Konfigurationsmodi. Damit kann der FPGA Entwickler die von ihm verwendete IP für seine Anwendungszwecke konfigurieren und maßschneidern. Als wesentliche Herausforderung stellt sich hier neben der Integration dieser IP Entwicklungen in die übrige Systeminstanz auch die optimale Konfiguration dar.

3.2.4 Synthese

Ziel des Syntheseschrittes ist die Abbildung des HDL Codes in entsprechende logische Formeln, die wiederum mittels der Wahrheitstabellen in den konfigurierbaren Logikblöcken des FPGAs umgesetzt werden können. Hierzu untersucht das Übersetzungswerkzeug den HDL Code zunächst auf verwendete Standardausdrücke wie z.B: eine Addition und generiert die entsprechen Standardbauteile wie z.B: einen Carry-Look-Ahead Addierer.
Anschließend müssen diese Standardbausteine in einer geeigneten Art und Weise miteinander verknüpft werden, sodass sich die gewünschte Funktionalität ergibt. Hier entsteht eine baumartige Struktur, die den gesamten Schaltkreis in seiner sogenannten Netlist wiederspiegelt.

3.2.5 Place & Route

Im Place & Route Schritt wird die zuvor generierte Netlist nun auf die Logikblöcke des FPGAs abgebildet. Dieser trivial wirkende Schritt birgt jedoch einige Besonderheiten. Zunächst muss in diesem Schritt sichergestellt werden, dass die vom Programmierer eingeführten Zeitbedingungen eingehalten werden. Gibt der Programmierer zum Beispiele eine bestimmte Taktrate vor, so müssen Bauteile, die miteinander in Interaktion stehen, entsprechend nahe zueinander liegen, sodass Signallaufzeiten minimiert werden. Hat der Programmierer die Zeitbedingungen zu streng gewählt, also zum Beispiel die Taktrate zu hoch angesetzt und das FPGA kann diese nicht unterstützen, so scheitert diese Phase des Kompilationsprozesses und der Programmierer muss entweder die Zeitbedingungen anpassen oder seine Implementierung ändern. Damit ist die Wahl der passenden Zeitbedingungen als ein iterativer Prozess zu sehen, indem der Programmierer verschiedene Systemkonfigurationen ausprobieren muss. Hierbei werden insbesondere Parameter für die Synthese und das Place % Route variiert, als auch teilweise die Planung von Signalpfaden per Hand getätigt, sodass gerade das Place % Route als eigenständiger Entwicklungsschritt zu sehen ist (vgl. Abschnitt 6.3.3.
Zusätzlich bietet dieser Schritt ein großes Potential zur Optimierung an, da logische Funktionen durch syntaktisch Äquivalenzumformungen so verändert werden können, dass sie weniger Logikblöcke verwenden. Diese Äquivalenzumformungen erfolgen dann auf Basis der logischen Funktion des gesamten Schaltkreises, sodass Optimierungen alle Bauteile berücksichtigen.

3.2.6 Partielle Rekonfiguration

Partielle Rekonfiguration ist ein zusätzlicher Entwurfsschritt, der unabhängig von der eigentlichen Hardwaresynthese geschehen kann. Bei einer partiellen Rekonfiguration wird der FPGA Chip in verschiedene, logisch voneinander getrennte Regionen unterteilt, die unabhängig voneinander konfiguriert werden können. Anschließend kann Hardwarecode für jede Region einzeln synthetisiert, platziert und auf das FPGA geladen werden.
Der große Vorteil der partiellen Rekonfiguration ist die Tatsache, dass eine Rekonfiguration zur Laufzeit möglich ist. Auf diese Weise lassen sich Teile der Hardwareschaltung im laufenden Betrieb ändern und anpassen. Zusätzlich kann die Synthese insgesamt schneller ablaufen, da lediglich Teilbereich des FPGAs und der Schaltung betrachtet werden müssen. Im Gegenzug zu dieser schnellen Synthese können nun Optimierungen weniger stark genutzt werden, da diese sich nicht über die gesamte Schaltung des FPGAs erstrecken, sondern lediglich auf die einzelnen Regionen beschränkt sind.

3.3 Diskussion

Die FPGA Technologie existiert seit Mitte der 1980er Jahre und konnte seit dem eine gewisse Marktreife erreichen. Durch ihre Nähe zu ASICs bieten FPGAs eine sehr schnelle Architektur an, die gleichzeitig energiesparender als herkömmliche CPUs ist. So ist es nicht verwunderlich, dass Veröffentlichungen zum Teile Geschwindigkeitsvorteile um Faktoren 10 und mehr gegenüber klassischer Hardware berichtet haben (vgl. Abschnitt 1.1).
Mit dem heutigen Trend zu GPGPUs und deutlich schnelleren Prozessorgenerationen im Desktop- und Serverbereich konnte dieser Geschwindigkeitsvorteil jedoch durch klassische CPUs und GPGPUs wieder ausgeglichen bzw. teilweise sogar Überschritten werden. Der Grund hierfür liegt in der Tatsache, dass moderne CPUs und GPGPUs um eine Größenordnung schneller getaktet sind als FPGAs, sodass sich hier ein gemischtes Bild in der Performanz ergibt (vgl. [AMY09, KDW10, LRL+12, MWH13, BRS13]).
Vergleicht man jedoch den Energieverbrauch dieser Architekturen mit dem Energieverbrauch von FPGAs, so wird klar, dass der erhöhte Durchsatz von GPGPUs und CPUs teilweise durch einen deutlich höheren Energieverbrauch erreicht wurde (vgl. [KDW10, MWH13]). Damit bilden FPGAs in ressourcenbeschränkten Umgebungen weiterhin eine kostengünstige und flexible Alternative zu ASICs.
Die Programmierung von FPGAs stellt sich als herausfordernd dar, weil die Programmierung mit Hardwarebeschreibungssprachen auf einem sehr niedrigen Abstraktionsniveau geschieht (vgl. [BRS13]).
Betrachtet man die Umsetzung von Algorithmen auf FPGAs (vgl. Abschnitt 1.1), so fällt hier auf, dass diese Implementierungen sich vor allem auf die Umsetzung eines Algorithmus mit einer HDL konzentrieren. Die kritische Evaluierung möglicher Optimierungen und

Änderungen in dieser Implementierung bleiben oft wegen der hohen Komplexität aus. Dieses Vorgehen ist durch immer bessere Werkzeugunterstützung nicht mehr zeitgemäß. Neben dem Einsatz von Hersteller IP ist heute auch die Programmierung auf einem höheren Abstraktionsniveau durch High Level Synthese Werkzeuge möglich [CLN+11]. Die Implementierung stellt dann nicht mehr eine einzige, fixe Größe dar, sondern kann kritisch im Bezug auf verschiedene Optimierungen evaluiert werden.
Damit sind die Herausforderungen bei der Implementierung jedoch nicht verschwunden, sondern haben sich verschoben. Die Programmierung in einer HDL hat sich vereinfacht und kann in vielen Teile durch die Programmierung in einer Hochsprache ersetzt werden. Dennoch muss der generierte HDL Code kritisch evaluiert werden und die Implementierung gegebenenfalls angepasst werden. Zusätzlich bietet sich durch die Fülle an Intellectual Property zwar eine einfache Möglichkeit zum Systementwurf, doch müssen diese ebenfalls kritisch gegeneinander abgewogen werden. Gerade hier zeigt sich, dass die FPGA Programmierung als ein Systementwurf zu sehen ist, indem neben der Implementierung der eigentlich Funktionalität auch die übrigen Systemkomponenten ausgewählt und ihr Einfluss aufeinander abgewägt werden muss.

4 Gauß-Prozesse

Gauß-Prozesse (GP) umfassen eine Menge mathematischer Methoden zur Vorhersage von Datenpunkten basierend auf Trainingsdaten. Sie entstammen dem Teilgebiet der Statistik und des maschinellen Lernens.
Abschnitt 4.1 und 4.2 führen zunächst Grundbegriffe des maschinellen Lernens und der Bayes-Statistik mit Hilfe von [HTF01] ein. Anschließend wird in Abschnitt 4.3 mit Hilfe von [RW06] eine kurze Einführung in vollständige Gauß-Prozesse gegeben.

4.1 Grundbegriffe des maschinellen Lernens

Maschinelles Lernen beschäftigt sich mit Methoden zur automatischen Wissensaufbereitung und Wissensextraktion aus Datenmengen. Hierzu sei im Folgenden angenommen, dass die Daten als N Beobachtungen im d-dimensionalen reellen Raum $\vec{x}_1, \ldots, \vec{x}_N \in \mathbb{R}^d$ vorliegen. Um eine kompakte Schreibweise zu erlauben, seien diese Daten in einer Beobachtungsmatrix gesammelt:

$$X = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1d} \\ x_{21} & x_{22} & \cdots & x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{Nd} \end{pmatrix} = \begin{bmatrix} \vec{x}_1^T \\ \vdots \\ \vec{x}_N^T \end{bmatrix}$$

wobei $\vec{x}_i$ das i-te Beispiel, d.h. die i-te Zeile aus X bezeichne.
In vielen Fällen liegen zusätzlich zu den Beobachtungen auch korrespondierende Ausgaben y vor. In diesem Fall bestehen die Daten aus Paaren $(\vec{x}_1, y_1), \ldots, (\vec{x}_N, y_N)$ welche zu jeder Beobachtung $\vec{x}_i$ das Label $y_i \in \mathcal{Y} \subseteq \mathbb{R}$ assoziieren. Ähnlich zur Beobachtungsmatrix lassen sich die Label in einem Vektor $\vec{y} = (y_1, \ldots, y_N)^T \in \mathcal{Y}^N \subseteq \mathbb{R}^N$ sammeln, sodass sich der Datensatz kompakt als $\mathcal{D} = (X, \vec{y})$ aufschreiben lässt.
Um nun nützliches Wissen aus diesen Daten zu extrahieren, wurde in der Statistik und dem maschinellen Lernen eine Vielzahl verschiedener Verfahren entwickelt. Diese lassen sich im Wesentlichen in zwei Kategorien unterteilen:

- **Unüberwachtes Lernen:** Das Verfahren soll Muster, wie z.B: Gruppen ähnlicher Elemente in den Beobachtungen X finden. Unüberwachtes Lernen benötigt keine Label.

- **Überwachtes Lernen:** Das Verfahren soll einen Zusammenhang zwischen den Beobachtungen $\vec{x}_i$ und den entsprechenden Ausgaben y_i finden, sodass dieser Zusammenhang für Vorhersagen genutzt werden kann. Hierzu wird angenommen, es existiert ein Modell $f_\Theta(\vec{x}) = y$ mit Parametern Θ, welches den Zusammenhang zwischen $\vec{x}$ und y perfekt beschreibt. Das Verfahren versucht nun, anhand der gegebenen Daten $\mathcal{D}$ die Parameter Θ des Modells zu schätzen. Bezeichne $\widehat{\Theta}$ diese Parameterschätzung, so ergibt sich eine Vorhersage $f_{\widehat{\Theta}}(\vec{x}) = \widehat{f}(\vec{x}) = \widehat{y}$, die den Zusammenhang von $\vec{x}_i$ und y_i auf Basis der Daten $\mathcal{D}$ und dem gewählten Verfahren beschreibt.

Ist durch die Anwendung bereits im Vorfeld klar, dass es nur eine endliche Menge möglicher Label gibt, z.B: $y \in \mathcal{Y} = \{-1, +1\}$, so spricht man von einer Klassifikation. Hier ist es das Ziel, jede Beobachtung $\vec{x}$ der entsprechenden Klasse y zuzuweisen.
Ist die Menge möglicher Label reelwertig, d.h. $\mathcal{Y} = \mathbb{R}$, so spricht man von einer Regression. Um die Güte eines Modells $\widehat{f}$ zu bestimmen, ist es entscheidend, wie gut dieses Modell das Label für neue, bisher ungesehenen Beobachtungen $\vec{x}$ vorhersagen kann. Hierzu sei angenommen, dass eine Testmenge $\{(\vec{x}_1, y_1), \ldots, (\vec{x}_M, y_M)\}$ vorliegt, die jeder neuen Beobachtung $\vec{x}_i$ das wahre Label y_i zuweist.
Im Falle der Regression lässt sich nun mit Hilfe des Modells die quadratische Abweichung der Vorhersage $\widehat{f}(\vec{x}_i) = \widehat{y}_i$ vom wahren Label y_i berechnen. Bezeichne $\mathcal{D}_{Test}$ den Testdatensatz mit M Testbeispielen und $var(\mathcal{D}_{Test})$ die Varianz der Label in den Testbeispielen, so ergibt sich der Standardized Mean Squared Error (SMSE):

$$SMSE(\widehat{f}_\Theta) = \frac{1}{M \cdot var(\mathcal{D}_{Test})} \sum_{i=1}^{M} (y_i - \widehat{y}_i)^2 \tag{4.1}$$

Im Falle eines sehr einfachen Lernverfahrens, welches immer den Mittelwert aller Label $\widehat{f}(\vec{x}) = \frac{1}{N} \sum_{i=1}^{N} y_i$ unabhängig von der Beobachtung $\vec{x}$ vorhersagt, ergibt sich als Referenz ein SMSE von ca. 1 (vgl. [RW06] Kapitel 8).
Im Falle der Klassifikation ist der SMSE leider wenig aussagekräftig, da er stark von den gewählten Klassenlabels abhängt. Hier ist es daher üblich, die Anzahl der korrekt klassifizierten Beispiele zu zählen. Für die binäre Klassifikation mit zwei Label ergibt sich damit:

$$Accuracy = \frac{TP + TN}{M} \tag{4.2}$$

wobei TP (True Positive) die Anzahl der korrekt klassifizierten Beispiele mit positivem Label und TN (True Negative) die Anzahl der korrekt klassifizierten Beispiele mit negativem Label bezeichnet. Eine Accuracy von 1 bedeutet also eine perfekte Vorhersage, wohingegen eine Accuracy von 0 bedeutet, dass keinerlei Vorhersage stimmte.

Es stellt sich nun die Frage, inwiefern Test- und Trainingsdaten aufgeteilt werden sollten, um Verfahren untereinander fair zu vergleichen. Hierzu eignen sich die folgenden Ansätze:

- **Train/Test-Split:** Der Datensatz $\mathcal{D}$ wird in einen Trainingsdatensatz $\mathcal{D}_{Train}$ und einen Testdatensatz $\mathcal{D}_{Test}$ unterteilt. Der Trainingsdatensatz dient zur Berechnung des Modells $\widehat{f}$, wohingegen der Testdatensatz die ungesehenen, neuen Beobachtungen zur Verfügung stellt.
  Dieses Vorgehen eignet sich besonders für große Datenmengen, da lediglich ein einziges Modell berechnet werden muss. Die Güte hängt jedoch stark von der Aufteilung der Daten in Training- und Testdaten ab. Eine ungünstige Aufteilung spiegelt sich in einer schlechten Performanz des Verfahrens wieder, wohingegen eine sehr günstige Aufteilung ein vermeintlich schlechteres Verfahren deutlich besser bewertet.

- **Leave-One-Out:** Bei diesem Vorgehen nutzt man jede Beobachtung aus dem Datensatz $\mathcal{D}$ einmalig als Testinstanz und verwendet die übrigen $N-1$ Beobachtungen zum Trainieren des Modells. Somit müssen insgesamt N Modelle berechnet werden, wodurch dieses Vorgehen robuster gegenüber einzelnen, ungünstigen Test- und Trainingsdatensätzen ist, aber auch einen höheren Rechenaufwand erfordert.

- **k-fache Kreuzvalidierung:** Die k-fache Kreuzvalidierung bietet einen Kompromiss zwischen dem einfachen Test/Train Split und dem Leave-One-Out Verfahren. Hier wird der Datensatz in insgesamt k Teilmengen unterteilt und es werden insgesamt k Modelle trainiert. Die i-te Teilmenge dient zum Testen des i-ten Modells, wohingegen die übrigen $k-1$ Teilmengen zum Trainieren verwendet werden. Dann wird pro Modell der Vorhersagefehler berechnet, sodass anschließend der Mittelwert der k Vorhersagefehler eine Bewertung des Verfahrens ermöglicht.

4.2 Grundbegriffe der Bayes-Statistik

Gauß-Prozesse basieren nahezu vollständig auf klassischer Wahrscheinlichkeitsrechnung in welcher Vorhersagefehler explizit modelliert werden, sodass Ergebnisse transparent interpretiert werden können und durch korrekte Fehlerrechnung echte Konfidenzen angegeben werden können. Im Folgenden werden daher die Grundlagen der Bayes-Statistik präsentiert.

In der klassischen Bayes-Statistik nimmt man zunächst an, dass die Funktionswerte y einer bestimmten Verteilung p mit dem Parametervektor Θ entstammen. Mit Hilfe des Satzes von Bayes lässt sich die Verteilung der Parameter Θ gegeben der Trainingsdaten $\mathcal{D}$ berechnen:

$$p(\Theta|\mathcal{D}) = \frac{p(\mathcal{D}|\Theta)p(\Theta)}{p(\mathcal{D})} \tag{4.3}$$

Diese Parameterverteilung kann nun dazu genutzt werden, die Verteilung p zu parametrisieren. Hierzu kann ein Maximum-a-posteriori Schätzer benutzt werden. Dieser wählt denjenigen Parameter $\widehat{\Theta}$ aus, der nach der Verteilung $p(\Theta|\mathcal{D})$ am wahrscheinlichsten ist. Da der Ausdruck $p(\mathcal{D})$ konstant für alle Parameter Θ ist, hat dieser keinen Einfluss auf eine Mximum-a-posteriori-Wahl:

$$\widehat{\Theta} = \arg\max_{\Theta}\{p(\Theta|\mathcal{D})\} = \arg\max_{\Theta}\{p(\mathcal{D}|\Theta)p(\Theta)\} \tag{4.4}$$

Mit Hilfe dieser Schätzung wird der wahrscheinlichste Parameter für die angenommen a priori Verteilung p anhand der Daten bestimmt. Um nun eine konkrete Vorhersage für unbekannte Datenpunkte $\vec{x}$ zu machen, kann der Erwartungswert genutzt werden:

$$\widehat{f}(\vec{x}) = \widehat{y} = \int_y p(y|\widehat{\Theta}, \vec{x}) \cdot y dy \tag{4.5}$$

Die effektive Schätzung des Parameters $\widehat{\Theta}$ und die Berechnung von $\widehat{y}$ beruhen im Wesentlichen auf der konkreten Verteilung p. Hierfür kann eine Normalverteilungsannahme getroffen werden, da sie eine intuitive Interpretation liefert. Bei der Normalverteilung nimmt man an, dass ein Normalfall als Mittelwertvektor m existiert. Zusätzlich modelliert man eine Abweichung vom Normalfall als eine Varianz Σ:

Definition 4.2.1 (Normalverteilung) Eine d-dimensionale Beobachtung $\vec{x} \in \mathbb{R}^d$ heißt normalverteilt mit Mittelwert $\vec{m} \in \mathcal{X} \subseteq \mathbb{R}^d$ und Kovarianz $\Sigma \in \mathbb{R}^{d \times d}$, d.h.

$$\vec{x} \sim \mathcal{N}(\vec{m}, \Sigma) \text{ bzw. } \mathcal{N}(\vec{x}, \vec{m}, \Sigma), \tag{4.6}$$

wenn sie mit Wahrscheinlichkeit

$$p(\vec{x}|\vec{m}, \Sigma) = (2\pi)^{-\frac{N}{2}} |\Sigma|^{-\frac{1}{2}} e^{-\frac{1}{2}(\vec{x}-\vec{m})^T \Sigma^{-1} (\vec{x}-\vec{m})} \tag{4.7}$$

auftreten kann. Die Verteilung $N(\vec{m}, \Sigma)$ wird Normalverteilung oder Gaußverteilung genannt.

4.3 Vollständige Gauß-Prozesse

Gauß-Prozesse erweitern die Normalverteilungsannahme der multivariaten Normalverteilung von einfachen Wertevektoren auf Funktionen. Eine umfassende Übersicht über Gauß-Prozesse bietet [RW06], weshalb sich die folgenden Überlegungen daran anlehnen.
Zunächst seien Gauß-Prozesse intuitiv anhand eines Beispieles eingeführt: Betrachtet man die Kräfte, die auf eine Person in einem Auto während der Beschleunigung wirken, so gibt es sicherlich einen Normalfall - der Fahrer betätigt das Gaspedal, Kraftstoff wird in den Motor gepumpt, das Auto bewegt sich zunächst langsam bis es schließlich immer weiter beschleunigt und die Insassen in die Sitze drückt. Unabhängig von diesem Normalfall fühlt

sich die Beschleunigung in einem Formel-1 Wagen jedoch sicherlich anders an, als die in einem Familienvan. Diese Abweichungen vom Normalfall spiegeln sich als Varianzen in der Beschleunigungskurve wieder und können von vielen physikalischen Faktoren abhängen. Gauß-Prozesse modellieren nun diesen Normalfall und die Varianzen, sodass die Vorhersage des konkreten physikalischen Prozesses - also die Beschleunigung eines Formel-1 Wagens oder die eines Familienvans - anhand der Trainingsdaten $\mathcal{D}$ möglich wird.

Um eine Normalverteilung für Funktionen zu definieren, müssen Funktionen in geeigneter Art und Weise dargestellt werden. Hierzu sei angenommen, dass die Funktion $f : \mathcal{X} \to \mathcal{Y}$ als Wertetabelle vorliegt:

$\vec{x}_1$	$\vec{x}_2$	$\vec{x}_3$	$\vec{x}_4$	$\vec{x}_5$	$\ldots$	$\vec{x}_S$
$f(\vec{x}_1)$	$f(\vec{x}_2)$	$f(\vec{x}_3)$	$f(\vec{x}_4)$	$f(\vec{x}_5)$	$\ldots$	$f(\vec{x}_S)$

Ist bei dieser Darstellung die Indexierung $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_S$ implizit klar - d.h. das Funktionsargument $\vec{x}_i$ ist durch die Position in der Tabelle fest vorgegeben, so reicht es, den Vektor von Funktionswerten $\vec{f} = (f(\vec{x}_1), f(\vec{x}_2), \ldots, f(\vec{x}_S))^T$ zu betrachten. Dieser Funktionswertevektor stellt zunächst eine Stichprobe der eigentlichen Funktion f an den Stellen $\vec{x}_i$ dar. Betrachtet man nun unendliche viele Stellen mit $S \to \infty$ und wählt die Abstände zwischen den Stellen hinreichend klein, so stellt der Vektor $\vec{f}$ die gesamte Funktion f dar[1]. Ausgehend von dem Wertevektor $\vec{f}$ kann man nun eine Gauß-Verteilung für Funktionen definieren. Hierzu sei zunächst die Randverteilungseigenschaft der Normalverteilung eingeführt:

Satz 4.3.1 (Randverteilung der Normalverteilung)
Sei $\vec{x} = \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \end{bmatrix} \sim \mathcal{N}\left(\vec{m} = \begin{bmatrix} \vec{m}_1 \\ \vec{m}_2 \end{bmatrix}, \Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}\right)$ normalverteilt, dann sind die Randverteilungen für $\vec{x}_1$ und $\vec{x}_2$ ebenfalls Normalverteilungen, d.h.

$$\vec{x}_1 \sim \mathcal{N}(\vec{m}_1, \Sigma_{11}) \text{ bzw. } \mathcal{N}(\vec{x}_1, \vec{m}_1, \Sigma_{11})$$

$$\vec{x}_2 \sim \mathcal{N}(\vec{m}_2, \Sigma_{22}) \text{ bzw. } \mathcal{N}(\vec{x}_2, \vec{m}_2, \Sigma_{22})$$

Ein Beweis dieses Satzes ist im Anhang zu finden.

Sei also an dieser Stellen angenommen, dass der konkrete physikalische Prozess durch den die Beobachtungen $\mathcal{D}$ entstanden sind als eine Funktion f mit Wertevektor $\vec{f}$ dargestellt wird. Dann kann man weiterhin annehmen, dass dieser Vektor einer Normalverteilung mit einem Mittelwert und einer Kovarianz folgt. Der Wertevektor der Trainingsdaten $\vec{y}$ bildet

[1] An dieser Stelle sei darauf hingewiesen, dass sich bereits für einfache Funktionen eine Indexierung für den Vektor $\vec{f}$ nicht mehr eindeutig finden lässt, da die Menge der reellen Zahlen $\mathbb{R}$ überabzählbar unendlich ist [Can92]. Im Sinne einer anschaulichen Herleitung sei jedoch angenommen, dass ein Vektor $\vec{f}$ dennoch existiert.

damit eine Stichprobe der Funktion f an N Stellen und stellt daher eine Randverteilung dieser Verteilung dar, die ebenfalls normalverteilt ist. Damit motiviert sich nun die folgende Definition von Gauß-Prozessen mit Hilfe der multivariaten Gaußverteilung:

Definition 4.3.1 (Gauß-Prozess) Ein Gauß-Prozess (GP) ist eine Kollektion von Zufallsvariablen $\{f(\vec{x})|\vec{x} \in \mathbb{R}^d\}$, sodass jede endliche Teilkollektion von Zufallsvariablen (multivariat) gaußverteilt ist:

$$f(\cdot) \sim \mathcal{GP}(m(\cdot), k(\cdot, \cdot)) \tag{4.8}$$

mit Mittelwertfunktion $m(\vec{x})$ und Kovarianzfunktion $k(\vec{x}, \vec{x}')$:

$$\begin{aligned} m(\vec{x}) &= \mathbb{E}[f(\vec{x})] & (4.9) \\ k(\vec{x}, \vec{x}') &= \mathbb{E}[(f(\vec{x}) - m(\vec{x}))(f(\vec{x}') - m(\vec{x}'))] & (4.10) \end{aligned}$$

Die Mittelwertfunktion $m(\vec{x})$ modelliert wie der Mittelwertvektor $\vec{m}$ den Normalfall, wohingegen die Kovarianzfunktion bzw. Kernfunktion $k(\vec{x}, \vec{x}')$ die Ähnlichkeit zwischen zwei Datenpunkten $\vec{x}$ und $\vec{x}'$ beschreibt. Ein großer Kernfunktionswert modelliert eine hohe Ähnlichkeit, wohingegen ein niedriger Wert eine geringe Ähnlichkeit zwischen $\vec{x}$ und $\vec{x}'$ angibt.

4.3.1 Gauß-Prozesse zur Regression

Nimmt man nun an, die Trainingsdaten $\mathcal{D} = (X, \vec{y}_\mathcal{D})$ sind nach einem GP entstanden, so folgt der Wertevektor $\vec{y}_\mathcal{D} = (y_1, y_2, \dots, y_N)^T$ einer multivarianten Gaußverteilung mit Mittelwertvektor $\vec{m}_\mathcal{D} = (m(\vec{x}_1), m(\vec{x}_2), \dots, m(\vec{x}_N))^T$ und Kovarianzmatrix

$$K(X, X) = \begin{pmatrix} k(\vec{x}_1, \vec{x}_1) & k(\vec{x}_1, \vec{x}_2) & \dots & k(\vec{x}_1, \vec{x}_N) \\ k(\vec{x}_2, \vec{x}_1) & k(\vec{x}_2, \vec{x}_2) & \dots & k(\vec{x}_2, \vec{x}_N) \\ \vdots & \vdots & \ddots & \vdots \\ k(\vec{x}_N, \vec{x}_1) & k(\vec{x}_N, \vec{x}_2) & \dots & k(\vec{x}_N, \vec{x}_N) \end{pmatrix}$$

Bezeichne $\vec{x}$ eine neue Beobachtung deren passender Funktionswert y vorherzusagen ist. Dann ergibt sich nach GP Definition, dass der Vektor $\vec{y} = (y_1, y_2, \dots, y_N, y)^T$ ebenfalls normalverteilt mit $\vec{m} = (m(\vec{x}_1), m(\vec{x}_2), \dots, m(\vec{x}_N), m(\vec{x}))^T$ und Kernmatrix $K = \begin{bmatrix} K(X, X) & K(X, \vec{x}) \\ K(\vec{x}, X) & k(\vec{x}, \vec{x}) \end{bmatrix}$ ist, wobei $K(X, \vec{x})$ und $K(\vec{x}, X)$ den Vektor der Kernfunktionen zwischen den Trainingsdaten X und der neuen Beobachtung $\vec{x}$ bezeichnet. An dieser Stelle sei explizit darauf hingewiesen, dass die Verteilung des Vektors $\vec{y}$ vollständig bekannt ist, d.h. der Mittelwertvektor $\vec{m}$ und die Kovarianzmatrix K können vollständig berechnet werden.

Des Weiteren sind die Einträge $y_1, \dots, y_N$ ebenfalls bekannt, womit der Wert y durch die Normalverteilungsannahme von $\vec{y}$ nicht mehr frei wählbar ist: Durch die Randverteilungseigenschaft der Normalverteilung ergibt sich, dass y ebenfalls eindimensional normalverteilt

ist. Darüber hinaus muss diese Verteilung zu den bekannten Werten $y_1, \ldots, y_N$ passen, d.h. die Verbundverteilung von $y_1, \ldots, y_N$ und y muss zu der Normalverteilung mit Mittelwertvektor $\vec{m}$ und Kernmatrix K gehören.
Dieser Umstand wird in der Konditionalisierungseigenschaft der Normalverteilung festgehalten:

Satz 4.3.2 (Konditionalisierung der Normalverteilung)
Sei $\vec{x} = \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \end{bmatrix} \sim \mathcal{N}\left(\vec{m} = \begin{bmatrix} \vec{m}_1 \\ \vec{m}_2 \end{bmatrix}, \Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}\right)$ normalverteilt, dann ist die bzgl. $\vec{x}_1$ konditionalisierte Verteilung $\vec{x}_2|\vec{x}_1$ von $\vec{x}_2$ eine Normalverteilung mit:

$$\vec{x}_2|\vec{x}_1 \sim \mathcal{N}(\vec{m}_2 + \Sigma_{21}^T \Sigma_{11}^{-1}(\vec{x}_1 - \vec{m}_1), \Sigma_{22} - \Sigma_{12}^T \Sigma_{11}^{-1} \Sigma_{12})$$

Für einen Beweis sei auch an dieser Stelle auf den Anhang verwiesen. Unter Verwendung von Gleichung 4.5 mit den Parametern $\widehat{\Theta} = (m(\cdot), k(\cdot, \cdot))$ kann nun der Erwartungswert als Schätzung für den Funktionswert $\widehat{f}(\vec{x})$ benutzt werden. Dieser entspricht bei einer Normalverteilung gerade dem Mittelwert (vgl. Satz A.0.3 im Anhang), wobei im Folgenden auf die explizite Angabe von $\widehat{\Theta}$ verzichtet wird:

$$\begin{aligned} \widehat{f}(\vec{x}) &= \int_y p(y|\widehat{\Theta}) \cdot y dy \\ &= \int_y \mathcal{N}(m(\vec{x}) + K(\vec{x}, X)K(X, X)^{-1}(\vec{y} - \vec{m}), \\ &\qquad k(\vec{x}, \vec{x}) - K(\vec{x}, X)K(X, X)^{-1}K(X, \vec{x})) \cdot y dy \\ &= m(\vec{x}) + K(\vec{x}, X)K(X, X)^{-1}(\vec{y} - \vec{m}) \end{aligned} \tag{4.11}$$

Die Beobachtung von echten physikalischen Prozessen unterliegt häufig kleineren Messfehlern, sodass die gemessenen Funktionswerte y_i in den Trainingsdaten nicht notwendigerweise den wahren Funktionswerten $f(\vec{x}_i)$ des physikalischen Prozesses entsprechen müssen. Als einfache Näherung kann diese Abweichung als unabhängig und identischverteilter Fehler $\varepsilon \sim \mathcal{N}(0, \sigma_n^2)$ angenommen werden, sodass für die Messungen folgt:

$$y_i = f(\vec{x}_i) + \varepsilon$$

Bezeichne I_N die $N \times N$ Einheitsmatrix, dann kann dieser Messfehler in Gleichung 4.11 aufgenommen werden:

$$\widehat{f}(\vec{x}) = m(\vec{x}) + K(\vec{x}, X)[K(X, X) + I_N \sigma_n^2]^{-1}(\vec{y} - \vec{m}) \tag{4.12}$$

4.3.2 Gauß-Prozesse zur Klassifikation

Gleichung 4.12 gibt eine Möglichkeit zur Regression von beliebigen Datenpunkten $\vec{x}$ basierend auf den Trainingsdaten $\mathcal{D}$. Im Folgenden sei die Klassifikation für den Zweiklassenfall $\mathcal{Y} = \{-1, +1\}$ kurz angerissen, wobei sich die Klassifikation von mehreren Klassen auf

diesen Fall zurückführen lässt.
Nimmt man also den Fall der binäre Klassifikation mit $\mathcal{Y} = \{-1, +1\}$ an, so wird die Vorhersage mit Gleichung 4.12 vermutlich nie exakt -1 oder $+1$ ergeben. Um dennoch zu einer sinnvollen Klassifikation zu gelangen, kann man die Vorhersagen zunächst als Wahrscheinlichkeiten interpretieren: Ein Wert nahe $+1$ entspricht einer hohen Wahrscheinlichkeit, dass Klasse $+1$ vorhergesagt werden soll und ein Werte nahe -1 entspricht einer hohen Wahrscheinlichkeit, dass die Klasse -1 gesucht wird. Wahrscheinlichkeiten sind auf das Intervall $[0, 1]$ beschränkt und müssen in Summe insbesondere Eins ergeben. Daher wird die Vorhersage $\widehat{f}(\vec{x})$ mit Hilfe einer Sigmoid-Funktion $\lambda(\cdot)$ auf das Intervall $[0, 1]$ beschränkt:

$$p(y|\vec{x}, \mathcal{D}) = \lambda(y \cdot \widehat{f}(\vec{x})))$$

Eine einfache Wahl von λ ist die logistische Funktion

$$\lambda(a) = \frac{1}{1 + e^{-a}}$$

Aus praktischer Sicht ist also vor Allem das Vorzeichen der Vorhersage $\widehat{f}(\vec{x})$ entscheidend: Sind die Vorzeichen gleich, so ergibt sich ein positives Vorzeichen für a in $\frac{1}{1+e^{-a}}$, wodurch dieser Term gegen 1 geht. Sind die Vorzeichen unterschiedlich, so ergibt sich ein negatives Vorzeichen für a in $\frac{1}{1+e^{-a}}$ womit der Gesamtterm gegen 0 geht.
Mit Gleichung 4.4 ergibt sich schließlich die Vorhersage der wahrscheinlichsten Klasse:

$$\widehat{y} = \arg\max_{y}\{\lambda(y \cdot \widehat{f}(\vec{x}))\} = \arg\max\{\lambda(+1 \cdot \widehat{f}(\vec{x})), \lambda(-1 \cdot \widehat{f}(\vec{x}))\} \tag{4.13}$$

4.4 Anwendung von Gauß-Prozessen

Bisher wurde angenommen, dass die Mittelwertfunktion $m(\cdot)$, sowie die Kernfunktion $k(\cdot, \cdot)$ bekannt und durch den Benutzer vorgegeben sind. In vielen praktischen Anwendungen ist die Mittelwertfunktion $m(\cdot)$ nicht bekannt, sodass eine passende Annahme getroffen werden muss. Hier eignen sich prinzipiell zwei Vorgehensweisen: Entweder man schätzt diese Funktion aus den Daten $\mathcal{D}$ oder man nimmt einen konstanten Mittelwert $m(\cdot) = m$ an. Letzteres Vorgehen stellt keine wirkliche Einschränkung für Gauß-Prozesse dar, da ein GP keinerlei Annahmen über die Funktionen selber trifft, sondern lediglich eine Annahme trifft, wie diese Funktion verteilt ist. Durch die Konditionalisierung bzgl. der Trainingsdaten $\mathcal{D}$ betrachtet der GP nur Funktionen, die zu den Trainingsdaten passen. Damit spielt der Mittelwertvektor nur eine untergeordnete Rolle in der Vorhersage, sodass dieser der Einfachheit halber häufig als $m(\cdot) = 0$ gewählt wird (vgl. [RW06], Kapitel 2 und 5).
Im Falle der Kernfunktion greift diese Argumentation jedoch nicht, da die Kernfunktion den Zusammenhang zwischen einzelnen Datenpunkten modelliert. Die Wahl einer passenden Kernfunktion spielt eine zentrale Rolle für eine hohe Vorhersagegüte. Eine radiale

Basisfunktion (RBF-Kernel) folgt auf natürliche Weise der Ähnlichkeitsinterpretation einer Kernmatrix, sodass diese eine sinnvolle Kernfunktion darstellt:

$$k(\vec{x}, \vec{x}\,') = l \cdot exp\left(-\frac{||\vec{x} - \vec{x}\,'||^2}{2 \cdot \sigma^2}\right) = l \cdot exp\left(-\gamma||\vec{x} - \vec{x}\,'||^2\right) \tag{4.14}$$

Hier beschreibt $\gamma = \frac{1}{2 \cdot \sigma^2} \in \mathbb{R}$ und $l \in \mathbb{R}$ freie Parameter zur Skalierung, die passend zu den Trainingsdaten gewählt werden müssen.
Bei der Anwendung eines Gauß-Prozesses zur Regression oder Klassifikation kann zunächst $\alpha = [K(X, X) + I_N \sigma_n^2]^{-1}(\vec{y} - \vec{m})$ in einer Art Trainingsphase vorberechnet werden, sodass bei der Anwendung lediglich $m(\vec{x}) + K(\vec{x}, X) \cdot \alpha$ berechnet werden muss. Zentrale Operation in der Trainingsphase ist die Invertierung von $[K(X, X) + I_N \sigma_n^2]$.
Rasmussen et al. schlagen daher einen Algorithmus zur Berechnung von α vor, der ausnutzt, dass die Kernmatrix $K(X, X)$ symmetrisch und positiv definit ist (vgl. [RW06] Kapitel 2). Dieser Algorithmus verwendet eine Cholesky-Zerlegung, um eine untere Dreiecksmatrix zu berechnen, sodass dann mittels Vorwärts-Rückwärts-Einsetzen die gesuchte Inverse berechnet werden kann. Bei geeigneter Implementierung hat die Cholesky-Zerlegung eine Laufzeit von $\mathcal{O}(\frac{1}{2}N^3)$ [KM11], wohingegen die Laufzeit von Vorwärts-Rückwärts-Einsetzen mit $\mathcal{O}(N^2)$ zu bewerten ist [CLR10], was zu einer Gesamtlaufzeit von $\mathcal{O}(N^3)$ führt.
Des Weiteren ist bekannt, dass sich die Matrixinvertierung mittels Divide and Conquer auf Matrixmultiplikationen zurückführen lässt, sodass die Laufzeit der Invertierung von der Laufzeit des verwendeten Multiplikationsalgorithmus abhängt [CLR10]. Für die Matrixmultiplikation ist eine untere Schranke für die Laufzeit mit $\Omega(N^2 logN)$ bekannt [Raz02] . Dennoch gibt es bisher[2] keinen Algorithmus, der diese Schranke ausnutzen kann. Aktuell bietet der schnellste bekannte Multiplikationsalgorithmus für quadratische Matrizen eine Lauftzeit von $\mathcal{O}(N^{2,373})$, sodass eine Matrixinvertierung prinzipiell auch mit dieser Laufzeit möglich wäre [Wil14]. Dieser Algorithmus kann seinen Geschwindigkeitsvorteil erst bei extrem großen Matrizen ausspielen, die die Grenzen heutiger Hardware sprengen (vgl. [Rob05]), sodass in der Praxis häufig der Strassen-Algorithmus mit $\mathcal{O}(N^{2,807})$ verwendet wird [CLR10]. Dieser hat für kleinere Matrizen ($N \approx 1000$) ebenfalls fast kubische Laufzeit und ist numerisch instabiler als klassische Verfahren wie der Gauß-Jordan Algorithmus (vgl. [Mil74]), sodass sein Nutzen an dieser Stelle hinterfragbar ist.
Des Weiteren ist für die Matrixinvertierung bekannt, dass diese gerade bei größerer Dimension durch Rundungsfehler numerisch instabil werden kann. Um diesen Effekt zu kompensieren, ist es daher üblich σ_n^2 entsprechend groß zu wählen, sodass die Hauptdiagonale von $[K(X, X) + I_N \sigma_n^2]$ entsprechend groß ist, was wiederum zu größeren Eigenwerten und damit zu einer numerischen Stabilität bei der Invertierung führt (vgl. [FO08]).

[2] Stand 2015.

4.5 Diskussion

Gauß-Prozesse bieten eine Methode zur Regression, die vollständig auf Annahmen der Normalverteilung beruht. Damit lassen sich neben dem Mittelwert auch die Varianz, sowie die log-likelihood der Daten angeben (siehe hierzu [RW06] Kapitel 2). Durch Nutzung einer Sigmoidfunktion kann diese Regression in eine Klassifikation umgewandelt werden, wobei hier die resultierende Verteilung für die Vorhersage keine Normalverteilung ist.
Durch die Anwendung von Kernfunktionen lassen sich verschiedene Ähnlichkeitsmaße für Gauß-Prozesse definieren (siehe hierzu [RW06] Kapitel 4), sodass ein GP für verschiedene Anwendungsgebiete einsetzbar ist. Bei Verwendung eines RBF-Kernels mit geeigneten Skalierungsparametern sind Gauß-Prozesse äquivalent zum k-nächste Nachbarn (k-NN) Verfahren (vgl. [HTF01], Kapitel 13 und [RW06], Kapitel 7).
Zusätzlich lässt sich eine gewisse Ähnlichkeit von GP zu Spline Methoden herleiten, die im Wesentlichen auf der Approximation einer Funktion durch Funktionsteile basiert. Auch hier lässt sich eine äquivalente Formulierung bei der entsprechenden Wahl der Kernfunktion für einige Spline-Methoden wählen (vgl. [RW06] Kapitel 6).
Betrachtet man die Regression mit Gauß-Prozessen ohne Messfehlern in den Beobachtungen (d.h. $\varepsilon = 0$), so zeigt sich zunächst, dass durch die Randverteilungseigenschaft jede mögliche, zu den Daten passende Funktion vorhergesagt werden kann. Durch die Anwendung der Kernfunktion wird hier nun eine gewisse Form der Stetigkeit vorausgesetzt, sodass die vorhergesagte Funktion sich nicht beliebig sprunghaft verhält. Ein ähnliches Vorgehen wurde bereits in sogenannten regularization networks benutzt, die diese Stetigkeitsbedingung explizit in die Modellberechnung aufnehmen. Damit lässt sich auch hier eine Äquivalenz von Gauß-Prozessen zu regularization networks zeigen, wobei diese sogar unabhängig von der konkreten Kernfunktion ist (vgl. [RW06] Kapitel 6).
Trotz der Namesähnlichkeit führen regularization networks und Gauß-Prozesse keine Regularisierung auf Modellebene durch, sondern nehmen lediglich die passenste Funktion im Sinne der Daten und der Kernfunktion an. Das durch GP berechnete Modell α enthält immer genau N Einträge, d.h. jede Trainingsbeobachtung trägt zur Vorhersage bei. Andere Methoden wie z.B. die SVM nehmen neben der Stetigskeitsbedingung der vorhergesagten Funktion eine Regularisierung explizit in die Berechnung des Modells mit auf. Damit kann das berechnete Modell weniger als N Einträge enthalten, wodurch es weniger stark die Trainingsdaten $\mathcal{D}$ auswendig lernt.
Ein GP Modell modelliert diese Abstraktion nicht explizit, sodass diese sich nur aus einer geschickte Wahl der Kernfunktion ergibt. Bei einer ungeschickte Wahl der Kernfunktion lernt ein Gauß-Prozess die Daten lediglich auswendig, womit sich auch hier wieder die Ähnlichkeit zu k-NN mit $k = 1$ zeigt.

5 Online Gauß-Prozesse

Im ersten Kapitel der vorliegenden Arbeit wurden die Möglichkeiten von Gauß-Prozessen zur Lösung von Problemen im Bereich der eingebetteten Systeme vorgestellt. Daraufhin wurden FPGAs als mögliche Systemarchitektur für eingebettete Systeme diskutiert. Im vorangeganenen Kapitel wurden dann Gauß-Prozesse als Methode eingeführt.
Hier zeigte sich, dass die Trainingsphase von Gauß-Prozessen eine kubische Laufzeit in der Anzahl der Trainingsbeispiele hat. Diese Laufzeit ist bereits für Server- und Desktopsysteme herausfordernd und mit eingebetteten Systemen kaum zu bewältigen. Nimmt man in einem idealisierten Beispiel an, dass ein Prozessor mit 1 GHz Taktfrequenz in jedem Takt eine Operation durchführt und keinerlei anderen Prozesse auf der CPU tätig sind, so lässt sich die Dauer für eine Matrixinvertierung für $N = 50000$ abschätzen:

$$\frac{50000^3}{1 \text{ GHz}} = \frac{50000^3}{\frac{10^9}{1s}} = 125000s \approx 35 \text{ Tage}$$

Um die kubische Laufzeit der Matrixinvertierung zu umgehen, wurden in der Literatur einige approximative Lösungen für Gauß-Prozesse vorgestellt. Abschnitt 5.1 geht auf Approximationen der Kernmatrix $K(X, X)$ ein, wohingegen Abschnitt 5.2 Online GP Varianten vorstellt, die jeweils ein Trainingsbeispiel nach dem Anderen betrachten.

5.1 Approximative Gauß-Prozesse

Wegen der Allgemein hohen Komplexität der Matrixinvertierung sind eine Mengen an approximativen Methoden aufgekommen, die bereits in [QCR05, RW06, CCL$^+$13, LCH$^+$14, HHL15] sehr ausführlich untersucht und verglichen wurden. Aus diesem Grund sei an dieser Stelle nur die gemeinsame Grundidee dieser Verfahren angerissen und für weitere Details an die entsprecheden Arbeiten verwiesen.
Zunächst lassen sich zwei Beobachtungen festhalten. Zum einen ist die Berechnung der Kernmatrix $K(X, X)$ unter Ausnutzung der Symmetrie in $\mathcal{O}(\frac{1}{2}N^2)$ möglich. Zum Anderen tragen Nulleinträge in der Kernmatrix nicht zur Invertierung bei, sodass diese übersprungen werden können. Nimmt man also zunächst an, dass die Matrix $K(X, X)$ vorberechnet

ist, so kann eine andere Matrix $\widetilde{K} \in \mathbb{R}^{N \times N}$ berechnet werden, sodass K und $\widetilde{K}$ sich möglichst ähnlich sind:

$$||K(X,X) - \widetilde{K}||_F^2 \rightarrow min$$

Um nun eine echte Beschleunigung der Invertierung zu erhalten, sollte die Matrix $\widetilde{K}$ möglichst viele Nulleinträge beinhalten, die dann während der Invertierung übersprungen werden können. Hierzu gibt es eine Reihe von verschiedenen Ansätzen:
Smola und Schölkopf fügen in [SS00] dem Minimierungsproblem einen Regularisierungsanteil hinzu und zeigen dann, wie das von ihnen formulierte Problem durch verschiedene Heuristiken effizient gelöst werden kann.
Die Subset-of-Regressor (SOR) Methode von Silverman [Sil85] und Wahba et al. [WLG+99] hingegen betrachtet nur eine Teilmengen von m induzierenden Variablen, die zur Vorhersage genutzt werden. Damit lässt sich die Kernmatrix als eine 2×2 Blockmatrix $K(X,X) = \begin{bmatrix} K_1 & K_2 \\ K_2^T & K_3 \end{bmatrix}$ mit $K_1 \in \mathbb{R}^{m \times m}$ aufschreiben. Aus algorithmischer Sicht ist es dann ausreichend, lediglich den $m \times m$ Teilblock K_1 zu invertieren. Die geschickte Wahl der induzierenden Variablen bestimmt hier die Güte der Approximation und die gewonnene Beschleunigung.
Ausgehend von der SOR-Methde haben Snelson und Ghahramani in [SG05] und Quinonero und Rasmussen in [QCR05] weitere Approximationsmethoden hergeleitet, die jeweils eine andere Blockstruktur für $K(X,X)$ annehmen und so zu leicht unterschiedlichen Ausdrücken gelangen.

5.2 Online Gauß-Prozesse

Gemeinsame Basis der im vorherigen Abschnitt vorgestellten Approximationsmethoden ist die Annahme, dass die Kernmatrix $K(X,X)$ bereits vollständig vorliegt. Für eingebettete Systeme ist diese Annahme bereits herausfordernd. Betrachtet man in Anlehung an das vorherige Beispiel $N = 50000$ Trainingsbeispiele und benutzt für die Einträge der Kernmatrix einfache Gleitkommazahlen zu je 4 Byte, so ergibt sich für die quadratische Größe von $K(X,X)$ folgender Platzverbrauch:

$$4 \cdot N \cdot N \text{ b } = 4 \cdot 50000 \cdot 50000 \text{ b } = 10 \text{ Gb}$$

Um die vollständige Berechnung und Speicherung der Kernmatrix zu umgehen, bieten sich daher Online Algorithmen an, die jedes Trainingsbeispiel in konstanter Zeit und mit konstantem Speicherbedarf abarbeiten.
Hoang et al. leiten in [HHL15] für verschiedene GP Approximationen Anytime-Algorithmen (Anytime Sparse GP Regression) her, die prinzipiell auch als Online Algorithmen verwendet werden könne. Die Grundidee der Autoren basiert auf der Herleitung einer konkaven, ableitbaren Funktion die in ihrem Maximum der gesuchten Vorhersageverteilung

entspricht. Dann lässt sich diese Funktion mittels Stochastic Gradient Ascent in einem Anytime-Algorithmus maximieren.
Für die Anwendung von Stochastic Gradient Ascent müssen die Daten zuvor in einem Vorverarbeitungsschritt in gemeinsame Blöcke unterteilt werden. Zusätzlich zeigen diese Algorithmen erst bei mehreren Läufen über die Daten eine ausreichende Konvergenz, weshalb sie sich nicht für ein echtes Online Szenario für Vorhersagen während der Systemnutzung eignen.
Xu et al. stellen in [XLC+14] einen Online GP Algorithmus (GP-Localize) zur Lokalisierung von mobilen Robotern mittels Wifi Signale vor. Dieser Algorithmus nutzt insbesondere lokale Abhängigkeiten zwischen der aktuellen und der nächsten Position eines Roboters aus. Hierzu werden zunächst τ Beobachtungen gesammelt und anschließend zusammengefasst. Diese Zusammenfassung wird dann in die eigentlichen Vorhersage eingearbeitet, um so eine gewisse Lokalität zu den nächsten τ Beobachtungen zu schaffen.
Csató et al. haben in [CO02] einen Online-Algorithmus für Gauß-Prozesse (Projection GP) auf Basis der KL-Divergenz abgeleitet, der im Wesentlichen iterativ die konditionalisierte Gauß-Verteilung approximiert (Satz 4.3.2) und mit Hilfe des Matrix-Inversions-Lemmas (siehe Anhang A.0.1) effizient implementiert werden kann. Hierbei wird eine Menge von Trainingsbeispielen als Basisvektoren zur Laufzeit ausgewählt, sodass für einen konstanten Speicherbedarf und eine konstante Rechenzeit pro Beispiel gesorgt ist.
GP-Localize wurde speziell für den Einsatz bei der Lokalisierung für mobile Roboter mit Wifi Signalen konzipiert, wohingegen Projection GP mit Verwendung der KL-Divergenz einen allgemeineren Ansatz verfolgt, der besser zu der statistischen Interpretation von Gauß-Prozessen passt. In [XLC+14] konnten die Autoren zeigen, dass GP-Localize eine Online Variante der partially independent training conditional (PITC) Methode für Gauß-Prozesse ist. Quiñonero-Candela et al. konnten durch geschickte Notation bereits in [QCR05] zeigen, dass die PITC Methode und die bayesian committee machine (BCM) die gleiche mathematische Formulierung zugrunde legen. Schließlich wurde in [RW06], Kapitel 8 gezeigt, dass Projection GP eine besserer Performanz bietet als die BCM. Damit scheint Projection GP durch seine allgemeinere Verwendbarkeit, sowie durch den konstanten Speicher- und Rechenbedarf ein guter Kandidat für die Umsetzung auf einem FPGA zu sein.

5.2.1 Projection GP

Projection GP ist ein iteratives Verfahren, welches eine feste Anzahl von Beobachtungen in einer Menge von Basisvektoren speichert und neue Beobachtungen auf Grundlage dieser Basisvektoren bewertet, wodurch schließlich ein vollständiger Gauß-Prozess approximiert wird. Durch den konstanten Speicherbedarf lässt sich das Verfahren effizient implementieren, sodass es gut für den Einsatz in eingebetteten Systemen geeignet ist. Des Weiteren

basiert die Herleitung des Verfahrens im Wesentlichen auf Überlegungen der klassischen Wahrscheinlichkeitsrechnung, weshalb auch hier Ergebnisse transparent interpretiert werden können.
Bei vollständigen Gauß-Prozessen wird zur Vorhersage die bedingte Randverteilung für $\hat{y}|\vec{y}$ an der Stelle $\vec{x}$, also

$$\begin{aligned} & \mathcal{N}(m(\vec{x}) + K(\vec{x}, X)[K(X,X) + I_N\sigma_n^2]^{-1}(\vec{y} - \vec{m}), \\ & \quad K(\vec{x},\vec{x}) - K(\vec{x}, X)[K(X,X) + I_N\sigma_n^2]^{-1}K(X,\vec{x})) \\ = \; & \mathcal{N}(\widehat{f}(\vec{x}), var(\vec{x},\vec{x})) \end{aligned}$$

benutzt. Bezeichne $var(\vec{x}_i, \vec{x}_j)$ die Varianz zwischen den Beobachtungen $\vec{x}_i$ und $\vec{x}_j$, dann lassen sich die Parameter $\widehat{f}(\vec{x})$ und $var(\vec{x},\vec{x})$ dieser Verteilung mit Hilfe des Representationslemmas [CO02] durch eine Summe darstellen, welche sich durch geeignete Umformungen in einem iterativen Algorithmus ausdrücken lassen.

Lemma 1 (Representationslemma) Gegeben sei ein $\mathcal{GP}(m(\cdot), k(\cdot,\cdot))$ mit Mittelwertfunktion m und Kernel k. Seien ferner N Trainingsdaten $\mathcal{D}$ gegeben und bezeichne p die Dichtefunktion der Normalverteilung, dann lässt sich die marginalisierte konditionalisierte Normalverteilung $\mathcal{N}(\widehat{f}(\vec{x}), var(\vec{x}))$ darstellen als:

$$\begin{aligned} \widehat{f}(\vec{x}) &= m(\vec{x}) + \sum_{i=1}^{N} k(\vec{x}, x_i) q_i \\ var(\vec{x},\vec{x}) &= k(\vec{x},\vec{x}) + \sum_{i,j=1}^{N} k(\vec{x},\vec{x}_i) R_{i,j} k(\vec{x}_j, \vec{x}) \\ q_i &= \frac{1}{Z} \int \frac{\partial p(D|\vec{y})}{\partial f(x_i)} p(f) df \\ R_{i,j} &= \frac{1}{Z} \int p(f) \frac{\partial^2 p(\mathcal{D}|\vec{y})}{\partial \vec{y}_i \partial \vec{y}_j} - q_i q_j df \\ Z &= \int p(f) p(D|\vec{y}) df \end{aligned}$$

Ein detaillierter Beweis dieses Lemmas findet sich in [Csa02], weshalb an dieser Stelle lediglich die Beweisidee vorgestellt wird. Ausgehend von den Überlegungen aus Abschnitt 4.3 sei angenommen, dass eine Funktion f durch den Funktionswertevektor $\vec{f}$ dargestellt wird. Dann bilden die Trainingsdaten $\vec{y}$ eine Stichprobe dieser Funktion, auf dessen Basis man die Wahrscheinlichkeit für einen Funktionswert y für ein gegebenes $\vec{x}$ abschätzen kann. Sei im Folgenden zur Vereinfachung der Notation das Beispiel $\vec{x}$ in die Trainingsdaten $\mathcal{D}$ aufgenommen, so folgt mit Hilfe des Satzes von Bayes und der Definition der totalen Wahrscheinlichkeit:

$$p(y|\mathcal{D}) = \frac{p(\mathcal{D}|y)p(y)}{p(\mathcal{D})} = \frac{p(\mathcal{D})p(y)}{\int p(\mathcal{D}|\vec{y})p(\vec{y})d\vec{y}}$$

Zur Vorhersage wird der Erwartungswert der Verteilung $\mathcal{N}(\widehat{f}(\vec{x}), var(\vec{x}))$ (siehe Abschnitt 4.3) verwendet:

$$\begin{aligned}\widehat{f}(\vec{x}) &= \int p(y|\mathcal{D})ydy = \int p(y)y\frac{p(\mathcal{D})p(y)}{\int p(\mathcal{D}|\vec{y})p(\vec{y})d\vec{y}} \\ &= m(\vec{x}) + K(\vec{x}, X)[K(X, X) + I_N\sigma_n^2]^{-1}(\vec{y} - \vec{m})\end{aligned}$$

Ausgehend von dieser Formulierung führen die Autoren dann an, dass für eine Normalverteilung p mit Mittelwert $\vec{m}$ und Kovarianz Σ und einer Funktion $g : \mathbb{R}^d \to \mathbb{R}$, die selber und alle ihre Ableitungen langsamer als ein Polynom wächst, gilt:

$$\int p(x)xg(x)dx = \vec{m}\int p(x)g(x)dx + \Sigma\int p(x)\nabla g(x)dx$$

Diesen Satz nutzen die Autoren schließlich, um in einer Reihe von algebraischen Umformungen das gesuchte Lemma zu beweisen. Zu bemerken ist an dieser Stelle noch der Umstand, dass der Ausdruck $p(\vec{y}|\mathcal{D})$ lediglich alle Funktionswerte aus den Trainingsdaten $\mathcal{D}$ enthält. Das bedeutet, dass die Wahrscheinlichkeit für Funktionswerte, die nicht innerhalb der Trainingsdaten liegen gleich Null ist. Aus diesem Grund ist es ausreichend, das entsprechende Integral lediglich für die N Trainingspunkte auszuwerten, womit es in einer Summe zerfällt.
Problematisch erweist sich bei dieser Herleitung die Berechnung von $q_i, R_{i,j}$ und Z, da hier rechenintensive Integrale gelöst werden müssen. Als weitere Vereinfachung nehmen die Autoren daher an, dass die Daten bedingt unabhängig seien, d.h. es gilt $p(\mathcal{D}) = \prod_{i=1}^{N} p(y_i|\vec{x}_i)$, sodass folgt:

$$\begin{aligned}p(\mathcal{D}|\vec{y}) &= \frac{p(\vec{y}|\mathcal{D})p(\vec{y})}{p(\mathcal{D})} = \frac{\prod_{i=1}^{N} p(y_i|x_i)\prod_{i=1}^{N} p(y_i)}{p(\mathcal{D})} \\ &= \frac{p(y_N|x_N)p(y_N)\left[\prod_{i=1}^{N-1} p(y_i|x_i)p(y_i)\right]}{p(\mathcal{D})} \\ &= p(y_N|x_N)p_{N-1}(\vec{y})\frac{p(y_N)}{p(\mathcal{D})}\end{aligned}$$

Diese Rekursionsformel lässt sich nun wiederum in $q_i, R_{i,j}$ und Z einsetzen, sodass sich schließlich zur Berechnung von $\mathcal{N}(\widehat{f}(\vec{x}), var(\vec{x}))$ eine einfache Rekursionsformeln ergibt.
An dieser Stelle sei darauf hingewiesen, dass die bisherige Herleitung lediglich die bedingte Unabhängigkeit der Daten voraussetzt und ansonsten exakt ist. Bezeichne $X_{(t)}$ die Beob-

achtungsmatrix der ersten t Beispiele, so ergibt sich für die Beobachtung $\vec{x}$ mit Label y folgende Rekursionsformel:

$$
\begin{aligned}
\vec{s}_{(t+1)} &= \begin{bmatrix} C_{(t)}K(\vec{x}, X_{(t)}) \\ 1 \end{bmatrix} = \begin{bmatrix} \vec{c}_1 k(\vec{x}, \vec{x}_1) \\ \vdots \\ \vec{c}_t k(\vec{x}, \vec{x}_t) \\ 1 \end{bmatrix} = \begin{bmatrix} s_1 \\ \vdots \\ s_t \\ 1 \end{bmatrix} \\
q_{(t+1)} &= \frac{y_{t+1} - \widehat{f}_{(t)}(\vec{x})}{\sigma_n^2 + var_{(t)}^2(\vec{x})} \\
r_{(t+1)} &= -\frac{1}{\sigma_n^2 + var_{(t)}^2(\vec{x})} \\
\vec{\alpha}_{(t+1)} &= \begin{bmatrix} \vec{\alpha}_{(t)} \\ 0 \end{bmatrix} + q_{(t+1)}\vec{s}_{(t+1)} = \begin{bmatrix} \alpha_1 + q_{(t+1)}s_1 \\ \vdots \\ \alpha_t + q_{(t+1)}s_t \\ q_{(t+1)}s_t \end{bmatrix} \\
C_{(t+1)} &= \begin{bmatrix} C_{(t)} & 0 \\ 0 & 0 \end{bmatrix} + r_{(t+1)}\vec{s}_{(t+1)}\vec{s}_{(t+1)}^T \\
&= \begin{bmatrix} c_{11} + r_{(t+1)}s_1^2 & \dots & c_{1t} + r_{(t+1)}s_1 s_t & r_{(t+1)}s_1 \\ c_{21} + r_{(t+1)}s_1 s_2 & \dots & c_{2t} + r_{(t+1)}s_2 s_t & r_{(t+1)}s_2 \\ \vdots & \ddots & \vdots & \vdots \\ c_{t1} + r_{(t+1)}s_1 s_t & \dots & c_{tt} + r_{(t+1)}s_t^2 & r_{(t+1)}s_t \\ r_{(t+1)}s_1 & \dots & r_{(t+1)}s_t & 1 \end{bmatrix}
\end{aligned}
\tag{5.1}
$$

Für den ersten Rekursionsschritt, d.h. $t = 0$ ergeben sich keinerlei Werte für $\alpha_{(t)}$ und $C_{(t)}$, sodass die entsprechende Einträge verschwinden. Konkret gilt $\alpha_{(0)} = q_{(1)}$ und $C_{(0)} = r_{(1)}$. Die Vorhersagen nach t Schritten erfolgt dabei über:

$$\widehat{f}_{(t)}(\vec{x}) = K(\vec{x}, X_t)\vec{\alpha}_t \tag{5.2}$$

$$var_{(t)}(\vec{x}) = k(\vec{x}, \vec{x}) + K(X_t, \vec{x})C_{(t)}K(\vec{x}, X_t) \tag{5.3}$$

Intuitiv bildet $\alpha_{(t)}$ also das Modell der Vorhersage und $C_{(t)}$ modelliert die Varianz der Vorhersage nach den ersten t Trainingsbeispielen. Der Vektor $\vec{s}_{(t)}$ bezeichnet beim Hinzufügen der Beobachtungen $\vec{x}$ die Änderungsrichtung, in welcher das Modell und die Varianz geändert werden müssen. Damit ergibt sich in Analogie zu einem klassischen Newton-Abstieg eine Änderungsrichtung $\vec{s}_{(t)}$ mit einem Änderungsfaktor $q_{(t)}$ und $r_{(t)}$ für die Parameter α und C.

Betrachtet man einen vollständigen $\mathcal{GP}$ mit $m(\cdot) = 0$, so ergibt sich hier eine Ähnlichkeit mit:

$$\begin{aligned}\vec{\alpha}_{(N)} &\approx [K(X,X) + I_N\sigma_n^2]^{-1}(\vec{y} - \vec{m})\\ C_{(N)} &\approx -[K(X,X) + I_N\sigma_n^2]^{-1}\end{aligned}$$

Durch die Berechnung von $\vec{s}_{(t)}\vec{s}_{(t)}^T$ hat das Verfahren eine Laufzeit von $\mathcal{O}(t^2)$ im Schritt t, was zu einer Gesamtlaufzeit von $\mathcal{O}(N^3)$ bei N Beispielen führt. Um dieses Problem zu umgehen, betrachten die Autoren nicht mehr alle zuvorher gesehenen Beispiele $\vec{x}_i$, sondern Speichern lediglich τ Beispiele als Basisvektoren ab. Dadurch ist jeder Schritt mit einer Laufzeit von $\mathcal{O}(\tau^2)$ durchführbar, was zu einer Gesamtlaufzeit von $\mathcal{O}(N\tau^2)$ führt.
Es stellt sich die Frage, inwiefern die Menge der Basisvektoren begrenzt werden kann und wie die Parameter $\vec{\alpha}_{(t)}$ und $C_{(t)}$ dennoch im Sinne einer optimalen Approximation gewählt werden sollen.
Es ist zunächst zu bemerken, dass α und C sowohl vor dem Löschen eines Basisvektors, als auch nach dem Löschen eins Basisvektors die Parameter für eine Normalverteilung darstellen. Der Unterschied zwischen diesen beiden Verteilungen lässt sich ausnutzen, um den zu löschenden Basisvektoren geschickt zu wählen.
Hierzu sei zunächst angenommen, es gäbe zwei Parametersätze $P_1 = (\vec{\alpha}, C)$ und $P_2 = (\vec{\beta}, D)$, die die gleiche Dimension haben und auf der selben Menge $\mathcal{B}$ von Basisvektoren definiert sind[1]. Diese beiden Approximationen korrespondieren dann zu zwei verschiedenen Normalverteilungen mit leicht unterschiedlichem Mittelwert und einer leicht unteriedlichen Varianz. Der Unterschied zwischen diesen beiden Verteilungen wird durch KL-Divergenz beschrieben (vgl. [Csa02], Kapitel 3.2):

$$\begin{aligned}2KL(P_1||P_2) &= (\vec{\beta} - \vec{\alpha})(D + K(\mathcal{B},\mathcal{B})^{-1})^{-1}(\vec{\beta} - \vec{\alpha}) + tr[(C - D)(D + K(\mathcal{B},\mathcal{B})^{-1})^{-1}]\\ &\quad -ln|(C + K(\mathcal{B},\mathcal{B})^{-1})(D + K(\mathcal{B},\mathcal{B})^{-1})^{-1}| \qquad (5.4)\end{aligned}$$

Seien nun $\vec{\alpha}$ und C diejenigen Parameter der Verteilung zum Zeitpunkt $t + 1$ vor dem Löschen und $\vec{\beta}$ und D diejenigen Parameter der Verteilung zum selben Zeitpunkt, jedoch nach dem Löschen eines Basisvektors aus $X_{(t+1)}$. Dann kommt das Löschen dem Setzen der entsprechenden Einträge in $\vec{\beta}$ und D zu Null gleich, wodurch sich nun der Beitrag der einzelnen Beispiele zur KL-Divergenz berechnen lässt.

[1] Ist dies nicht der Fall, können beide Basismengen vereinigt werden und die entsprechenden Einträge in $\vec{\alpha}$ und C, respektiv $\vec{\beta}$ und D auf 0 gesetzt werden.

Im Folgenden ist auf die explizite Angabe des Zeitstempels $t+1$ verzichtet, sodass der Subindex die entsprechende Position in den Vektoren bzw. Matrizen angibt. Seien außerdem $Q = K(X_{(t+1)}, X_{(t+1)})^{-1}$ und $S = (C^{-1} + Q^{-1})^{-1}$, dann ergibt sich für den Beitrag des i-ten Beispiels zur KL-Divergenz folgende Scorefunktion:

$$\varepsilon(i) = \frac{\alpha_i^2}{Q_{ii} + C_{ii}} - \frac{S_{ii}}{Q_{ii}} + ln\left(1 + \frac{C_{ii}}{Q_{ii}}\right) \tag{5.5}$$

Anschließend kann derjenige Vektor entfernt werden, der den geringsten Score hat, sodass sich beide Verteilungen weiterhin maximal ähnlich sind.
Um die Berechnung von Q und S zu vermeiden, argumentieren die Autoren zunächst, dass die letzten beiden Summanden bei einer Regression im Verlaufe des Trainings gegen Null streben (vgl. [Csa02] Kapitel 3.4), womit sich folgende approximierte Scorefunktion ergibt:

$$\varepsilon(i) = \frac{\alpha_i^2}{Q_{ii} + C_{ii}} \tag{5.6}$$

Um die rechenintensive Matrixinvertierung von $K(X_{(t+1)}, X_{(t+1)})^{-1}$ zu vermeiden, kann eine iterative Invertierung mit dem Matrix-Inversions-Lemma erfolgen:

$$\begin{aligned} Q_{(t+1)} &= \begin{bmatrix} Q_{(t)} & 0 \\ 0 & 0 \end{bmatrix} + \gamma_{(t+1)}^{-1} \begin{bmatrix} Q_{(t)}K(\vec{x}_{t+1}, X_{(t)})K(X_{(t)}, \vec{x}_{t+1})Q_{(t)} & Q_{(t)}K(X_{(t)}, \vec{x}_{(t+1)}) \\ Q_{(t)}K(\vec{x}_{(t+1)}, X_{(t)}) & 1 \end{bmatrix} \\ \gamma_{(t+1)}^{-1} &= [k(\vec{x}_{t+1}, \vec{x}_{t+1}) - K(\vec{x}_{t+1}, X_{(t)})Q_{(t)}K(X_{(t)}, \vec{x}_{t+1}))]^{-1} \end{aligned} \tag{5.7}$$

Auch an dieser Stelle sei angemerkt, dass die Matrix $Q_{(t)}$ zunächst analog zur Matrix $C_{(t)}$ in jedem Schritt um eine Dimension wächst, ihre Maximalgröße jedoch durch die Anzahl der Basisvektoren begrenzt ist.
Abschließend bleibt die Frage zu klären, inwiefern die neuen Parameter $\vec{\alpha}_{(t+1)}$ und $C_{(t+1)}$ nach dem Löschen eines Basisvektors geändert werden müssen. Hierzu nehmen die Autoren an, dass der letzte Basisvektor $\vec{x}$ den geringsten Score hat und daher entfernt werden soll. Dies stellt keine wirkliche Einschränkung dar, da durch Vertauschung der Zeilen und Spalten derjenige Vektor mit dem geringsten Score jederzeit an die Stelle des letzten Basisvektors verschoben werden kann.
Bezeichnen β und D wiederum die Parameter der Approximation nach dem Löschen eines Basisvektors, dann ist der letzte Eintrag in β eine Null und die letzte Zeile und Spalte aus D enthalten ebenfalls nur Nulleinträge. Die übrigen Einträge lassen sich frei wählen und sollten so gewählt sein, dass die durch β und D beschriebene Verteilung möglichst ähnlich zu der durch $\vec{\alpha}_{(t+1)}$ und $C_{(t+1)}$ beschriebenen Verteilung ist. Die Ähnlichkeit wird erneut durch die KL-Divergenz mit Formel 5.4 beschrieben, sodass durch eine Minimierung dieser Funktion eine entpsrechende Anpassung von $\vec{\alpha}_{(t+1)}$ und $C_{(t+1)}$ ableitbar ist. Die Anpassung der inversen Kernelmatrix $Q_{(t+1)}$ erfolgt wieder mit Hilfe des Matrix-Inversions-Lemma. Zusammenfassend ergibt sich folgende Formel zum Löschen des letzten Basisvektors aus

$X_{(t)}$, wobei auch hier auf die Angabe des Zeitstempels der Übersicht halber verzichtet wurde:

$$\begin{aligned}
\vec{\alpha}_{(t+1)} &= \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_t \end{bmatrix} - \frac{\alpha_{t+1}}{C_{t+1,t+1} + Q_{t+1,t+1}} \left(\begin{bmatrix} Q_{1,t+1} \\ \vdots \\ Q_{t,t+1} \end{bmatrix} + \begin{bmatrix} C_{1,t+1} \\ \vdots \\ C_{t,t+1} \end{bmatrix} \right) \\
C_{(t+1)} &= \begin{bmatrix} C_{11} & \dots & C_{1t} \\ \vdots & \ddots & \vdots \\ C_{t1} & \dots & C_{tt} \end{bmatrix} + \frac{1}{Q_{t+1t+1}} \begin{bmatrix} Q_{1,t+1} \\ \vdots \\ Q_{t,t+1} \end{bmatrix} \begin{bmatrix} Q_{1,t+1} \\ \vdots \\ Q_{t,t+1} \end{bmatrix}^T \\
&\quad - \frac{1}{C_{t+1,t+1} + Q_{t+1,t+1}} \left(\begin{bmatrix} Q_{1,t+1} \\ \vdots \\ Q_{t,t+1} \end{bmatrix} + \begin{bmatrix} C_{1,t+1} \\ \vdots \\ C_{t,t+1} \end{bmatrix} \right) \left(\begin{bmatrix} Q_{1,t+1} \\ \vdots \\ Q_{t,t+1} \end{bmatrix} + \begin{bmatrix} C_{1,t+1} \\ \vdots \\ C_{t,t+1} \end{bmatrix} \right)^T \\
Q_{(t+1)} &= \begin{bmatrix} Q_{11} & \dots & Q_{1t} \\ \vdots & \ddots & \vdots \\ Q_{t1} & \dots & Q_{tt} \end{bmatrix} - \frac{1}{Q_{t+1t+1}} \begin{bmatrix} Q_{1,t+1} \\ \vdots \\ Q_{t,t+1} \end{bmatrix} \begin{bmatrix} Q_{1,t+1} \\ \vdots \\ Q_{t,t+1} \end{bmatrix}^T
\end{aligned} \tag{5.8}$$

Die Matrix $Q_{(t+1)}$ bildet nach dem Löschen eines Vektors aus $X_{(t+1)}$ auch weiterhin die korrekte Inverse für $K(X_{(t+1)}, X_{(t+1)})$, wohingegen $\vec{\alpha}_{(t+1)}$ und $C_{(t+1)}$ sich nicht eindeutig aus der aktuellen Matrix der Basisvektoren $X_{(t+1)}$ ergeben, sondern auch Informationen von bereits gelöschten Basisvektoren enthalten können.

Die hier vorgestellte Methode nimmt implizit an, dass die Wahrscheinlichkeitsdichte p eine Normalverteilung ist (siehe Lemma 1). Im Falle der Klassifikation (vgl. Abschnitt 4.3.2) ist dieser Umstand wegen der Nutzung einer Sigmoidfunktion nicht mehr gegeben, sodass die bisherigen Überlegungen nicht mehr gültig sind. Um Projection GP dennoch für die Klassifikation anwenden zu können, schlagen die Autoren daher vor, die nach jedem Schritt t berechnete Verteilung in eine Normalverteilung umzuwandeln. Hierzu nutzen die Autoren abermals die KL-Divergenz, um die im Sinne der KL-Divergenz passendste Normalverteilung zu ermitteln.

Aus praktischer Sicht muss so lediglich ein Projektionsschritt in den Algorithmus eingeführt werden, wohingegen die übrigen Überlegungen weiter gültig bleiben.

5.3 Diskussion

Projection GP ist ein Online Algorithmus, der die Parameter einer konditionalisierten Normalverteilung iterativ approximiert um so einen vollständigen Gauß-Prozess zu approximieren.

Durch die Nähe von Gauß-Prozessen zum k-NN Verfahren bietet sich mit Projection GP zusätzlich ein Online Algorithmus zur Berechnung von k-NN bei geeigneter Wahl der Kernfunktion an, der einen konstanten Speicherbedarf hat.

Insgesamt benötigt Projection GP drei Parameter. Analog zu einem vollständigen Gauß-Prozess müssen die Kernfunktion $k(\cdot, \cdot)$ und der Messfehler σ_n anwendungsspezifisch gewählt werden. Die Kernfunktion beschreibt auch hier die Ähnlichkeit der Beispiele zueinander, wohingegen der Messfehler σ_n nicht nur Fehler in den Messungen modelliert, sondern auch ein gewisse numerische Stabilität des Verfahrens gewährleisten kann (vgl. Abschnitt 4.4). Zusätzlich muss bei Projection GP auch die Anzahl der Basisvektoren τ vom Benutzer angegeben werden. Diese Angabe hat zunächst Einfluss auf die Vorhersagegüte und die Laufzeit (siehe Kapitel 7). Des Weiteren kann die Anzahl der Basisvektoren als eine Art Regularisierung benutzt werden, da der Modellvektor so auf τ Einträge beschränkt wird und damit ein simples Auswendiglernen der Daten vermieden werden kann. Diese Regularisierung wird jedoch durch einen Benutzerparameter und nicht durch das Verfahren selbst erreicht. Ist der Benutzerparameter schlecht gewählt, so hat dies entsprechende Auswirkungen auf die Vorhersagegüte. Andere Verfahren, wie z.B. die SVM modellieren eine Regularisierung im Verfahren selbst, sodass während des Trainings eine optimale Modellgröße gewählt wird.
Um diesen Unterschied zu Methoden mit Regularisierung etwas zu entschärfen, führen die Autoren in [CO02] einen Neuheitsgrenzwert ξ ein. Hierbei argumentieren die Autoren, dass der Änderungsfaktor der inversen Kernmatrix γ während des Trainings im gewissen Sinne die Neuheit des Beispiels $\vec{x}$ repräsentiert. Daher sollte ein Beispiel nur dann in die Menge der Basisvektoren X aufgenommen werde, wenn es ausreichend neue Informationen enthält, also $\gamma > \xi$ gilt. Bei der Anwendung von Projection GP ließe sich dann τ vergleichsweise hoch wählen, wobei das Verfahren auf Basis von ξ nur dann ein Beispiel in die Menge der Basisvektoren hinzufügt, wenn dieses neue Informationen enthält, sodass hier eine gewisse Form der Regularisierung durchgeführt wird.
Dieses Vorgehen birgt jedoch zum einen das Problem, dass die Anzahl der Basisvektoren nicht während des Trainings erhöht werden kann, falls dies nötig ist. Zum anderen argumentieren die Autoren selbst, dass dieses Vorgehen vor allem die numerische Stabilität, aber nicht unbedingt die Vorhersagegüte erhöht. Zu guter Letzt stellt sich aus praktischer Sicht die Frage, inwiefern ξ gewählt werden muss. Gerade hier zeigt sich ein großes Problem dieses Vorgehens, da kaum eine Intuition über γ und die Einträge in einer inversen Kernmatrix für eine Kernfunktion k gegeben werden kann. Damit ist es also schwierig ξ sinnvoll zu wählen.

6 | Implementierung

In Kapitel 3 wurde die Funktionsweise von FPGAs detailliert beschrieben und auf die Besonderheiten bei der Programmierung eingegangen. Kapitel 4 hat dann die Methode der vollständigen Gauß-Prozesse erklärt, woraufhin Kapitel 5 verschiedene Algorithmen zur Anwendung von Gauß-Prozessen vorgestellt hat. Im folgenden Kapitel soll eine Implementierung von Projection GP auf dem FPGA erfolgen, wobei hier auch auf die Umsetzung des übrigen Gesamtsystems eingegangen wird.
Abschnitt 6.1 geht hierbei zunächst auf die Implementierung von Projection GP auf klassischer Hardware ein, sodass dann Abschnitt 6.2 die Implementierung von Projection GP auf einem FPGA vorstellt.

6.1 Implementierung in einer Hochsprache

Bevor die Implementierung von Projection GP auf einem FPGA vorgestellt wird, soll zunächst kurz auf die Implementierung in einer Hochsprache eingegangen werden. Um eine gewisse Allgemeingültigkeit zu gewährleisten, werden hier nur übliche Sprachkonstrukte einer Hochsprache wie Schleifen oder Felder verwendet. Die nachfolgenden Überlegungen sollten sich daher problemlos auf prozeduale und objektorientierte Sprachen wie `C,C++` oder `Java` anwenden lassen.

Speicherlayout

Zentraler Bestandteil von Projection GP ist der Parametervektor α und die Matrizen C und Q. Diese Parameter wachsen dabei in jedem Schritt um eine Dimension an, bis die Maximaldimension τ bzw. τ^2 erreicht ist. Dabei fügt der Algorithmus zunächst jedes Beispiel in die Menge der Basisvektoren ein und erhöht damit α, C und Q in jedem Fall um eine Dimension. Erst danach wird ein Basisvektor gelöscht, sodass α, C und Q wieder um eine Dimension schrumpfen. Damit benötigt α im ungünstigsten Fall eine Größe von $\tau + 1$ Elementen, wohingegen C und Q Platz für $(\tau + 1)^2$ Elemente benötigen.
Durch die sich vergrößernde Natur dieser Variablen bieten sich zunächst dynamische Datenstrukturen wie z.B: eine Liste zur Implementierung an. Dies hätte den Vorteil, dass neue Einträge in α bzw. C und Q lediglich an die Liste angehängt werden müssen und in

jedem Schritt genau soviel Speicher verbraucht wird, wie notwendig ist. Zum Entfernen müssen dann zunächst Listeneinträge vertauscht und anschließend die letzten Einträge gelöscht werden. Das Löschen von Listeneinträgen ist trivial, wohingegen das Vertauschen schwieriger ist, da hier die Listenzeiger entsprechend angepasst werden müssen.
Da die Maximalgröße von α, C und Q bereits statisch zur Kompilierzeit bekannt ist, kann auf einfache Felder der Größe $\tau+1$ bzw. $(\tau+1)^2$ zurückgegriffen werden. Diese Felder belegen dann zwar zu Beginn des Algorithmus - bevor die Menge der Basisvektoren vollständig befüllt wurde - mehr Speicher als eine einfache Listenimplementierung, bieten dafür jedoch konstante Zugriffszeiten auf jedes Element im Feld an. Zusätzlich wird keinerlei Verwaltungsmehraufwand durch Zeiger betrieben, was den Speicherbedarf insgesamt nochmals verringert und Zugriffszeiten beschleunigt.
Um hier bereits der Implementierung auf einem FPGA möglichst nahe zu kommen, wird der Vektor α und die Matrizen C, Q jeweils als ein kontinuierliches Feld im Speicher abgebildet. Sei der erste Eintrag in einem Feld mit Null indexiert, so ergibt sich der Zugriff auf die Elemente eines Vektors auf natürliche Art und Weise, wohingegen eine Matrix M wie folgt im Speicher organisiert sei:

$$\begin{pmatrix} m_{0,0} & m_{0,1} & \cdots & m_{0,\tau} \\ m_{1,0} & m_{1,1} & \cdots & m_{1,\tau} \\ \vdots & \vdots & \ddots & \vdots \\ m_{\tau,0} & m_{\tau,1} & \cdots & m_{\tau,\tau} \end{pmatrix} \longrightarrow \begin{array}{|c|c|c|c|c|c|c|c|c|} \hline m_{0,0} & m_{0,1} & \cdots & m_{0,\tau} & m_{1,0} & m_{1,1} & \cdots & m_{1,\tau} & \cdots \\ \hline \end{array}$$

Abbildung 6.1: Umsetzung einer Matrix als kontinuierliches Feld im Speicher.

Der Index des Elementes $m_{i,j}$ einer $(\tau + 1) \times (\tau + 1)$ Matrix im korrespondierenden Feld lässt sich dann wie folgt berechnen:

$$index(i, j) = i \cdot (\tau + 1) + j$$

Diese Art der Implementierung scheint zunächst konterintuitiv, da ein zweidimensionales Feld eine Matrix auf direktem Wege abbildet und ohne zusätzliche Indexberechnung auskommt. Dem Gegenüber steht der Umstand, dass ein Speicher üblicherweise nicht zweidimensional ist, sondern als ein großes, kontinuierliches Feld mit aufsteigendem Adressraum ausgelegt ist. Tatsächlich führen Compiler implizit eine entsprechende Transformation zweidimensionaler Felder zu eindimensionalen Feldern auf dem Stack durch und implementieren die gezeigte Indexberechnung (vgl. Abschnitt 7.5 in [CT11]). Im Hinblick auf die Performanz einer Implementierung in einer Hochsprache hat diese Indexberechnung also keinerlei Einfluss, weshalb sie durchaus auch weggelassen werden kann. Im Hinblick auf die Implementierung auf einem FPGA hat diese Implementierung jedoch den Vorteil, dass Blockram besser ausgenutzt werden kann (siehe Abschnitt 6.3.1).

Neben dem Vektor α und den Matrizen C und Q benötigt das Verfahren auch eine Datenstruktur für die Mengen der Basisvektoren X. Diese lässt sich in analoger Art und Weise als Matrix implementieren.
Zusätzlich werden drei statische Felder der Größe $\tau+1$ als temporäre Variablen verwendet. Der Vektor $\vec{s}$ wird dabei in einem Feld s gespeichert und die Kernfunktionswerte $K(X,\vec{x})$ der Basisvektoren X zu dem aktuellen Beispiel $\vec{x}$ seien in einem Feld k gespeichert. Zu guter Letzt speichert das Feld e die Einträge zur Änderung der Matrix Q, d.h. $Q \cdot K(\vec{x}, X)$.

Algorithmenlayout

Die Rekursionsformeln von Projection GP wurden in Abschnitt 5.2.1 vorgestellt. Die mathematische Matrix-Vektor-Notation eignet sich jedoch nur bedingt für eine Implementierung, weshalb der Algorithmus hier zunächst in Summenschreibweise umformuliert wird. In Anlehnung an die Implementierung in einer Hochsprache sei auch hier bereits eine Indexierung beginnend bei Null verwendet. Bezeichne α, s, e, C und Q die entsprechenden Felder wie bereits diskutiert, sowie `bvCnt` die aktuelle Anzahl der Basisvektoren mit `bvCnt` $\leq \tau$. Des Weiteren bezeichne $\vec{x}$ die aktuell bearbeitete Beobachtung mit Label y, dann zeigt Algorithmus 1 den Ablauf zum Hinzufügen eines Beispieles bei Projection GP.

Algorithmus 1 Projection GP: Füge Beispiel $(\vec{x}, y)$ hinzu

```
1:  for i = 0, ..., bvCnt − 1 do
2:      k_i = k(x⃗, x⃗_i)
3:      s_i = Σ_{j=0}^{bvCnt−1} C_{i,j} · k_j
4:      e_i = Σ_{j=0}^{bvCnt−1} Q_{i,j} · k_j
5:  end for
6:  m = Σ_{j=0}^{bvCnt−1} k_j · α_i + m(x⃗)
7:  σ_2 = Σ_{i=0}^{bvCnt−1} s_j · k_j + k(x⃗, x⃗)
8:  q = (y_t − m) · (σ_2 + σ_n)^{−1}
9:  r = −(σ_2 + σ_n)^{−1}
10: γ = − Σ_{i=0}^{bvCnt−1} e_i · k_i
11: α_t = s_bvCnt · q
12: Q_{bvCnt,bvCnt} = γ^{−1}
13: C_{bvCnt,bvCnt} = r
14: for i = 0, ..., bvCnt − 1 do
15:     α_i = α_i + s_i · q
16:     Q_{bvCnt,i} = Q_{i,bvCnt} = −e_i · γ^{−1}
17:     C_{bvCnt,i} = C_{i,bvCnt} = s_i · r
18:     for j = 0, ..., bvCnt − 1 do
19:         C_{i,j} = C_{i,j} + s_i · s_j · r
20:         Q_{i,j} = Q_{i,j} + e_i · e_j · γ^{−1}
21:     end for
22: end for
```

Der Index des zu löschenden Basisvektors kann direkt aus Formel 5.6 berechnet werden, wobei hier insbesondere gilt `bvCnt` $= \tau$. Des Weiteren müssen die Einträge in α und die Zeilen und Spalten in C und Q so getauscht werden, dass der zu löschende Basisvektor zuletzt hinzugefügt wurde. Bezeichne hierzu `tausche`$(A_{i,j}, A_{i,k})$ und `tausche`$(A_{j,i}, A_{k,i})$ eine entsprechende Tauschfunktion, die Zeile bzw. Spalte j und Zeile bzw. Spalte k der Matrix A miteinander vertauscht. Analog tausche die Funktion `tausche`(a_i, a_j) die Einträge i und j im Vektor a miteinander. Algorithmus 2 zeigt die Scoreberechnung inklusive

Algorithmus 2 Projection GP: Berechne Scorefunktion und bereite löschen vor

1: $d = 0$
2: $s = \frac{\alpha_0}{Q_{0,0}C_{0,0}}$
3: **for** $i = 0, \ldots, \tau - 1$ **do**
4: $t = \frac{\alpha_i}{Q_{i,i}C_{i,i}}$
5: **if** $t < s$ **then**
6: $s = t$
7: $d = i$
8: **end if**
9: **end for**
10: tausche(α_τ, α_d)
11: **for** $i = 0, \ldots, \tau - 1$ **do**
12: tausche$(C_{i,d}, C_{i,\tau})$
13: tausche$(Q_{i,d}, Q_{i,\tau})$
14: **for** $i = 0, \ldots,$ `bvCnt` **do**
15: tausche$(C_{d,i}, C_{\tau,i})$
16: tausche$(Q_{d,i}, Q_{\tau,i})$
17: **end for**
18: **end for**

Vertauschung.
Schlussendlich müssen zum Löschen des Basisvektors die Felder α, C und Q mittels Formel 5.8 aktualisiert werden, was in Algorithmus 3 zu sehen ist.

Algorithmus 3 Projection GP: Lösche Basisvektor

1: **for** $i = 0, \ldots, \tau - 1$ **do**
2: $\alpha_i = \alpha_i - \frac{C_{\tau,i}+Q_{\tau,i}}{C_{\tau,\tau}+Q_{\tau,\tau}} \cdot \alpha_M$
3: **for** $j = 0, \ldots, \tau - 1$ **do**
4: $C_{i,j} = C_{i,j} + \frac{Q_{\tau,i}Q_{\tau,j}}{Q_{\tau,\tau}} - \frac{C_{\tau,i}C_{\tau,j}+C_{\tau,i}Q_{\tau,j}+Q_{\tau,i}C_{\tau,j}+Q_{\tau,i}Q_{\tau,j}}{C_{\tau,\tau}+Q_{\tau,\tau}}$
5: $Q_{i,j} = Q_{i,j} - \frac{Q_{\tau,i}Q_{\tau,j}}{Q_{\tau,\tau}}$
6: **end for**
7: **end for**

6.2 FPGA Systementwurf

In diesem Kapitel soll nun auf Grundlage der zuvor vorgestellten Implementierung die Umsetzung von Projection GP auf einem FPGA vorgestellt und diskutiert werden.
Eine Implementierung für ein FPGA unterscheidet sich vor allem dadurch von einer Hochsprachenimplementierung, dass ein FPGA keinerlei Laufzeitumgebung oder andere Abstraktionen wie ein Dateisystem, einen Stack oder einen Adressraum anbietet. Ein FPGA bietet lediglich die Möglichkeit zur Umsetzung von logischen Funktionen, sowie Schnittstellen zu einigen Standardkomponenten wie Blockram und DSP Elementen. Des Weiteren enthalten FPGA Entwicklerboards häufig weitere Schnittstellen zur Ein- und Ausgabe wie Ethernet oder eine serielle Schnittelle. Zusätzlich ist auf vielen FPGA Boards weiterer Speicher in Form von `DDR RAM` zu finden.
Aus diesem Grund handelt es sich bei der Programmierung eines FPGAs nicht nur um die Implementierung des gewünschten Algorithmus, sondern zusätzlich um das Design eines

ganzen Systems. Hierbei müssen insbesondere auch die Datenkommunikation, Taktraten der einzelnen Subsysteme, sowie die Verbindung einzelner Module untereinander in Betracht gezogen werden.
Der nächste Abschnitt beschreibt kurz das verwendete Enwicklerboard und die zugehörige Software, wohingegen die darauffolgenden Abschnitte auf einzelne Teile der Systemarchitektur eingehen.

6.2.1 Verwendete Hard- und Software

Für die vorliegende Arbeit stand das `Artix-7 AC701 Evaluation Kit` von der Firma `Xilinx` zur Verfügung. Dieses Entwicklerboard enthält den `XC7A200T-2FBG676C` FPGA Chip der Firma `Xilinx` mit insgesamt 215360 Logikzellen, 269200 Flipflops, 13149Kb zusätzlichem Blockram, sowie 740 DSP-Einheiten. Die Logikzellen (LUT) der Firma `Xilinx` können anders als in Kapitel 3.1 gezeigt bis zu sechs Eingaben und eine Ausgabe verarbeiten und dienen üblicherweise nicht als Flipflop. `Xilinx` vier Logikzellen mit Acht zusätzlichen Flipflops in einem sogenannte Slice zusammen. Ein Slice kann so gleichzeitig zum Teil als Logik und zum Teil als Speicher genutzt werden, wobei bei voller Ausnutzung aller Slices maximal 2888Kb Speicher realisieren werden kann [Xili].
Der Blockram ist in statischen RAM-Zellen mit jeweils 36Kb auf dem FPGA Chip implementiert. Jede RAM-Zelle enthält zwei Ein-/Ausgabeports, womit es möglich ist, gleichzeitig auf zwei logische voneinander getrennte, 18Kb große Adressräume zuzugreifen. Die DSP-Einheiten enthalten jeweils einen pre-adder[1], einen Multiplizierer, einen Addierer, sowie einen Akkumulator, die für Berechnungen auf dem FPGA Chip genutzt werden können [Xilb].
Das Entwicklerboard selbst enthält eine Reihe verschiedener Ein- und Ausgabemöglichkeiten. Neben Ethernet, PCIe und einer seriellen Schnittelle finden sich auch analoge Ein- und Ausgabeports, sowie ein HDMI Ausgang auf dem Board. Es gibt eine Vielzahl von Knöpfen und Schaltern und ein kleines LCD Display. Zusätzlich finden sich 1 GB `DDR` Speicher auf dem Entwicklerboard. Die Programmierung des FPGAs erfolgt über eine USB JTAG Schnittstelle [Art].
Passend zu dem Entwicklerboard bietet die Firma `Xilinx` das Design- und Synthese-Tool `Vivado` an [Viva]. Aufbauend auf der Skriptsprache `TCL` [TCL] bietet das Tool eine Schnittstelle zur Programmierung von HDL-Code, zur Integration von Intellectual Property in das Design, sowie eine Möglichkeit zur eigentlichen Synthese mit diverse Optimierungseinstellungen und der darauffolgenden Implementierung des Bitstreams auf dem FPGA an. Zusätzlich kann das Tool durch Ausnutzung der internen FPGA Charakteristika Abschätzungen zu Signallaufzeiten und dem Energieverbrauch der Implementierung machen.
Ergänzend zum eigentlichen Synthese-Tool bietet `Xilinx` außerdem `Vivado High Level`

[1]Dieser führt die Addition vor dem nachfolgenden Befehl aus.

`Synthesis` als Werkzeug an [Vivb]. Dieses Tool ermöglicht die automatische Übersetzung von `C,C++` oder `SystemC` Code in `VHDL` oder `Verilog` Code, wobei dabei speziell die Eigenschaften der `Xilinx` FPGAs, wie z.B: Blockram oder DSP Einheiten ausgenutzt werden. Beide Tools wurden im Laufe dieser Arbeit in Version `2015.2` verwendet.

6.2.2 Systemarchitektur

Wie bereits angedeutet, handelt es sich bei der Programmierung von FPGAs nicht nur um die Programmierung mit einer HDL, sondern zusätzlich um das Design der eigentlichen Ausführungsumgebung.
Um das Design einfach und modular zu halten, aber dennoch eine möglichst gute Integration in existierende Systeme zu gewährleisten, wurde daher auf den intensiven Einsatz von Intellectual Property zurückgegriffen die vom Hersteller `Xilinx` für ihre FPGAs freigegeben wurde.
Hier stellt sich zunächst die Frage, inwiefern neue Beobachtungen $\vec{x}$ an den Algorithmus auf dem Board kommuniziert werden und inwieweit mit passende Vorhersagen $\widehat{y}$ geantwortet werden kann.
Das verwendete Entwicklerboard stellt mit PCIe und Ethernet zwei verschiedene Schnittstellen zur Kommunikation bereit. PCIe ist hierbei für den Einbau des FPGAs in einen Host-PC gedacht, wohingegen Ethernet für den Betrieb als Einzelgerät genutzt wird. Damit bietet PCIe höhere Datenübertragungsraten bei einem höheren Stromverbrauch, wohingegen Ethernet niedrigere Datenübertragungsraten mit geringerem Stromverbrauch bei einem Betrieb als Einzelsystem anbietet. Durch seinen niedrigen Stromverbrauch und den Betrieb als Einzelsystem eignet sich Ethernet für den Bereich der eingebetteten Systeme besser (vgl. [Dun]), sodass im Laufe dieser Arbeit der TCP/IP Protokollstapel über Ethernet als Kommunikationstechnologie verwendet wird.
Um die Implementierung des TCP/IP Protokollstapels in Hardware zu umgehen, kommt der `MicroBlaze` Soft-Prozessor der Firma `Xilinx` zum Einsatz. Dieser setzt einen vollständigen Mikroprozessor auf dem FPGA um, welcher mit `C` programmiert wird. Damit können bereits existierenden TCP/IP Stack Implementierung benutzt werden, sodass lediglich das Anwendungsprotokoll selbst implementiert werden muss. Das resultierende Programm kann dann anschließend mit Hilfe der USB Schnittstelle auf das FPGA übertragen werden, wo es Datenpakete entgegennimmt und die Hardwareimplementierung von Projection GP passend ausführt.
Zusätzlich zu Ethernet wird eine serielle Schnittstelle zur Ausgabe und Fehlerbehandlung verwendet.
Abbildung 6.2 zeigt eine schematische Darstellung der FPGA Systemarchitektur, wobei im Folgenden auf die Einzelheiten der einzelnen Teilsystem eingegangen wird.

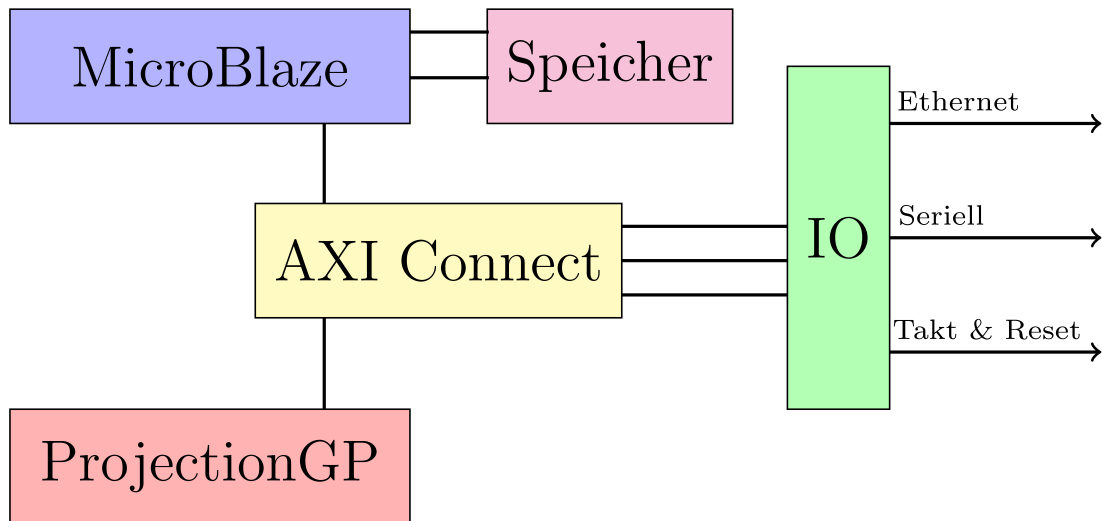


Abbildung 6.2: Schematische Darstellung der FPGA Architektur.

MicroBlaze Prozessor

Der `MicroBlaze` Soft-Prozessor der Firma `Xilinx` wurde speziell für den Einsatz auf `Xilinx` FPGAs entwickelt. Er setzt eine Harvard-Achitektur um, in der Daten- und Instruktionsspeicher getrennt voneinander sind, sodass Daten und Intruktionen gleichzeitig aus dem Speicher geladen werden.
Der verwendete RISC Befehlssatz wurde in Anlehnung an den DLX Befehlssatzes aus [HP11] entwickelt, welcher wiederum auf dem bekannten MIPS Befehlssatz basiert. Damit nutzt der `MicroBlaze` Prozessor eine einfache 32 Bit Load/Store Architektur und kann in den meisten Fällen eine Instruktion pro Taktzyklus abarbeiten. Im Juni 2009 wurde der `MicroBlaze` Prozessor in den Linux Kernel aufgenommen und wird seit Version 4.6 vom `gcc` unterstützt [GCC].
Da der `MicroBlaze` Prozessor als Soft-Prozessor mit Hilfe der Logikzellen des FPGAs implementiert wird, kann er in vielerlei Hinsicht konfiguriert werden. Ziel der vorliegenden Arbeit ist es, eine energiesparende Implementierung zu erstellen, die dennoch eine vergleichbare Geschwindigkeit zu handelsüblicher Hardware bietet. Im gewählten Szenario muss der Soft-Prozessor lediglich den TCP/IP Protokollstapel abarbeiten und einzelne Datenpakete entpacken. Aus diesem Grund wird der Prozessor möglichst ressourcenschonend konfiguriert. Das bedeutet insbesondere das Abschalten aller Gleitkomma-Recheneinheiten, sowie das Weglassen jeglicher Ganzzahlendivision. Da kein Betriebssystem zum Einsatz kommt, kann die Unterbrechungsbehandlung auch abgeschaltet werden. Um dennoch eine schnelle Entwicklung zu ermöglichen, wird eine Debugeinheit verwendet, die den Prozessorstatus, wie z.B. Programmzähler und Registerinhalte an einen externen PC weiterleitet.
Um die Verwendung des `MicroBlaze` Soft-Prozessor zu erleichtern, bietet `Xilinx` die in-

tegrierte Entwicklungsumgebung `Vivado SDK` basierend auf `eclipse` an [Vivc]. Das Tool wird so genutzt, dass es den durch `Vivado` generierten Bitstrom für das FPGA zunächst lädt und anhand von Metadaten erkennt, ob ein `MicroBlaze` Prozessor verwendet wird. Ist dies der Fall, so werden die verwendeten Submodule wie Ethernet oder die Serielle Schnittelle in den Speicherbereich des Prozessors eingebunden. Dann werden sogenannte Board-Support-Packages vorkompiliert, die Treiberdefinitionen für die eingebundenen Module enthalten. Auf diesem Wege wird eine speicherbezogene Adressierung der Peripherie (Memory Mapped I/O) implementiert, sodass Entwickler über Macrofunktionen auf die Ein- und Ausgabeport der Peripherie zugreifen können. Die Zuweisung der korrekten Adressräume muss hier zuvor bei der Synthese geschehen und kann durch das Synthesetool `Vivado` verifiziert werden. Hier ist insbesondere darauf zu achten, dass Instruktionen- und Datensegment an der Adresse `0x00` starten sollten und die Speicherbereiche für die Peripherie sich nicht überlappen.
Zusätzlich können innerhalb dieser SDK die Aufrufe für den Compiler und Linker zur Kompilation des C-Codes detailliert konfiguriert werden. Wird ein Board-Support-Package umkonfiguriert oder die Implementierung geändert, so muss gegebenenfalls der Linkeraufruf angepasst werden, um die Heap- und Stackgröße anzupassen. Diese sollten mindestens so groß sein, dass alle von der Implementierung verwendeten Heapvariablen auf den Heap passen und alle Funktionsaufrufe, sowie lokale Variable auf den Stack passen. Aus diesem Grund bietet es sich an, den Heap und den Stack zunächst relativ großzügig zu konfigurieren und dann nach der Konfiguration der Board-Support-Packages diese Schrittweise zu verringern. Für die im Folgenden vorgestellte Konfiguration des TCP/IP Protokollstapels Lightweight IP wurde schließlich eine Heap- und Stackgröße von jeweils Rund 20 KB gewählt.

Netzwerkkommunikation

Neben dem maschinellen Lernalgorithmus bildet die Netzwerkkommunikation einen zentralen Bestandteil des FPGA Systems. Aus diesem Grund sei kurz auf die Einzelheiten der Netzwerkkommunikation anhand von [KR08] eingegangen, wobei die folgenden Erläuterungen bewusst kurz gehalten sind.
Netzwerkkommunikation erfolgt in verschiedenen Schichten, wobei jede Schicht eine bestimmte Aufgabe hat und die Anforderungen der darüberliegenden Schicht erfüllt. Tabelle 6.1 zeigt das `ISO-OSI` Referenzmodell für Netzwerkkommunikation, welches insgesamt Sieben verschiedene Schichten definiert. Der TCP/IP Protokollstapel bildet eine Umsetzung dieses Referenzmodells, wobei hier insgesamt nur vier Schichten implementiert sind.
Auf der untersten Ebene definierte Ethernet eine Technolgie zur kabelgebunden, lokalen Datenübertragung (LAN). In der Bitübertragungsschicht beschreibt Ethernet die physikalischen Eigenschaften des Übertragungsmedium, d.h. Leitungsinnenwiderstand, Stecker-

format, sowie Taktrate oder die Codierung. Die Sicherungsschicht sorgt für eine fehlerfreie Datenübertragung, indem Daten in Pakete aufgeteilt werden, die mit Prüfsummen versehen sind. Zusätzlich werden Möglichkeiten zur Flusskontrolle, Kollisionserkennung, sowie Adressierung der Netzteilnehmer durch Media Access Control (MAC) bereitgestellt.
Das Address Resolution Protocol (ARP) dient der darüberliegenden IP Schicht zum Übersetzen der MAC-Adressen der einzelnen Netzteilnehmer in IP-Adressen. Die Vermittlungsschicht nutzt IP-Adressen schließlich um Datenpakete zwischen verschiedenen LANs zu versenden und korrekt zu adressieren. Die Transportschicht kann dann auf Basis der IP-Adressen eine Punkt-zu-Punkt Verbindung zwischen zwei Teilnehmern implementieren. UDP bildet hier ein verbindungsloses Protokoll, welches einzelne Datenpakete ohne Gewähr auf Ankunft bei der Gegenstelle versendet. Dem Gegenüber ist TCP ein verbindungsbehaftetes Protokoll, welches Mechanismen zum sicheren Versand von Datenpaketen beinhaltet, sodass hier die Ankunft eines Paketes bei der Gegenstelle gewährleistet ist.
In der darüberliegenden Anwendungsschicht kann der Benutzer schließlich das eigentliche Anwendungsprotokoll implementieren. Zusätzlich werden auf dieser Ebene Standardprotokolle, wie das Dynamic Host Configuration Protocol (DHCP) angeboten, welches für eine dynamische Vergabe der IP-Adressen im Netzwerk sorgt.

	ISO-OSI Schicht	TCP/IP Stack	Protokoll
7	Anwendung	Anwendung	Anwendungsprotokoll DHCP
6	Darstellung		
5	Sitzung		
4	Transport	Transport	UDP, TCP
3	Vermittlung	Internet	IP
2	Sicherung	Netzzugang	ARP
1	Bitübertragung		Ethernet

Tabelle 6.1: Der TCP/IP Protokollstapel im Vergleich zum ISO-OSI Modell.

TCP bietet im Vergleich zum einfacheren UDP eine sichere Datenübertragung an, sodass sich damit ein ein einfaches Ping-Pong Protokoll implementieren lässt:
Eine Gegenstelle sendet Trainingsbeispiele und Beobachtungen jeweils als einzelne TCP-Pakete an das FPGA und wartet dann auf ein entsprechendes Antwortpaket. Das FPGA entscheidet je nach Kontrollinformationen im Paket, ob es den Hardwareblock des maschinellen Lernalgorithmus im Trainings- oder Vorhersagemodus betreiben muss. Anschließend setzt es ein Antwortpaket mit der eventuellen Vorhersage und einem passenden Kontrollcode zusammen und sendet es an die Gegenstelle.
Abbildung 6.3 zeigt die Organisation der Nutzlast der TCP Pakete. Einfache Gleitkommazahlen belegen jeweils 4 Byte im Speicher, sodass jedes Merkmal in 4 Byte großen Blöcken im Paket gespeichert wird. Beim Training wird zusätzlich das Label y angehängt. Hier

antwortet das FPGA mit einem 1 Byte großen Antwortpaket, welches einen passenden Kontrollcode enthält.
Im Vorhersagemodus fügt die Gegenstelle kein Label hinzu, sodass das verschickte Paket hier 4 Byte kleiner ist. Das FPGA für dann eine Vorhersage durch und antwortet mit einem 5 Byte großen Paket, welches den Kontrollcode sowie die Vorhersage enthält.

0000 000X	$x_1^{(1)}$	$x_1^{(2)}$	$x_1^{(3)}$	$x_1^{(4)}$	$x_2^{(1)}$	$x_2^{(2)}$	$x_2^{(3)}$	$x_2^{(4)}$	$\cdots$	$y^{(1)}$	$y^{(2)}$	$y^{(3)}$	$y^{(4)}$

Abbildung 6.3: Umsetzung eines Beispiels $(\vec{x}, y)$ als TCP Paket. Jedes Kästchen repräsentiert einen Byte. Das niederwertigste Bit des ersten Bytes enthält eine 0 für Training und eine 1 für den Vorhersagemodus. Nachfolgenden kommen jeweils 4 Byte-Blöcke der einfachen Gleitkommazahlenrepräsentation der Merkmale $x_1, \ldots, x_d$. Im Falle des Trainingsmodus wird das Label ebenfalls als 4 Byte-Block angehängt.

Lightweight IP

Lightweight IP (`lwIP`) ist eine minimale TCP/IP Stack Implementierung für eingebettete Systeme, die sich vor allem durch ihre starke Konfigurierbarkeit auszeichnet. Für die vorliegende Arbeit sollte der `lwIP` Stack möglichst speichersparend konfiguriert sein, damit der Speicher des Soft-Prozessors verringert werden kann, um die Ressourcen des FPGAs zu schonen.
Die einfachste Möglichkeit zur Kommunikation ist eine direkte Punkt-zu-Punkt Verbindung, in der das FPGA und die Gegenstelle alleine und ohne weitere Netzwerkteilnehmer miteinander kommunizieren. Dieses Vorgehen hat zunächst den Vorteil, dass der gesamte Netzverkehr kontrolliert werden kann, da beide Teilnehmer bekannt sind. Durch die volle Kontrolle über das Netzwerk kann dann `lwIP` entsprechend konfiguriert werden. Ein solches Vorgehen ist jedoch in vielen Fällen unpraktikabel, da hier ein eigenes Netzwerk für das FPGA und die Gegenstelle aufgebaut werden muss. Soll die Gegenstelle oder das FPGA auf Daten außerhalb dieses Netzwerkes zugreifen, wird eine weitere Ethernetschnittstelle am FPGA oder der Gegenstelle benötigt. Bei der Konfiguration von `lwIP` in einer bestehenden Netzwerkumgebung muss darauf geachtet werden, dass andere Netzwerkteilnehmer ebenfalls Datenpakete versenden. Hier besteht insbesondere keine Kontrolle über ARP und UDP-Broadcastnachrichten anderer Teilnehmer, sodass die Größe von Sende- und Empfangsbuffern entsprechenden gewählt werden müssen.
Die hier vorliegenden `lwIP` Konfiguration wurde innerhalb des Lehrstuhlnetzwerkes des Lehrstuhls 8 für künstliche Intelligenz an der TU Dortmund getestet und lieferte hier eine stabile Netzwerkkommunikation. Tabellen 6.2 und 6.3 zeigen die entsprechende Konfiguration mit ihrem Standardwert und einer kurzen Erläuterung, wobei hier lediglich geänderte Parameter aufgeführt sind. Die Parameternamen orientieren sich an denen im

Board-Support Package von Xilinx verwendeten Namen[2]. An dieser Stelle sei auf einen kleinen Fehler in der mitgelieferten lwIP Version 1.4.1 im Board-Support Package von Xilinx hingewiesen. Xilinx bietet in Anlehnung an die C Funktion printf eine eigene Implementierung xil_printf zur Ausgabe von Zeichenketten über die serielle Schnittstelle an. Diese Implementierung enthält weitaus weniger Formatierungsoptionen, sodass die resultierende Instruktionsgröße deutlich kleiner ist. Die lwIP Variante im Board-Support Package respektiert diese Funktion nicht durchgehend, sodass sowohl printf, als auch xil_printf in der Implementierung benutzt werden. Tauscht man manuell alle Aufrufe von printf mit xil_printf aus, so reduziert sich die gesamte Codegröße um ca. 80 KB (vgl. Tabelle 6.6).

AXI Connect

Um die einzelnen Komponenten wie MicroBlaze Prozessor oder Ethernet miteinander zu verbinden, kommt das Advanced eXtensible Interface (AXI) Protokoll zum Einsatz [Xild]. Ein Großteil der von Xilinx implementierten Intellectual Property wie z.B. der MicroBlaze Prozessor unterstützt dieses Interface. Somit ist es sehr einfach, Komponenten von Xilinx miteinander zu verbinden, insofern die AXI Interfaces der Komponenten gleich konfiguriert sind.
AXI ist zum Zeitpunkt dieser Arbeit in Version 4 verfügbar und bietet drei Modi: AXI4, AXI4-Lite und AXI4-Stream. AXI4 ist dabei für hohe Geschwindigkeiten, AXI4-Lite für einfache, leichte Kommunikation und AXI4-Stream für einen hohen Durchsatz aufeinanderfolgender Daten ausgelegt. Alle drei Modi benutzen ein einfaches Master/Slave System mit einer Daten- und einer Kontrollleitung. Die Kontrollleitung gibt die Adressen der zu lesenden Bits an, wohingegen die eigentlichen Übertragung über die Datenleitungen erfolgt. Da im Wesentlichen Kontrollinformationen und einzelne TCP/IP Datenpakete über AXI zwischen den Komponenten kommuniziert werden müssen, wird auch hier im Sinne der Einfachheit AXI4-Lite benutzt.
Der MicroBlaze Prozessor steuert die übrigen Hardwaremodule, sodass er als Master-Device eingesetzt wird. Zusätzlich benötigt das Ethernetmodul Zugriff auf gemeinsamen Speicher, um dort die empfangenen Ethernetpakete abzulegen. Daher wird es ebenfalls als Master-Device betrieben. Zur Verbindung der einzelnen Module wird das AXI4 Interconnection Modul verwendet, welches den Zugriff der beiden Master-Devices auf die übrigen Module koordiniert.
Des Weiteren kommt Intellectual Property zur Unterbrechungssteuerung für die angeschlossene Peripherie und ein Modul zur Resetsteuerung des Boards zum Einsatz. Diese IPs lassen sich nur eingeschränkt konfigurieren und sorgen im Wesentlichen für die kor-

[2]Konfiguriert man lwIP außerhalb dieses Packages, so werden die Parameternamen vollständig groß geschrieben, d.h. aus mem_size wird MEM_SIZE etc.

Name	Konfiguration	Standardwert	Erläuterung
`arp_table_size`	2	10	Das ARP Protokoll übersetzt durch spezielle Anfragen im Netzwerk IP-Adressen in eine passende MAC-Adresse für das physikalische Übertragungsmedium. Im vorliegenden Anwendungsfall wird lediglich mit einer einzigen Gegenstelle kommuniziert, sodass hier ein Übersetzungseintrag genügt. Der Sicherheit halber wird Platz für einen weiteren Eintrag reserviert.
`ip_reass_max_pbufs`	32	256	Das IP Protokoll schreibt eine Fragmentierung, d.h. Aufteilung von zu großen IP Paketen in einzelne Teilpakete vor. Diese Aufteilung kann beim Sender direkt oder bei einer Vermittlungsstation wie z.B: einem Router geschehen und ist geräteabhängig. Es wird ein ein Sicherheitsbuffer von 32 Elementen vorgesehen.
`mem_size`	1000 Byte	131072 Byte	`lwIP` implementiert ein eigenes Speichermanagment, womit dynamische Speicherallokationen innerhalb des TCP/IP Stacks unter der Kontrolle des Stacks selber liegt. Der hier gewählte Wert von 1 KB ist extrem knapp bemessen, liefert aber eine stabile Netzwerkkommunikation für den vorliegenden Anwendungsfall.
`memp_n_pbuf`	4	16	Anzahl der internen Buffer die `lwIP` zum Senden von Daten aus statischem Speicher wie `ROM` zur Verfügung hat. Die vorliegende Anwendung sendet und empfängt abwechselnd ein TCP Paket, sodass hier nie mehr als ein Paket zum Senden bereit steht. Zur Sicherheit sind hier insgesamt 4 Buffer vorgehalten.
`memp_n_tcp_pcb`	2	32	Anzahl der aktiven Buffer für eine TCP Verbindung. Die vorliegenden Anwendung sieht lediglich eine TCP Verbindung vor. Da die vorliegende Konfiguration möglichst Netzwerkunabhängig sein soll, wird zu Sicherheit ein zweiter Buffer für mögliche andere Verbindungen konfiguriert.
`memp_n_tcp_pcb_listen`	2	8	Anzahl der gleichzeitig offenen TCP Verbindungen. Dieser Wert wird analog zu `memp_n_tcp_pcb` gewählt.

Tabelle 6.2: Konfiguration von `lwIP`.

Name	Konfiguration	Standardwert	Erläuterung
`memp_n_tcp_pcb_seg`	32	256	Anzahl der gleichzeitig gespeicherten TCP Segmente bei der Fragmentierung von TCP Paketen. Dieser Wert ist analog zu `ip_reass_max_pbufs` gewählt.
`pbuf_pool_size`	10	256	Der Datentyp `pbuf` ist in `lwIP` der zentralen Datentyp für jegliche Datenpakete des physikalischen Mediums, d.h. sowohl TCP, UDP als auch ARP Pakete werden zunächst in diesem Buffer gespeichert. 10 Einträge führten für diese konkrete Anwendung zu einem minimalen Speicherverbrauch, bei gleichzeitiger stabiler Netzwerkkomunikation.
`tcp_snd_buf`	2920 Byte	8192 Byte	Der Sendebuffer für TCP Nutzlasten. Ein TCP Paket kann bis zu 1460 Bytes Nutzlast ohne Fragmentierung transportieren. Damit können in diesem Buffer die Nutzlast von bis zu zwei vollen TCP Paketen gespeichert werden.
`DHCP`	`off`	`on`	Aktiviert die automatische Vergabe von IP Adressen auf Basis der MAC-Adresse unter Verwendung des DHCP Protokolls. Wird die IP Adresse statisch in den Quellcode eingebunden, so muss keine automatische Vergabe per DHCP erfolgen. Damit ist DHCP unnötig und kann abgeschaltet werden.
`UDP`	`off`	`on`	Für den vorliegenden Anwendungsfall wird lediglich TCP benötigt. Durch das Abschalten von DHCP kann damit die Unterstützung von UDP ebenfalls komplett abgeschaltet werden, sodass UDP Pakete vollständig ignoriert werden.
`phy_link_speed`	`1000`	`AUTODETECT`	Das verwendete Ethernetmodul wird mit einer Geschwindigkeit von 1 Gbps konfiguriert, sodass hier keine automatische Konfiguration notwendig ist.

Tabelle 6.3: Konfiguration von `lwIP` (Fortsetzung).

rekte Kommunikation der Module untereinander, weshalb für eine detaillierte Auflistung auf den Schaltplan im Anhang B.1 verwiesen ist.
Der `MicroBlaze` Prozessor, sowie die übrigen Module müssen mit einem Takt versehen werden. Üblicherweise werden alle Module mit dem gleichen Takt betrieben, sodass Resets und Datenübertragungen synchron verlaufen. Um jedoch eine lose Kopplung zwischen dem maschinellen Lernalgorithmus und dem restlichen System zu erlauben, werden zwei verschiedene Taktgeber im System verwendet. Der erste Taktgeber betreibt den `MicroBlaze` Prozessor und die angeschlossene Peripherie mit 50 MHz, wohingegen der zweite Taktgeber nur den maschinellen Lernalgorithmus betreibt. Die möglichen Taktraten für Projection GP werden ausführlich in Abschnitt 6.3.1 diskutiert.
Das Ethernet Subsystem wird mit einem Takt von 125 MHz für eine Übertragungsgeschwindigkeit von 1 Gbps betrieben, wobei das Ethernetmodul zusätzlich einen Referenztakt von 200 MHz benötigt [Xilc].

IO

Wie bereits angesprochen, wird Ethernet als zentrales Kommunikationsmittel verwendet. Wie in Tabelle 6.1 gezeigt, definiert Ethernet sowohl Eigenschaften der Bitübertragungsschicht, als auch Eigenschaften der Sicherungsschicht. Um eine entsprechende Datenkommunikation zwischen beiden Schichten zu implementieren, wurde das Media-independent Interface (MII) standardisiert. Zusätzlich zum eigentlichen Datenaustausch können über das Management Data Input/Output (MDIO) Interface Kontrollinformationen zwischen den beiden Schichten ausgetauscht werden (siehe `IEEE` Standard `802.3`, [IEE]).
Der Ethernetstandard beschreibt eine Menge verschiedener Ethernervarianten mit unterschiedlichen Übertragungsgeschwindigkeiten und Steckerformaten. Je nach konkreter Variante, beschreibt die MII und die MDIO Spezifikation leicht andere Register, Taktraten und Kontrollsignale. Das bedeutet konkret, dass MII und MDIO für den gewünschten Ethernet-Betriebsmodus konfiguriert werden müssen.
`Xilinx` bietet eine Vielzahl verschiedener Ethernet Implementierungen für verschiedene Geschwindigkeiten und Boards an [Xila, Xilc, Xilf]. Diese unterscheiden sich jeweils in ihren Eigenschaften und ihrer Benutzung. In der vorliegenden Arbeit wurde das `AXI 1G/2.5G Ethernet Subsystem` verwendet, da es sowohl eine einfache `AXI` Schnittstelle anbietet, als auch eine Abstraktion der Bitübertragungs- und Sicherungsschicht in einem Modul bietet. Es unterstützt Geschwindigkeiten von 10, 100 und 1000 Mbps und bietet Zugriff auf verschiedene MII Protokollvarianten.
Tabelle 6.4 zeigt die Portbelegungen für das `AXI 1G/2.5G Ethernet Subsystem` auf dem verwendeten `Artix-7 AC701 Evaluation Board`. In Anlehnung an die Verwendung des `AXI4-Lite` Protokolls wurde zur Verbindung zwischen Bitübertragungsschicht und Sicherungsschicht das Reduced media-independent Interface (RMII) gewählt. Diese Protokoll-

variante kommt im Gegensatz zu den übrigen MII Protokollen mit einer minimalen Anzahl an Signalen und Signalefinitionen aus, wodurch seine Verwendung vergleichsweise einfach ist. Durch seine Einfachheit spezifiziert RMII jedoch nicht, ob die Netzwerkkarte im Half-Duplex oder Full-Duplex Modus betrieben werden soll und macht auch keine Angaben über die Übertragungsgeschwindigkeit (vgl. [IEE]). Für einen korrekten Betrieb müssen daher beide Schichten im gleichen Betriebsmodus konfiguriert sein. Zusätzlich muss dieser Betriebsmodus von der Gegenstellen im Netzwerk akzeptiert werden, weshalb bei einer Benutzung von RMII darauf zu achten ist, dass die Betriebsmodi entsprechend per Hand konfiguriert werden.
Für das MDIO Protokoll ist die Liste der Wahlmöglichkeiten kurz, da hier nur das MDIO MDC Protokoll existiert. Zusätzlich bietet `Xilinx` die Implementierung eines eigenen Protokolls an, worauf an dieser Stelle verzichtet wurde.
`DIFFCLK` und `PYHRST_N` beschreiben die Ports für den Takt und den Reset auf dem FPGA Board und müssen boardspezifisch gewählt werden (siehe [ASC]).
Ethernet teilt die zu übertragenden Daten in sogenannte Etherframes auf. Ethernetframes sind bis zu 1522 Byte groß, wobei hiervon 1500 Byte die Nutzlast ausmachen und lediglich 22 Byte für MAC-Adresse und Kontrollinformationen im Paketkopf verwendet werden. Aus diesem Grund werden Sende- und Empfangsbuffer jeweils mit 4 KB Größe konfiguriert, sodass diese jeweils bis zu zwei Pakete mit maximaler Nutzlast aufnehmen können. Eine Unterstützung für sogenannten Jumbo-Frames, also Ethernetframes die mehr als 1500 Byte Nutzlast tragen wird abgeschaltet, sodass diese Pakete schlichtweg ignoriert werden. Das `AXI 1G/2.5G Ethernet Subsystem` wird im Full-Duplex Modus mit 1 Gbps Übertragungsgeschwindigkeit betrieben, was entsprechend in der `lwIP` Konfiguration eingetragen ist (siehe Tabelle 6.3).
Zur Ausgabe von Entwicklerinformationen kommt zusätzlich eine Serielle Schnittstelle zur Anwendung. Der Einfachheit halber wird hier das `AXI UARTLITE` Modul verwendet und das `RS232 Uart` Protokoll mit 8 Bit und einer Baudrate von 9600 konfiguriert.

IP Interface	Board Interface
ETHERNET	RGMII
MDIO	MDIO MDC
DIFFCLK	Custom
PHYRST_N	phy reset out

Tabelle 6.4: Portkonfiguration des `AXI 1G/2.5G Ethernet Subsystem`.

Speicher

Der `MicroBlaze` Prozessor benötigt Speicher für Instruktionen und Daten. Zusätzlich müssen der Soft-Prozessor und das `AXI 1G/2.5G Ethernet Subsystem` TCP-Pakete über

einen gemeinsamen Speicher austauschen.
Hier ist die Verwendung von chipinternem Blockram oder die Benutzung des DDR Speichers des Entwicklerboards möglich. Die nachfolgende Diskussion zeigt jedoch in Tabelle 6.5, dass DDR Speicher einen vergleichsweise hohen Energieverbrauch hat, sodass auf die Verwendung von Blockram zurückgegriffen wird.
Als gemeinsamer Speicher von Prozessor und Ethernetsystem wird eine einfache Sende- und Empfangswarteschlange verwendet. Den Buffergrößen innerhalb des Ethernetmoduls folgenden, werden diese Warteschlangen auch jeweils mit je 4 KB Speicher ausgestattet.
Die Größe von Daten- und Instruktionsspeicher für den Prozessor richtet sich nach dem auszuführenden Programm, d.h. im Wesentlichen nach der Größe des TCP/IP Stacks. Vorgreifend zeigt Tabelle 6.6 eine Übersicht der resultierenden Programmgröße bei einer Minimalkonfiguration von lwIP, wobei hier knapp 220 KB Daten- und Instruktionsspeicher benötigt werden.
An dieser Stelle sei darauf hingewiesen, dass der MicroBlaze Prozessor über einen Prefetching Mechanismus verfügt. Dieser sorgt dafür, dass während der Ausführung einer Instruktion die nachfolgenden Daten und Instruktionen aus dem Speicher geladen werden. Dies hat den Vorteil, dass die Pipelineverarbeitung (vgl. Abschnitt 6.3.2) ohne Verzögerung durch Zugriff auf Speicher geschehen kann. Damit der Prefetching Mechanismus jedoch nicht auf Speicherinhalte außerhalb des Speichers zugreift, empfiehlt Xilinx den Speicher nicht vollständig mit Daten und Instruktionen zu belegen, sondern einen kleinen Speicherbereich unbenutzt zu lassen [Xile]. Damit wird der lokale Speicher des MicroBlaze Prozessors auf insgesamt $256 \cdot 1024 = 262144 \approx 260$ KB konfiguriert, was einen Spielraum von Rund 40 KB für weiteren Programmcode und den Prefetching Mechanismus ermöglicht.

6.2.3 Diskussion

Im vorangegangenen Abschnitt wurde die Konfiguration eines FPGA Systems vorgestellt, welches durch die Verwendung von AXI unabhängig vom eigentlichen Lernalgorithmus ist. Nach Wissen des Autors ist dieser Ansatz neu, da die in der Literatur zu findenden FPGA Systeme üblicherweise das Gesamtsystem mit Lernalgorithmus implementieren. Das hier vorgestellte System kann jedoch unabhängig vom Lernalgorithmus benutzt werden, sodass dieser modular ausgetauscht werden kann. Diese Modularität konnte während der Entwicklung ausgenutzt werden, um die langen Synthesezeiten von Projection GP zu umgehen (siehe Tabelle 6.9). Hierzu wurde zunächst eine einfaches Perzeptron (vgl. [HTF01], Abschnitt 4.5.1) mit Hilfe der High Level Synthese implementiert und als Stellvertreter für Projection GP genutzt. Anschließend konnte die bereits vorgestellte Systemarchitektur entwickelt werden, indem verschiedene Konfigurationen des Gesamtsystems und lwIP ausprobiert und verglichen werden konnten.

Der Fokus auf eine möglichst platz- und energiesparendes Gesamtsystem scheint naheliegend, muss jedoch auf Basis der Anwendung entsprechend evaluiert werden. Das von `Xilinx` unter [ASC] angeboten Beispiel zur Implementierung einer Ethernetverbindung auf einem FPGA sieht für eine hohen Datendurchsatz die Benutzung von chipexternem `DDR` Speicher mittels direct memory access (DMA) vor.
Der mit Projection GP gewählte Algorithmus hingegen hat einen konstanten Speicherbedarf, sodass hier alleine schon durch die Wahl des Algorithmus die Benutzung von chipexternem `DDR` Speicher fragwürdig ist. Zusätzlich hat das implementierte Kommunikationsprotokoll einen abschätzbaren Speicheraufwand, sodass durch eine geschickte Konfiguration von `lwIP` der `DDR` Speicher nicht notwendig ist.
Tabelle 6.5 zeigt die Unterschiede zu dem hier vorgestellten System und einer Konfiguration, die sich an dem von `Xilinx` unter [ASC] vorgestellten Beispiel mit `DDR RAM` anlehnt. Es ist jeweils die Anzahl der verbrauchen Logikzellen (LUT), die Anzahl der verbrauchten Flipflops (FF), die Menge des verbrauchten Blockrams (BRAM), sowie die Menge der verbrauchten der DSP-Elemente angegeben. Der Energieverbrauch dieser Systeme hängt neben den verwendeten FPGA Ressourcen von der benutzen Peripherie,sowie der Last des Soft-Prozessors ab. Führt dieser berechnungsintensive Programme aus, so steigt der Energieverbrauch des Systems. Aus diesem Grund zeigen letzten beiden Spalten den durch das Synthesetool `Vivado` geschätzten durchschnittlichen bzw. maximalen Energieverbrauch.
Im Falle des `DDR` Speichers werden knapp 20 % der Logikzellen und 10 % der Flipflops verbraucht. Blockram wird hier lediglich zu 7 % benötigt, wohingegen 2 % der DSP-Elemente ausgenutzt werden. Insgesamt benötigt diese Konfiguration $2,1 - 2,2$ Watt Energie.
Dem Gegenüber benötigt eine Konfiguration ohne `DDR` Speicher lediglich knapp 5 % der Logikzellen und knapp 3 % an Flipflops, wodurch diese Konfiguration in beiden Fällen knapp um einen Faktor von Vier ressourcenschonender ist.
Um das Fehlen des `DDR` Speichers zu kompensieren benötigt diese Konfiguration mit knapp 21 % des Blockrams, was ca. drei mal mehr als eine Konfiguration mit `DDR` Speicher ist. DSP-Elemente werden in dieser Konfiguration gar nicht benötigt. Insgesamt benötigt ein System ohne `DDR` Speicher lediglich $0,47 - 0,56$ Watt Energie, wodurch es auch hier einen Faktor von ca. Vier ressourcenschonender ist.
Es fällt auf, dass die Anzahl der Logikzellen bei Benutzung des `DDR` Speichers ca. 15 Prozentpunkte höher ist als eine Konfiguration ohne `DDR` Speicher. Der Grund hierfür liegt an der zusätzlichen Zugriffslogik, die benötigt wird um auf den externen Speicher zuzugreifen und die notwendige Wiederauffrischung der Speicherinhalte im flüchtigen `DDR` Speicher durchzuführen, die Teilweise innerhalb der Zugriffslogik konfiguriert werden kann (vgl. [Xilm], Kapitel 1). Des Weiteren erfolgt ein Zugriff auf den `DDR` Speicher über das DMA Protokoll, welches zusätzliche Logik benötigt.
Des Weiteren verbraucht der `DDR` Speicher selbst ca. 1.5W Energie, wodurch sich der gesamte Energieverbrauch deutlich erhöht.

	LUT [%]	FF [%]	BRAM [%]	DSP [%]	∅ Energie [W]	Max Energie [W]
Mit DDR	19,62	9,6	7,67	2,03	2,154	2,268
Ohne DDR	4,85	2,65	21,37	0	0,472	0,566

Tabelle 6.5: FPGA Ausnutzung und Energieverbrauch für das System ohne maschinellen Lernalgorithmus.

Durch das Weglassen des DDR Speichers steht weniger Speicher auf dem Board zur Verfügung, sodass die Benutzung von Blockram eingeschränkt werden muss. Tabelle 6.6 zeigt den Speicherbedarf des Lightweight IP Protokollstapels in seiner Standardkonfiguration und in der hier vorgestellten Konfiguration. Zum Testen der Konfiguration wurde in beiden Fällen ein einfacher Echoserver auf dem FPGA ausgeführt. Dieser nimmt zunächst Datenpakete entgegen, entpackt diese und schickte dann ihre Nutzlast in einem neuen Paket an die Gegenstelle zurück.
Der `.text` Eintrag beschreibt die Größe des Textsegmentes des Programms und umfasst damit die Speichergröße aller Programminstruktionen. Im Falle einer Standardkonfiguration belegt dieses Segment ca. 150 KB Speicher, wohingegen die Minimalkonfiguration hier 100 KB Speicher benötigt. Das `.data` Segment umfasst initialisierte globale bzw. lokale, statische Variablen. Hier kommt die Minimalkonfiguration mit etwa einem halben Kilobyte Speicher aus, wohingegen die Standardkonfiguration fast 2 KB benötigt. Das `.bss` Segment speichert alle nicht initialisierten Variablen und ist mit Abstand das größte Segment. Im Falle der Standardkonfiguration benötigt dieses Segment fast 750 KB, wohingegen die Minimalkonfiguration hier lediglich mit 115 KB auskommt. Zusammenfassend wurde so die Gesamtgröße des TCP/IP Protokollstapels um einen Faktor von ca. Vier verringert.

Konfiguration	`.text` [Byte]	`.data` [Byte]	`.bss` [Byte]	Gesamtgröße [Byte]
Standardkonfiguration	153480	1816	749080	904376
Minimalkonfiguration	105637	440	115336	221413

Tabelle 6.6: Vergleich der Programmgrößen des TCP/IP Protokollstapels unter Minimalkonfiguration und Standardkonfiguration. Die Funktionalität wurde mit einem einfachen Echoserver getestet, der alle empfangenen Daten über TCP an die Gegenstelle zurücksendet.

An dieser Stelle zeigt sich, dass der gewählte Algorithmus durchaus starke Auswirkungen auf die eigentliche Implementierung und die Systeminstanz haben kann (vgl. Abschnitt 2). Sowohl das Perzeptron, als auch Projection GP sind Online Algorithmen, die jeweils eine Beobachtung nach der anderen konsumieren. Das Kommunikationsprotokoll kann diesem Umstand Rechnung tragen, indem nur dann Daten versendet werden, wenn der Algorithmus bereit ist, sie zu konsumieren.
Damit konnte `lwIP` für genau diesen Anwendungszweck konfiguriert werden, wodurch der

chipexterne DDR Speicher obsolet wurde, was wiederum zu einer Ersparnis von knapp $1,5\text{W}$ führt.
Das vorgestellte Ping-Pong Protokoll zur Kommunikation bildet hierbei sicherlich einen den Extremfall, da immer nur genau eine Beobachtung versendet wird. Um die Effizienz des Systems zu steigern, ist es möglich insgesamt P Pakete zusammen zu versenden, zwischenzuspeichern und anschließend mit nur einem Antwortpaket zu bestätigen. Auf diese Art und Wiese wird der Kommunikationsmehraufwand minimiert und die Paketumlaufzeit spielt eine geringere Rolle bei der Performanz.
Die hier gezeigte Konfiguration behält eine Kompatibilität zum Ethernet- und TCP-Standard bei, indem Buffer auf Hinblick der maximalen Paketgrößen konfiguriert wurden, sodass sich so eine Verwendbarkeit in Netzwerken mit mehreren Teilnehmern ergibt.
Kann man jedoch sicherstellen, dass sich die Netzwerkteilnehmer an gewissen Konventionen halten[3], so bietet sich hier noch weiterer Spielraum für Optimierungen. Der Ethernetstandard sieht eine maximale Nutzlast von 1500 Byte vor. Damit kann ein einziges TCP Paket 1460 Bytes Nutzlast tragen kann, was wiederum 365 einfachen Gleitkommazahlen entspricht. Ist durch die Anwendung und den Algorithmus jedoch bekannt, dass lediglich z.B. 21 Merkmale verwendet werden (siehe hierzu Kapitel 7), so kann mit Label und Konfigurationsbyte die maximale Paketgröße zu $21 \cdot 4 + 4 + 1 = 89$ Byte abgeschätzt werden. Damit kann dann die maximale TCP-Nutzlast in `lwIP` zu 89 Byte konfiguriert werden, was nur noch 6% der ursprünglichen Größe entspricht. Analog kann dann die maximale Nutzlast der Ethernetframes unter Berücksichtigung des TCP/IP Headers auf 129 Byte verringert werden, was nochmal eine deutliche Verringerung des Speicherbedarfs von `lwIP` bedeuten würde, sodass die Verwendung von weiter Blockram minimiert wird.

6.3 FPGA Implementierung

Im vorangegangenen Abschnitt wurde ein FPGA System mit Netzwerkkommunikation beschrieben, welches den Hardwareblock eines maschinellen Lernalgorithmus passend ausführen kann. In diesem Abschnitt soll nun auf Basis der Hochsprachenimplementierung aus Abschnitt 6.1 der entsprechende Hardwareblock für Projection GP entwickelt werden. Dazu wird die eigentliche Implementierung in Abschnitt 6.3.1 gezeigt, wohingegen Abschnitt 6.3.2 auf Optimierungen dieser Implementierung eingeht. Anschließend wird dieser Abschnitt und das Kapitel mit einer Diskussion beendet.

6.3.1 Projection GP Implementierung

vorgestellten Hochsprachenimplementierung wurde zunächst in `C` implementiert, da das High Level Synthese Werkzeuge `Vivado High Level Synthesis` von `Xilinx` diesen auto-

[3] Dies wäre zum Beispiel in einer direkten Punkt-zu-Punkt-Verbindung ohne weitere Netzteilnehmer möglich.

matisiert in passenden Hardwarecode umwandeln kann. Damit kann die Funktionalität des Algorithmus zunächst in `C` entwickelt und getestet werden und anschließend kann die so entstandene Implementierung auf die besonderen Eigenschaften der Hardware angepasst und optimiert werden.
Zusätzlich verkürzt sich die Entwicklungszeit, da vergleichsweise einfache, aber langwierige Programmieraufgaben wie Pipeline-Implementierung oder Loop-Unrolling automatisiert durchgeführt werden können [Xilh]. Zu guter Letzt kann `Vivado High Level Synthesis` besondere Eigenschaften wie Blockram oder DSP Elemente des FPGA Chips direkt ausnutzen, sodass an dieser Stelle Logikzellen eingespart werden könne. Damit erzeugt das HLS Tool in den meisten Fällen vergleichbare Ergebnisse zu einer manuellen Implementierung bei weitaus kürzeren Entwicklungszeiten [CLN+11, NLB+13].
Bei der Erzeugung von effizientem Hardwarecode muss jedoch weiterhin die unterliegende Struktur des FPGAs berücksichtigt werden:

- Jegliche verwendete Variablen werden entweder in Flipflops, die aus den Logikzellen selbst generiert werden, gespeichert oder mittels Blockram auf dem FPGA realisiert. Das HLS Werkzeug versucht eine passende Umsetzung basierend auf dem Quellcode zu wählen, wobei globale und statische Variable in Blockram und lokale Variable in Flipflops implementiert werden [Vivb].
  Ein Zugriff auf den `DDR` Speicher muss bei Bedarf manuell implementiert werden.

- Das Speicherlayout des FPGAs kennt keinerlei Speicheradressen, sodass Adressmanipulation und insbesondere Zeiger kein Äquivalent auf dem FPGA haben. Generell bietet das FPGA keinerlei Unterstützung für dynamische Speicherallokation.[4]

- FPGAs sind im Gegensatz zu üblichen integrierten Schaltungen um eine Größenordnung geringer getaktet [BRS13]. Das bedeutet, dass sie pro Sekunde Rund 1000 Operationen weniger durchführen können als handelsübliche CPUs oder GPGPUs. Um dennoch eine ausreichende Geschwindigkeit zu erreichen, sollten Berechnungen nach Möglichkeit parallelisiert werden und jeglicher Verwaltungsmehraufwand auf ein Minimum reduziert werden.

Die Logikzellen des FPGAs haben eine gewissen Signalverzögerung, die sich aus ihren physikalischen Eigenschaften ergibt, sodass die einzelnen Bauteile wie Addierer oder Multiplizierer auch einer Signalverzögerung δ unterliegen. Das bedeutet, dass ein stabiles Ausgangssignal erst nach einer Verzögerung δ nach dem Anliegen eines Eingangssignals generiert wird.
Neben der bauteilbedingten Verzögerung beeinflussen auch die Signallaufzeiten die gesamte Signalverzögerung. Werden zwei miteinander verbundene Bauteile in zwei verschiedenen

[4] `Vivado High Level Synthesis` versucht die benötigten Speichergrößen bei der Verwendung von `new` oder `malloc` aus dem übrigen Code abzuleiten, um dennoch sinnvoll den Blockram nutzen zu können. Ist dies nicht möglich, so lässt sich die verwendete Implementierung nicht in Hardwarecode umwandeln.

Regionen des FPGAs implementiert, so muss das Signal zusätzlich eine räumlich größere Distanz zurücklegen, was die Signalverzögerung wiederum erhöht. Die Auswahl der passenden Regionen erfolgt im Place & Route Schritt während der Synthese und hängt vom übrigen System ab, sodass die tatsächliche Signalverzögerung erst am Ende des Place & Route Schrittes angegeben werden kann.
Ziel bei der Implementierung von Projection GP ist die optimale Ausnutzung der FPGA Hardware. Hierzu sei zunächst angemerkt, dass das HLS Tool exakt die Anzahl der benötigten Taktzyklen c für die Bearbeitung eines Beispieles angibt. Des Weiteren gibt es eine Schätzung über die maximale Signallaufzeit Δ und die Anzahl der insgesamt für Speicher und Logik benötigten Logikzellen u an.
Der Kehrwert der maximalen Signallaufzeit gibt die maximale Taktfrequenz $f = \Delta^{-1}$ an. Üblicherweise wird die Signallaufzeit in Nanosekunden angegeben, wohingegen für die Angaben der Taktfrequenz Megaherz benutzt wird, sodass sich nach SI-Einheitenpräfixe folgende Umrechnung ergibt:

$$f = \frac{1}{\Delta \cdot ns} = \frac{1}{\Delta \cdot 10^{-9} s} = \frac{10^9}{\Delta s} = \frac{1000}{\Delta} \cdot 10^6 \cdot \frac{1}{s} = \frac{1000}{\Delta}\text{MHz}$$

Um den Designprozess transparent zu halten, lässt sich nun folgendes Optimierproblem formalisieren. Bezeichne hierzu s die Gesamtanzahl der Logikzellen des FPGAs und $T = \frac{f}{c}$ den theoretisch maximalen Durchsatz, so ist diejenige Implementierung gesucht, die T maximiert:

$$\begin{gathered} \max T \\ \text{sodass } u \leq s \end{gathered}$$

Speicherlayout

Abbildung 6.4 zeigt das Speicherlayout von Projection GP. Die Implementierung orientiert sich an denen in Abschnitt 6.1 diskutierten Überlegungen zum Speicherlayout. Die Dimension der Beispiele und die Anzahl der Basisvektoren wird statisch zur Kompilierzeit angegeben, sodass die Größen der verwendeten Felder ebenfalls zur Kompilierzeit bekannt sind. Durch die Benutzung eindimensionaler Felder kann das HLS Werkzeug die Felder direkt als Blockram implementieren, sodass keine Logikzellen verwendet werden (vgl. [Xilh]). Die Initialisierung mit Nullen erfolgt bei der gewählten Syntax standardmäßig beim Einschalten des FPGAs.

Algorithmenlayout

Die Implementierung des Algorithmus ergibt sich auf natürliche Art und Wiese aus den Formulierungen in Abschnitt 6.1, weshalb an dieser Stelle nur exemplarisch einige Teile diskutiert werden.

```
#define numBV 100
#define bvMax (numBV+1)
#define dim 21

float C[bvMax * bvMax] = {0};
float Q[bvMax * bvMax] = {0};
float e[bvMax] = {0};
float k[numBV] = {0};
float s[bvMax] = {0};
float alpha[bvMax] = {0};
float basisVectors[bvMax*dim] = {0};
unsigned int bvCnt = 0;
```

Abbildung 6.4: Speicherdefinition von Projection GP.

Wesentlicher Bestandteil von Projection GP ist die Berechnung der Kernfunktion. Abbildung 6.5 zeigt die Implementierung eines RBF-Kernels (siehe Abschnitt 4.4) mit Skalierungsparametern γ und l, welche ebenfalls statisch zur Kompilierzeit festgelegt werden. Des Weiteren wird die Dimensionalität der Beispiele `pX1` und `pX2` explizit in der Funkti-

```
#define gamma_kernel 0.5
#define len 1

float K(float const pX1[dim], float const pX2[dim]) {
    float sum = 0.0;
    for (unsigned int i = 0; i < dim; ++i) {
        sum += (pX1[i] - pX2[i])*(pX1[i] - pX2[i]);
    }

    return len*std::exp((float) - gamma_kernel * sum);
}
```

Abbildung 6.5: Kernfunktionsberechnung von Projection GP.

onsdeklaration angegeben, was in `C`-Code Implementierung zunächst unüblich ist, da die Felder in entsprechende Zeiger übersetzt werden (siehe [C9905]). An dieser Stelle ermöglicht die explizite Größenangabe dem HLS Tool jedoch diese bei der Generierung des HDL Codes zu berücksichtigen, sodass die Ein- und Ausgabeports des generierten Hardwaresubmoduls die entsprechenden Feldgrößen haben.
Zusätzlich vereinfacht die Verwendung des `const` Schlüsselwortes hier die Generierung der Schaltung, da so rein lesender Zugriff auf die Variablen sichergestellt ist, sodass das HLS Tool keine Kontrollleitung zum Schreiben erzeugen muss.
Abbildung 6.6 zeigt die Portdefinitionen des Projection GP Hardwaremoduls. Mittels `#pragma` Anweisung wird `Vivado HLS` dazu angewiesen, entsprechende Ein- und Ausgabe-

ports für das Hardwaremodul zu generieren. Hier sei angemerkt, dass `Vivado HLS` keine Präprozessormacros innerhalb dieser `#pragma` Anweisungen unterstützt, sodass zunächst entsprechende Präprozessormacros definiert werden müssen (vgl. [Xilh]).
Um mit den übrigen Komponenten reibungslos zusammen arbeiten zu können, wird für das Projection GP Hardwaremodul ein `AXI4-Lite` Interface generiert und die Wortbreite entsprechend der Dimension der Beispiele festgelegt. Des Weiteren gibt es einen Kontrollport `pPredict`, der bestimmt ob das Hardwaremodul im Vohersage- oder im Trainingsmodus betrieben wird.
Zur Vorhersage wird eine einfache Schleife verwendet, die mit dem Label `PREDCITION` versehen ist. Dieses Label wird nicht als bedingte Sprunganweisung verwendet, sondern dient der Übersicht im `Vivado HLS` Tool, damit eine Zuordnung der Logikzellen zu den passenden Codeteilen über die Textlabels erfolgen kann.

```
1  #define PRAGMA_SUB(x) _Pragma (#x)
2  #define DO_PRAGMA(x) PRAGMA_SUB(x)
3
4  float projection_gp(float const pX[dim], float const pY, bool const pPredict) {
5  DO_PRAGMA(HLS INTERFACE s_axilite port=pX depth=dim);
6  #pragma HLS INTERFACE s_axilite port=pY
7  #pragma HLS INTERFACE s_axilite port=pPredict
8  #pragma HLS INTERFACE s_axilite port=return
9
10         if (!pPredict) {
11                 train_bv_set(pX,pY);
12                 return 0;
13         } else {
14                 float sum = mean;
15                 PREDICTION:for (unsigned int i = 0; i < numBV; ++i) {
16                         sum += K(&basisVectors[i*dim],pX)*alpha[i];
17                 }
18
19                 return sum;
20         }
21 }
```

Abbildung 6.6: Aufruf von Projection GP.

6.3.2 Optimierung des Durchsatzes

Wie bereits angedeutet, muss die Implementierung auf die Hardware des FPGAs angepasst werden, damit der Durchsatz T maximiert werden kann. An dieser Stelle lässt sich kein allgemeines Vorgehen angeben, da der Durchsatz von vielen Faktoren wie z.B. dem konkreten FPGA Chip oder der übrigen Systemarchitektur abhängt.
Dennoch gibt es eine Reihe von allgemeinen Optimierungen, die prinzipiell für alle FPGAs

in Frage kommen, sodass diese im Folgenden kurz erläutert und diskutiert werden. Um die Diskussion weniger abstrakt zu gestalten, werden die Auswirkungen der Optimierungen anhand der Schleife in der Kernfunktionsimplementierung aus Abbildung 6.5 in Zeile $5-8$ diskutiert.
Mit Hilfe der High Level Synthese lassen sich verschiedene Optimierungen durch Einsatz von Präprozessordirektiven und Änderungen im `C`-Code leicht ausprobieren, um so schließlich eine passende Hardwareimplementierung zu finden.
Als Referenz sei zunächst eine einfache, nicht optimierte Hardwareimplementierung dieser Schleife vorgestellt. Hier ist zunächst zu bemerken, dass offensichtlich zwei Subtraktions- und eine Multiplikationseinheit benötigt wird. Des Weiteren wird der in `sum` gespeicherte Wert aufaddiert, was der Funktionalität eines Akkumulators entspricht. Dieser wird üblicherweise durch eine Additionseinheit, einem Speicherregister und einer Kontrolllogik, die den Zugriff auf das Register steuert, implementiert.
Zusätzlich wird eine Kontrolllogik benötigt, die den Schleifenkopf implementiert, d.h. die die Zählervariable i enthält, diese inkrementiert und die entsprechenden Speicherbereiche `pX1[i]` und `pX2[i]` adressiert.
Abbildung 6.7 zeigt diese Umsetzung in Hardware, wobei hier eine einzige Kontrolllogik für den Schleifenkopf und den Akkumulator verwendet wird. Die Kontrolllogik adressiert intern die entsprechenden Speicherbereiche `pX1[i]` und `pX2[i]` und gibt die resultierenden Signale an die Subtraktionseinheiten weiter. Die dort berechneten Differenzen werden an die Multiplikationseinheit weitergeleitet, welche das Ergebnis an die Additionseinheit weitergibt. Die Additionseinheit addiert diesen Wert auf den zuvor in `sum` gespeicherten Wert und speichert ihn anschließend wieder in `sum` ab.
Der Übersicht halber wurde auf die explizite Angabe des Taktsignals für alle Bauteile verzichtet. Sie werden alle synchron mit dem gleichen Takt betrieben.
Die hier gezeigte Implementierung benötigt genau einen Taktzyklus pro Durchlauf des Schleifenrumpfes, was zu einer Gesamtlaufzeit von `dim` Takten für eine `dim`-dimensionale Beobachtung führt.
Die Logikeinheit, sowie Speicher und übrigen Bauteile wie Additions- oder Multiplikationseinheit werden mit Logikzellen des FPGAs implementiert.
In dem gezeigten Beispiel umfasst die längste serielle Verschaltung ohne Hinzuzählen der Logikeinheit 4 Bauteile. Damit muss die maximale Signalverzögerung hier mindestens mit $\Delta \geq 4\delta$ abgeschätzt werden.

Pipelining

Pipelining ist ein Konzept aus dem Bereich der Mikroprozessoren und stellt heute einen wichtigen Grundbaustein moderner Mikroarchitekturen dar. Die Grundidee besteht darin, komplexe Befehle in M kleinere Teilbefehle zu unterteilen und pro Teilbefehl eine eigene

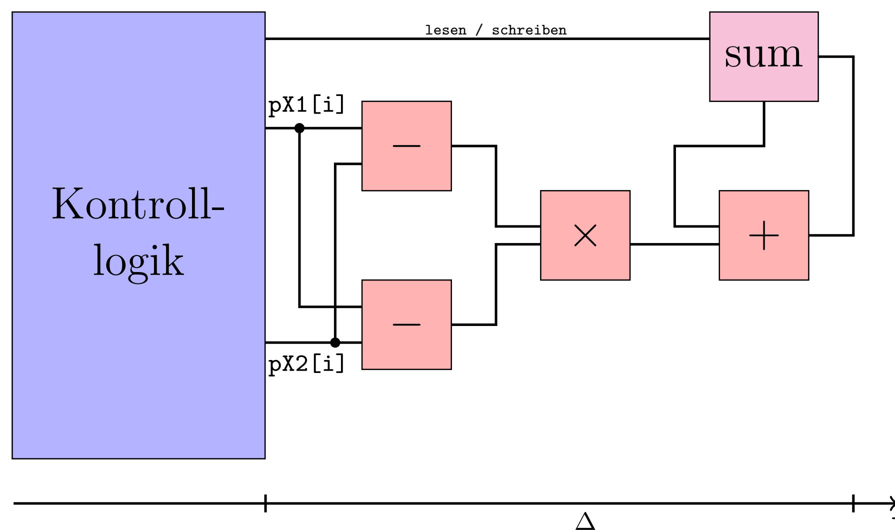


Abbildung 6.7: Schematische Schaltung für die Berechnung der Kernfunktion.

Ausführungseinheit zu verwenden. Für die Abarbeitung eines Gesamtbefehls müssen dann zunächst die M Teilbefehle ausgeführt werden, was insgesamt M Taktzyklen benötigt. Damit erhöht Pipelining die Anzahl der benötigten Taktzyklen um die Tiefe der Pipeline. Dem Gegenüber steht die Tatsache, dass die Teilbefehle deutlich einfacher und damit schneller zu bearbeiten sind, wodurch sich die Taktfrequenz der Ausführungseinheiten erhöhen lässt. Des Weiteren ist zu beobachten, dass ein Gesamtbefehl pro Taktzyklus lediglich eine einzige Ausführungseinheit belegt, sodass die übrigen $M-1$ Ausführungseinheiten frei sind. Diese können durch andere Befehle belegt werden, wodurch eine gewisse Form der Parallelität entsteht.

Abbildung 6.8 zeigt beispielhaft die Schleife der Kernfunktionsberechnung in einer Pipelining Implementierung. Die Kontrolllogik adressiert wie gewohnt die korrekten Speicherbereiche der Felder `pX1` und `pX2` und koordiniert den Zugriff auf die Variable `sum`. Zwischen den Recheneinheiten sind nun Speicherregister zu finden, die den aktuell anliegenden Wert abspeichern und den zuvor gespeicherten Wert an die nachfolgende Recheneinheit ausgeben.

Dies hat zur Folge, dass Pipelining mehr Logikzellen als eine klassische Implementierung benötigt, da zusätzliche Register synthetisiert werden. Des Weiteren muss die Kontrolllogik die Taktverzögerung beim Zugriff auf `sum` berücksichtigen. In dem hier gezeigten Beispiel wird erst mit drei Takten Verzögerung ein Wert für `sum` generiert sein und erst danach wird pro Taktzyklus ein neuer Wert für `sum` berechnet. Ist das Ende der Schleife erreicht, d.h. alle Einträge in den Feldern `pX1` und `pX2` sind abgearbeitet worden, so muss die Kontrolllogik die Pipeline Drei weitere Takte betreiben, um die restlichen Zwischenergebnisse ans Ende der Pipeline zu propagieren. Somit führt Pipelining zusätzlich zu einer

komplexeren Kontrolllogik, die den Anfang und das Ende der Pipeline besonders mitberücksichtigen muss.
Dem Gegenüber steht nun der Umstand, dass Signallaufzeiten deutlich verringert werden. Durch das Einfügen der Register, müssen die Signale nur noch eine Recheneinheit und das nachfolgende Register durchlaufen. Damit halbiert sich die maximale Signalverzögerung Δ im betrachteten Beispiel, wodurch sich die Taktrate erhöhen lässt. Des Weiteren lassen sich nun Teile der Pipeline besser räumlich voneinander trennen, sodass der Aufwand im Place % Route Schritt geringer wird (siehe [Xilh]). Zusätzlich bietet Pipelining die bereits angesprochene Parallelität: Sowohl die Subtraktions-, als auch die Multiplikations- und die Additionseinheiten können bei voller Pipeliningauslastung vollständig parallel arbeiten, sodass alle Recheneinheiten optimal ausgenutzt werden.
Im verwendeten HLS Werkzeug `Vivado High Level Synthesis` kann die Generierung einer Pipeline mittels der Präprozessordirektive `#pragma HLS PIPELINE` innerhalb des Schleifenrumpfes für die gewünschte Schleife eingeschaltet werden.

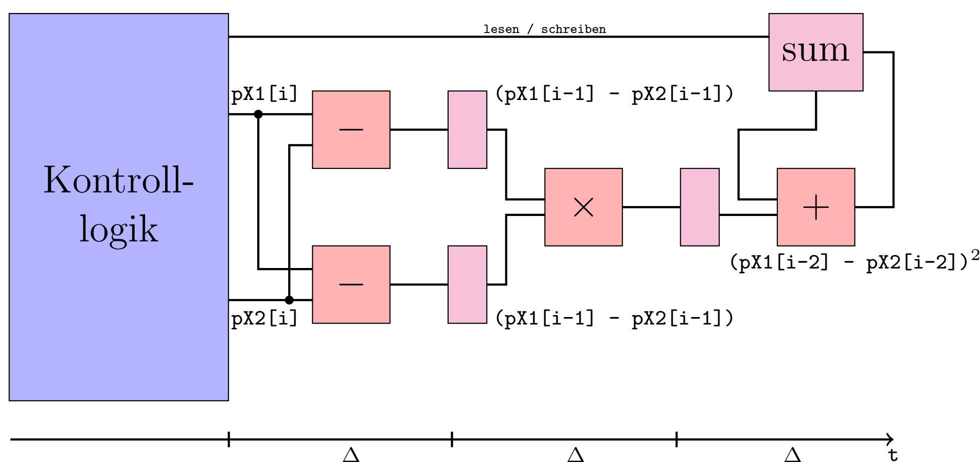


Abbildung 6.8: Schematische Schaltung einer Pipeline für die Berechnung der Kernfunktion

Schleifenabrollen

Das Schleifenabrollen ist ein Konzept, welches vor allem im Bereich der Compileroptimierung bekannt es. Hierbei wird eine Schleife durch eine äquivalente Schleife ersetzt, die M Kopien des Schleifenrumpfes enthält. Der Zugriff auf Elemente innerhalb dieses neuen Schleifenrumpfes, sowie die Manipulation der Zählvariable im Schleifenkopf müssen dann entsprechend der Anzahl der Rumpfkopien angepasst werden. Im extremsten Fall wird M auf die Anzahl der Schleifendurchläufe gesetzt, sodass die Schleife vollständig abgerollt wird. Dann ist der Schleifenkopf nicht mehr notwendig.
In klassischen CPUs bietet das Abrollen von Schleifen den Vorteil, dass der Schleifenkopf seltener oder im Extremfall gar nicht ausgewertet werden muss, wodurch der Verwaltungs-

aufwand verringert wird. Im Gegensatz dazu erhöht sich die Codegröße.
Im Falle von FPGAs bedeutet Schleifenabrollen, dass insgesamt M Schaltungen des Schleifenrumpfes implementiert werden. Abbildung 6.9 zeigt eine vollständig abgerollte Hardwareimplementierung der Schleife für die Kernfunktionsberechnung. Es ist auf einen Blick ersichtlich, dass die benötigte Anzahl der Logikzellen deutlich höher liegt als z.B: beim Pipelining, da Funktionseinheiten nicht wiederverwendet werden. Die Taktfrequenz in diesem Beispiel ähnelt der Taktfrequenz aus der einfachen Implementierung aus Abbildung 6.7, wobei die längste sequentielle Schaltung hier für die gezeigte Baumstruktur mit $2 + log_2(\texttt{dim})$ gegeben ist.
Der große Vorteil von Schleifenabrollen auf FPGAs besteht darin, dass die gesamte Schleifenberechnung nur noch 2 Taktzyklen dauert: Im ersten Takt werden alle Differenzen parallel berechnet, multipliziert und aufsummiert. Im zweiten Takt kann der generierte Wert in die Variable `sum` geschrieben werden. Die Kontrolllogik zerfällt hier auf ein Minimum. Sie muss lediglich dafür sorgen, dass die korrekten Werte `pX1[i]` und `pX2[i]` an den entsprechenden Funktionseinheiten anliegen und mit einem Takt Verzögerung das berechnete Ergebnis in `sum` geschrieben wird.
Im verwendeten HLS Werkzeug `Vivado High Level Synthesis` kann Schleifenabrollen mit einem Faktor M mittels der Präprozessordirektive `#pragma HLS UNROLL factor=M` im Schleifenrumpf für die gewünschte Schleife eingeschaltet werden. Des Weiteren kann `#pragma HLS UNROLL` verwendet werden, um die gesamte Schleife abzurollen.

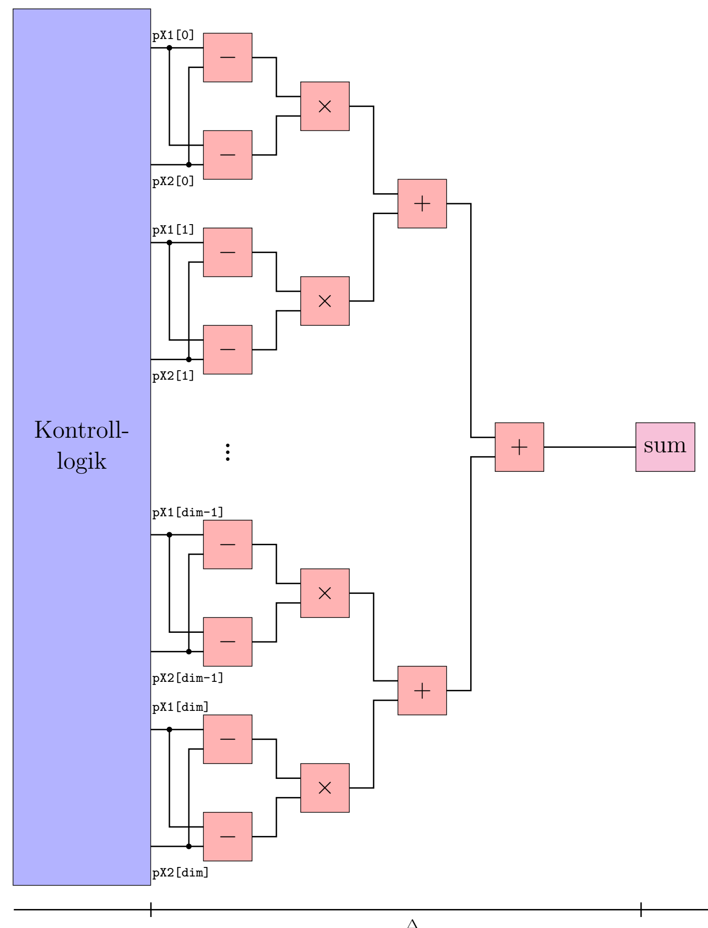


Abbildung 6.9: Schematische Schaltung einer abgerollten Schleifen für die Berechnung der Kernfunktion.

Geschachtelte Schleifen

Wesentlicher Bestandteil der vorgestellten Hochsprachenimplementierung in Abschnitt 6.1 sind geschachtelte Schleifen, die über die Matrixstrukturen iterieren. Geschachtelten Schleifen lassen sich nicht unabhängig voneinander optimieren, sondern müssen zusammen, als Ganzes betrachtet werden.

Pipelining lässt sich konzeptionell nicht auf triviale Weise auf geschachtelte Schleifen anwenden. Bei der Ausführung geschachtelter Schleifen wird zunächst immer die innere Schleife vollständig abgearbeitet und erst danach wird der Schleifenrumpf der äußeren Schleife weiter ausgeführt. Die Ausführung der inneren Schleife ist hierbei häufig von der Zählvariablen der äußeren Schleife abhängig (siehe z.B: Algorithmus 3). Damit muss die Pipeline der äußeren Schleife solange angehalten werden, bis die Implementierung der inneren Schleife vollständig abgearbeitet ist. Eine Ausführung beider Schleifen in einer Pipeline ist nicht sinnvoll, da sie keinen Geschwindigkeitsvorteil bieten würden, sodass das verwendete HLS Tools dies nicht erlaubt [Xilh].

Für das erfolgreiche Pipelinen einer geschachtelten Schleife werden alle innenliegenden

Strukturen automatische in eine flache Hierarchie übersetzt d.h. innere Schleifen werden abgerollt und Funktionen an die entsprechende Stelle kopiert [Xilh].
Das vollständige Abrollen geschachtelter Schleifen bietet mitunter den größten Geschwindigkeitsvorteil, ist aber aufgrund des hohen Verbrauches an Logikzellen häufig unpraktikabel. Zwar steigt die Anzahl der benötigten Logikzellen linear in der Anzahl der geschachtelten Schleifen und Schleifendurchläufe an, doch ist die Anzahl der Schleifendurchläufe häufig datenabhängig, wohingegen die Gesamtanzahl der Logikzellen des FPGAs konstant ist.
Die Kombination von Schleifenabrollen und Pipelining hat sich in der Praxis als guter Kompromiss zwischen Logikzellenverbrauch und Geschwindigkeit gezeigt, wobei hier sowohl die Taktrate vergleichsweise hoch, als auch die Taktverzögerung vergleichsweise niedrig ist (siehe hierzu [Xilh]).

Algorithmenspezifische Optimierung

In den vorangegangenen beiden Abschnitten wurden Pipelining und Schleifenabrollen vorgestellt. Die Wahl der passenden Optimierung muss immer anwendungsspezifisch erfolgen, sodass diese für die konkrete Implementierung getroffen werden muss.
Dennoch lassen sich Abhängigkeiten beider Optimierungen untereinander und zu der Struktur der zu optimierenden Schleife aufzeigen, sodass sich Richtlinien für eine sinnvolle Anwendung dieser Optimierungen ableiten lassen.
Pipelining erhöht zunächst die Latenz der Hardware, da eine höhere Anzahl an Taktzyklen benötigt wird. Des Weiteren wird die Pipeline in einer Anfangs- und Endphase nicht vollständig ausgelastet, sodass ihr Benefit in diesen Phasen nicht zum Tragen kommt. Aus diesem Grund müssen die Pipeliningtiefe und die Dauer der Pipelineausführung in einem passenden Verhältnis zueinander stehen. Auf algorithmischer Ebene gibt die Anzahl der Schleifendurchläufe die Anzahl der Pipelineausführung vor, wohingegen die Pipelinegröße durch die Anzahl der Befehle im Schleifenrumpf gegeben ist. Im ungünstigsten Fall werden genau so viele Befehle im Schleifenrumpf durchgeführt, wie es Schleifendurchläufe gibt, sodass die Pipeline lediglich einmal während des gesamten Durchlaufes vollständig ausgelastet ist. Aus diesem Grund sollte die Anzahl der Schleifendurchläufe deutlich höher sein, als die Anzahl der Teilberechnungen im Schleifenrumpf.
Schleifenabrollen bietet eine einfache Möglichkeit ganze Teilberechnungen des Algorithmus in nur einem Taktzyklus zu berechnen, wodurch es den größten Geschwindigkeitsvorteil verspricht. Dem gegenüber steht nicht nur die begrenzte Anzahl an Logikzellen auf dem FPGA, sondern auch Restriktionen der gewünschten Taktrate. Durch Schleifenabrollen ergeben sich sehr breite Schaltungsstrukturen, die im Place & Route Schritt auf dem FPGA platziert werden müssen. Hier bietet sich dem Synthesetool aufgrund des Abrollens wenig Spielraum für Optimierungen der Signallaufzeiten, sodass die Taktrate beim Abrollen

tendenziell geringer ausfällt (vgl. [Xilh]). Im ungünstigsten Fall verringert sich die Taktfrequenz so stark, dass die Berechnung zwar in nur einem Takt abgeschlossen werden kann, dieser Takt aber mehrere Millisekunden benötigt, wodurch der Vorteil von Schleifenabrollen vollständig negiert ist.

Die bisher geführten Diskussionen haben implizit angenommen, dass die Anzahl der Schleifendurchläufe in allen Fällen bekannt sind. Ist dies nicht der Fall, so kann eine Schleife nicht sinnvoll abgerollt werden, da schlichtweg der maximale Grad des Abrollens nicht bekannt ist. Wird durch Angabe des Abrollgrades lediglich ein Teil der Schleife abgerollt, so muss eine zusätzlich Abbruchlogik generiert werden, um die Schleifen passend zu verlassen. Diese Abbruchlogik muss trotz des Abrollens die Überprüfung der Zählvariable im Schleifenkopf implementieren, wodurch sich kaum ein Vorteil gegenüber einer nicht abgerollten Variante ergibt.
Im Falle von Pipelining ist dieses Problem weniger kritisch, da hier lediglich die Anzahl der Befehle im Schleifenrumpf statisch bekannt sein muss, um eine sinnvolle Pipeliningtiefe zu definieren. Dennoch muss auch hier eine entsprechende Abbruchlogik generiert werden, die zusätzlich die Pipelineausführung sauber beendet wenn die Schleife datenabhängig verlassen wird[5].
Für die Implementierung von Projection GP bedeutet das zunächst, dass Schleifenabrollen nicht angewendet werden kann, da die Anzahl der Berechnungsschritte von der aktuellen Anzahl der Basisvektoren `bvCnt` abhängt (vgl. Abschnitt 6.1). Um den Durchsatz der vorliegenden Implementierung weiter zu erhöhen, sollte jedoch ein Großteil der Schleifen abgerollt werden.
Betrachtet man den algorithmischen Ablauf von Projection GP genauer, so fällt auf, dass sich der Algorithmus in zwei Phasen unterteilen lässt. In der ersten Phase werden Basisvektoren hinzugefügt, d.h. `bvCnt` erhöht sich mit jedem betrachteten Beispiel $\vec{x}_i$ um Eins. In der zweiten Phase hingegen wird jeweils ein Basisvektor hinzugefügt, dann die Scorefunktion berechnet und anschließend ein Basisvektor gelöscht, sodass `bvCnt` effektiv konstant bleibt.
Die Schleifendurchläufe sind also insbesondere in der zweiten Phase bekannt, wodurch sich Schleifenabrollen anwenden lässt. Insgesamt ergeben sich drei Ansätze zur Nutzung von Schleifenabrollen für Projection GP auf dem FPGA:

1. Die erste Phase des Algorithmus wird auf dem Host-PC ausgeführt und die resultierenden Matrizen C und Q, sowie der Vektor α dann entweder während der Synthesephase in den Quellcode eingebettet oder vor Ausführung auf das FPGA mittels TCP/IP übertragen und in einen entsprechenden Speicherbereich kopiert.

[5] Dies kann z.B. durch eine Kombination einer bedingte Sprunganweisung und dem `break` Schlüsselwort passieren.

2. Die erste Phase des Algorithmus wird auf dem FPGA selber ausgeführt, wobei hierzu entweder eine nicht optimierte Hardwareeinheit oder der `MicroBlaze` Prozessor verwendet wird.

3. Die erste Phase des Algorithmus wird vollständig übersprungen und es wird direkt die zweite Phase ausgeführt. Hierbei müssen die Modellparameter α, C und die Basisvektoren X zunächst initialisiert werden. An dieser Stele kann zusätzliches Anwendungswissen verwendet werden, wobei dieses Vorgehen der Ausführung der ersten Phase auf dem Host-PC sehr nahe kommt.
   Ist kein Vorwissen bekannt, so können α, C und die Menge der Basisvektoren zufällig initialisiert werden. Im Falle der Matrix Q muss weiterhin gelten, dass sie eine Inverse der Kernmatrix abbildet. Da durch Überspringen der ersten Phase keinerlei sinnvollen Basisvektoren vorliegen, ist es daher zweckdienlich die Matrix Q als Einheitsmatrix zu initialisieren, um weiterhin einen Bezug zu einer Inversen Matrix zu behalten.

Zunächst ist festzuhalten, dass Projection GP als Online Algorithmus in einem Datenstrom, welcher Daten kontinuierlich produziert und die vom Algorithmus kontinuierlich verarbeitet werden müssen, angewendet werden kann. In einem unendlich langen Datenstrom wird die erste Phase von Projection GP lediglich in einer vergleichsweise kurzen Anfangsphase ausgeführt. Danach wird auf den übrigen Daten nur die zweite Phase des Algorithmus ausgeführt. Zusätzlich zeigen die Experimente in Abschnitt 7.3, dass für die vorliegenden Datensätze bereits eine im Vergleich zur Datenmenge relativ geringe Anzahl an Basisvektoren gute Ergebnisse liefert, womit sich die Ausführung der erste Phase weiter verkürzt.
Es zeigt sich also, dass der Algorithmus im Wesentlichen die zweite Phase ausführt, sodass hier eine Beschleunigung einen deutlichen Effekt hat.
Vorgehen Nummer Eins bietet sich vor allem an, wenn eine sinnvolle Vorberechnung der Matrizen möglich ist, da es den Mehraufwand der Hardware verringert. Dem Gegenüber wird der Algorithmus zum Teil auch auf einem externen Gerät ausgeführt, sodass das FPGA nicht mehr vollständig als Einzelgerät zu sehen ist.
Vorgehen Nummer Zwei behält die Eigenschaften des FPGAs als Einzelgerät bei, wobei hier ein höherer Hardwareaufwand gegeben ist. Gerade die Ausführung der ersten Phase auf dem Soft-Prozessor verlangt die Konfiguration von Gleitkommaeinheiten, welches sich konträr zu den bisher vorgestellten Entwurfsprinzipien verhält.
Vorgehen Nummer Drei vereinigt die Vorteile der beiden anderen Vorgehensweisen, wobei hier der Einfluss der zufällige Initialisierung von α und C auf die Vohersagegüte unklar ist.

6.3.3 Diskussion

Im vorangegangen Abschnitt wurde eine Implementierung von Projection GP vorgestellt und einige Optimierungen diskutiert. Diese Implementierung kann nun im Bezug auf das restliche System evaluiert werden.
Mit dem gewählten Speicherlayout lässt sich die maximale Anzahl der Basisvektoren `bvMax` bereits zur Kompilierzeit angeben. Bezeichne hierzu M den maximal zur Verfügungen stehenden Speicher auf dem FPGA:

$$2 \cdot \texttt{bvMax}^2 + 4 \cdot \texttt{bvMax} + \texttt{bvMax} \cdot \texttt{dim} \leq M$$

Es folgt:

$$\texttt{bvMax} \leq \sqrt{\frac{1}{2}M + \left(\frac{4 + \texttt{dim}}{4}\right)^2} - \left(\frac{4 + \texttt{dim}}{4}\right)^2$$

Das gewählte Speicherlayout verwendet Blockram zur Implementierung von α, C, Q und X. Der verwendete `XC7A200T-2FBG676C` FPGA Chip bietet insgesamt 13149Kb Blockram Speicher an. Tabelle 6.5 folgenden werden davon bereits ca. 22 % im restlichen System aus Abschnitt 6.2 verwendet, sodass dem Lernalgorithmus noch $\frac{13149 \cdot 0{,}78}{8} = 1282{,}0275$ KB Speicher zur Verfügung steht. Einfache Gleitkommazahlen belegen 4 Byte Speicher, sodass insgesamt ca. $M = 320506$ Gleitkommazahlen gespeichert werden können. Für das verwendete Beispiel mit $\texttt{dim} = 21$ ergibt sich eine maximale Anzahl der Basisvektoren von $\texttt{bvMax} \leq 361$. Die tatsächliche Anzahl an möglichen Basisvektoren im realen System wird etwas niedriger sein als diese obere Grenze, da lokale Variable und Optimierungen ebenfalls entsprechenden Speicher verbrauchen. Wie in Kapitel 7 gezeigt, liefern jedoch bereits $\texttt{bvMax} \approx 100$ Basisvektoren gute Ergebnisse.
Da das umliegende System lediglich knapp 5 % an Logikzellen verbraucht, kann der Algorithmus fast unabhängig davon optimiert werden. Hier ist zu beachten, dass mit steigender Anzahl der Basisvektoren auch die Anzahl der benötigten Logikzellen für das Abrollen von Schleifen steigt. Zusätzlich können geschachtelte Schleifen aufgrund der hohen Anzahl an benötigten Logikzellen nicht beliebig abgerollt werden.
Das High Level Synthese Werkzeug `Vivado HLS` ermöglicht eine Analyse des erzeugten Hardwareblocks im Bezug auf Taktverzögerung, geschätzte mögliche Taktfrequenz und geschätzten Logikzellenverbrauch, sodass verschiedene Optimierungen miteinander verglichen werden können. Da in der ersten Phase des Algorithmus die Anzahl der Schleifendurchläufe nicht bekannt ist, wird hier ein minimaler und ein maximaler Wert mittels der Präprozessordirektive `HLS LOOP_TRIPCOUNT min=0` bzw. `HLS LOOP_TRIPCOUNT max=bvMax` im Schleifenrumpf angegeben. Diese Direktive haben keinerlei Einfluss auf den generierten Hardwarecode, sondern dienen dem Tool lediglich zur Abschätzung der minimalen und maximalen Anzahl an Taktzyklen für die entsprechende Schleife.
Dem bisher verwendeten Beispiel folgenden, zeigen Tabelle 6.7 und Tabelle 6.8 die Auswirkungen der angesprochen Kombination von Schleifenabrollen und Pipelining, sowie der

Aufteilung des Algorithmus in zwei Phasen für Projection GP mit 100 Basisvektoren und 21-dimensionale Beobachtungen. Gezeigt wird jeweils die Anzahl der verbrauchten Logikzellen (LUT), die Anzahl der benutzen Flipflops (FF), sowie die Benutzung von Blockram (BRAM) und der DSP-Elemente (DSP). Zusätzlich ist die benötigte Latenz zur Bearbeitung eines Beispiels $\vec{x}_i$ in der Anzahl der Taktzyklen, sowie die durch das HLS Werkzeug geschätzte Taktrate angegeben, woraus sich dann der geschätzte Durchsatz ergibt. Die Implementierungen wurde jeweils in die beiden angesprochenen Phasen aufgeteilt, wobei hier zu beachten ist, dass zusätzlicher Hardwarecode generiert werden muss, um diese beiden Phase zu koordinieren. Damit ist die Gesamtlatenz und der Gesamtverbrauch der jeweiligen Algorithmen höher als die Summe der Einzelverbrauche der einzelnen Phasen.
Bei der Wahl der Optimierungen wurde versucht den Durchsatz zu maximieren, ohne dabei die Logikzellen des FPGAs unnötig zu verschwenden. Damit sind die für die Implementierung verwendeten Optimierungen nicht als erschöpfend und optimal zu betrachten, sondern bieten sicherlich noch Spielraum für eine tiefgreifendere Analyse in weiteren Arbeiten.
Zunächst fällt auf, dass die nicht optimierte Variante lediglich 28% der Logikzellen des Boards verwendet und damit nur knapp 94 Elemente pro Sekunde bearbeiten kann, wobei die Taktrate mit 200 MHz relativ hoch gewählt werden kann.
Führt man manuelle Optimierungen durch, so werden fast vier mal weniger Taktzyklen benötigt, wobei die Taktrate mit nur 91 MHz im Wesentlichen halbiert wurde. Das führt dazu, dass die optimierte Variante mit 156 Elementen pro Sekunde fast doppelt so schnell ist, wie die nicht optimierte Variante.
Teilt man den Algorithmus wie beschrieben in zwei Phasen auf, so zeigt sich zunächst ein paradoxes Bild. In der zweiten Phase muss über alle Basisvektoren iteriert werden, die Scorefunktion berechnet werden und ein entsprechender Basisvektor gelöscht werden. Damit wird in der zweiten Phase deutlich mehr Arbeit geleistet, als in der ersten Phase. Dem Gegenüber steht die Tatsache, dass nun Schleifengrenzen zur Übersetzungszeit bekannt sind, sodass Schleifen passend abgerollt werden können. Damit kann die Latenz im Vergleich zur zweiten Phase um einen Faktor von fast Acht verringert werden, was insgesamt zu einem Durchsatz von ca. 1212 Elementen pro Sekunde in der zweiten Phase führt. Die erste Phase kann mit ca. 34% an Logikzellen des FPGAs implementiert werden, wohingegen die schnelle, zweite Phase fast 55% Logikzellen benötigt, was zusammen 93% der Logikzellen des FPGAs verbraucht.
Implementiert man nur die zweite Phase des Algorithmus, so können die 34% Logikzellen aus der ersten Phase des Algorithmus für weitere Optimierungen genutzt werden, sodass dieser Implementierung insgesamt 76% der Logikzellen verbraucht. Damit kann sowohl die Taktrate auf 150 MHz erhöht werden, als auch die Latenz leicht verringert werden. Damit erreicht man insgesamt einen Durchsatz von ca. 2037 Elementen pro Sekunden, was um einen Faktor 21 schneller ist als die nicht optimierte Variante.

	LUT [%]	FF [%]	BRAM [%]	DSP [%]
Keine Optimierung	28	17	13	13
→ Phase 1	13,72	8,56	0	6,1
→ Phase 2	13,05	8,29	0	6
Mit Optimierung	93	59	11	17
→ Phase 1	33,97	24,59	0	7,16
→ Phase 2	54,14	34,27	0	7,7
Phase 2 alleine	76	42	11	33

Tabelle 6.7: Geschätzter Verbrauch der FPGA-Ressourcen von Projection GP nach Optimierungen für `dim` = 21 und `numBV` = 100. Es wurde jeweils der RBF-Kernel nach Abbildung 6.5 verwendet.

	Latenz	Geschätzter Takt [Mhz]	Geschätzter Durchsatz [#elem/s]
Keine Optimierung	2119918	200	94,34
→ Phase 1	2109406	200	94,8
→ Phase 2	2119914	200	94,34
Mit Optimierung	580888	91	156,65
→ Phase 1	580884	91	156,66
→ Phase 2	75030	91	1212,85
Phase 2 alleine	73619	150	2037,52

Tabelle 6.8: Geschätzte Taktrate, sowie geschätzter Durchsatz von Projection GP nach Optimierungen für `dim` = 21 und `numBV` = 100. Es wurde jeweils der RBF-Kernel nach Abbildung 6.5 verwendet.

Der durch das HLS Werkzeug generierte Hardwareblock kann nun in das zuvor beschriebene Gesamtsystem integriert werden. Anschließend kann der `C`-Code für den `MicroBlaze` Prozessor implementiert werden, der den Hardwareblock entsprechend ansteuert. Um Änderungen im Systemdesign einfach durchführen zu können und um Fehler in der Implementierung möglichst schnell zu finden wird - wie bereits angedeutet - zunächst ein stellvertretender Hardwareblock benutzt. Hierzu wurde ein einfaches Perzeptron für zweidimensionale Beispiele (vgl. [HTF01], Kapitel 4.5.1) implementiert. Ein Perzeptron ist ein einfacher linearer Klassifikator, der seine Vorhersage mit einem einfachen Gradientenverfahren anpasst. Diese Anpassung kann in einem Online Algorithmus formuliert werden, sodass die Hardwareschnittstelle von Projection GP und dem Perzeptron gleich implementiert werden kann. Beide Verfahren bekommen in ihrer Trainingsphase eine Beobachtung $\vec{x}_i$ mit Label y_i präsentiert, wohingegen in der Vorhersagephase nur die Beobachtung $\vec{x}$ zur Verfügung steht. Um zwischen den beiden Phasen zu unterschieden, wird ein entspre-

chendes Eingangssignal am Hardwareblock gesetzt.
Damit kann neben dem allgemeinen Systemdesign auch gleichzeitig die Verbindung von Projection GP zum Soft-Prozessors mit Hilfe des simpleren Perzeptrons entwickelt werden. Des Weiteren zeigt dieses Vorgehen die einfache Austauschbarkeit des maschinellen Lernalgorithmus: Durch das Nutzen der `AXI` Verbindung können die Hardwareblöcke verschiedener Lernalgorithmen einfach ausgetauscht werden, solange diese auch die `AXI` Schnittstelle respektieren. Lediglich der den Hardwareblock steuernde `C`-Code im Soft-Prozessor muss für die einzelnen Verfahren angepasst werden.
Für die nachfolgenden Experimente zur Vorhersagegüte von Projection GP (siehe Abschnitt 7) wurde jeweils nur die zweite Phase von Projection GP mit 100 Basisvektoren für 13 bzw. 21 dimensionale Beispiele erzeugt. Die Synthese wurde auf einem `Intel Xeon W3565` Rechner mit 8 Kernen und 24 GB Arbeitsspeicher ausgeführt. Um die Dauer der Synthese und Place & Route gering zu halten, wurde im Synthesetool `Vivado` die `Flow_RuntimeOptimized` Direktive für die Synthese und das Place & Route ausgewählt. Diese Direktive weist das Werkzeug dazu an, die Laufzeit der Synthese und des Place & Route Schrittes zu Ungunsten der allgemeinen Systemperformanz gering zu halten. Der Soft-Prozessor wurde in allen Systemen mit 50 MHz betrieben, wohingegen das Perzeptron und Projection GP mit 25 MHz betrieben werden.
Tabelle 6.9 zeigt den realen Ressourcenverbrauch der angesprochenen Systeme nach ihrer Synthese und dem Place & Route Schritt. Zusätzlich ist die Dauer der Synthese und des Place & Route Schrittes angegeben.
Wie zu erwarten, verbraucht das Entwicklungssystem mit Perzeptron nur eine minimale Anzahl an FPGA-Ressourcen, sodass hier die Belegung von Logikzellen, Flipflops, Blockram und DSP-Elementen vergleichbar zu einem System ohne maschinellen Lernalgorithmus aus Tabelle 6.5 ist. Die Generierung des Bitstroms dauert insgesamt knapp 15 Minuten und erlaubt daher ein schnelles Erkennen von Fehlern im Design.
Projection GP benötigte mit fast 70 % bzw. fast 80 % für `dim`=13 bzw. `dim`=21 einen Großteil der Logikzellen des FPGAs. Des Weiteren werden in beiden Implementierungen knapp $45 - 47$ % der Flipflops verwendet. Die Benutzung von Blockram ist mit knapp 32 % in beiden Fällen annähernd gleich. Im Falle von `dim`=13 werden knapp 40 % der DSP-Elemente benutzt, wohingegen für `dim`=21 nur 33 % der DSP-Elemente benötigt werden.
Die Generierung des Bitstroms von Projection GP für 13-dimensionalen Beispielen dauert insgesamt knapp $2,5$h, wohingegen eine Generierung für 21-dimensionale Beispiele fast $3,5$h dauert.
Es ist zunächst festzuhalten, dass die hier beschriebenen Implementierungen einen Großteil der FPGA-Ressourcen ausnutzen. Des Weiteren scheint die Schätzung des HLS Tools passend. Addiert man die in Tabelle 6.7 gezeigten Schätzungen mit den Werten in Tabelle 6.5, so ergeben sich annähernd die hier gezeigten Werte.
Unter Verwendung des Perzeptrons konnte die Entwicklungszeit entscheidend verkürzt

werden, da nun der Bistrom 10 bis 14 mal schneller generiert werden konnte, sodass Fehler einfacher entdeckt werden konnten.
An dieser Stelle fällt die nicht deterministische Natur des Place & Route Schrittes auf. Der innerhalb des Place % Route Schrittes verwendete Optimierer initialisiert zum Teil Startlösungen per Zufall. Im Falle von PGP für 13 Dimensionen schient der Optimierer hier eine günstige Startlösung gefunden zu haben, sodass sich die Dauer von Place & Route deutlich verkürzte. Dieser Argumentation folgenden, ist jedoch auch zu sehen, dass diese Konfiguration mehr DSP-Elemente benötigt. Hier zeigt sich zusätzlich zur zufälligen Wahl der Startlösung des Optimierers auch die gewählte Optimierdirektive, die zu Gunsten der Laufzeit das System weniger stark optimiert.
Die maschinellen Lernalgorithmen werden in dem hier gezeigten Beispiel mit lediglich 25 MHz Takt betrieben, was bei weitem nicht dem geschätzten Maximaltakt des HLS Werkzeuges entspricht (siehe 6.8). Eine Erhöhung der Taktrate wurde in Ansätzen während der vorliegenden Arbeit versucht, konnte jedoch nicht vollständig durchgeführt werden. Hier zeigte sich zunächst, dass durch Änderung der Optimierdirektiven diese nach Acht bis Zwölf Stunden keinen passenden Bitstrom generieren konnte. Eine genauere Untersuchung des Designs zeigte, dass lediglich ein geringer Anteil von Rund Zehn bis Zwölf Pfaden, die einen Reset einiger Blockrameinheiten durchführen die Zeitbedingungen nicht einhalten konnte, das übrige System aber problemlos mit Taktraten über 100 MHz betrieben werden kann. Neben einer weiteren Optimierung der Syntheseeinstellungen kann bei einer so geringen Anzahl an Pfaden auch eine händische Optimierung durch ein passendes Floorplanning (vgl. [Xilg]) erfolgen. Beim Floorplanning wird dem Synthesetool die Benutzung bestimmter FPGA Ressourcen vorgeschrieben, indem dedizierte Bereiche für bestimmte Teillogik vorgesehen wird. Einen ähnlichen Effekt wie das Floorplanning bietet die partielle Rekonfiguration, wobei hier ganze Teilbereiche des FPGAs für bestimmte Hardwareblöcke, unabhängig von ihrer Einbindung in das Gesamtsystem vorgesehen werden. Damit schient eine Erhöhung der Taktrate durchaus im Bereich des Möglichen, wobei auch hier eine tiefergehende Analyse erforderlich ist.

	LUT [%]	FF [%]	BRAM [%]	DSP [%]	Dauer der Synthese	Dauer von Place & Route
Perzeptron	5,83	3,29	21,9	1,35	00:09:51	00:04:31
PGP,`dim`=13	69,27	44,73	31,92	40,95	01:47:17	00:23:34
PGP,`dim`=21	79,36	47,22	32,6	33,24	01:59:11	01:22:40

Tabelle 6.9: Realer Verbrauch der FPGA-Ressourcen für das Entwicklungssystem mit einem einfachen Perzeptron, sowie für die zweite Phase von Projection GP (PGP). Das Perzeptron ist für zweidimensionale Beobachtungen konfiguriert und wird mit 25 MHz betrieben. Projection GP wird für 13 bzw. 21 Dimensionen synthetisiert, wobei jede Implementierung mit 25 MHz getaktet ist. Der `MicroBlaze` Prozessor ist in allen System unabhängig vom Algorithmus mit 50 MHz getaktet.

7 | Experimente

Im vorangegangenen Kapitel wurde ein System mit Netzwerkkommunikation und Soft-Prozessor für FPGAs entworfen und die Implementierung von Projection GP als zusätzliche Hardwareeinheit beschrieben. Dabei wurden insbesondere auf den Durchsatz von Projection GP und den Energieverbrauch des FPGAs eingegangen. In diesem Kapitel soll nun diese FPGA Implementierung mit einer Implementierung auf handelsüblichen CPUs verglichen werden.
Zunächst wird die Vorhersagegüte von Projection GP mit einem vollständigen Gauß-Prozess verglichen, um die Wahl dieses Algorithmus zu untermauern. Hierzu wird der Boston Housing Datensatz [UCI] und der Sarcos Inverse Kinematics Datensatz verwendet [RW].
Anschließend wird die vorgestellte Hochsprachenimplementierung auf Desktop-Hardware, sowie auf Hardware aus dem Bereich der eingebetteten Systeme ausgeführt und mit der FPGA Implementierung verglichen. Der Argumentation im vorangegangen Abschnitt folgenden, wird an dieser Stelle angenommen, dass die durch das HLS Werkzeug geschätzte Taktrate bei ausreichender Optimierung im Synthese Schritt und während des Place & Route Schrittes erreicht werden. Aus diesem Grund erfolgt die Untersuchung der FPGA Implementierung auf Basis der theoretischen Schätzung des HLS Werkzeuges. Ergänzend wird der tatsächlich erreichte Durchsatz für die beiden synthetisierten System mit einer Taktrate von 25 MHz angegeben.
Abschnitt 7.1 geht kurz auf die verwendeten Datensätze ein, wohingegen Abschnitt 7.2 die verwendeten Hardware präsentiert. Anschließend werden in Abschnitt 7.3 die Experimente zur Klassifikationsgüte vorgestellt. Abschnitt 7.4 und Abschnitt 7.5 untersuchen anschließend den Durchsatz, den Energieverbrauch, sowie die Energieeffizienz der vorgestellten Implementierungen. Das Kapitel wird mit einer Diskussion in Abschnitt 7.6 beendet.

7.1 Datensätze

Der Boston Housing Datensatz hat 506 Trainingsbeispielen mit je 13 Attributen. Das Ziel für diesen Datensatz ist die Vorhersage von Immobilienwerten für verschiedene Häuser im Gebiet um Boston in den USA auf Basis handdefinierter, numerischer Merkmale. Der

Datensatz repräsentiert einen kleinen Standarddatensatz, der bereits eingehend studiert worden ist. Mit lediglich 506 Trainingsbeispielen kann man einen vollständigen Gauß-Prozess auf dem Datensatz berechnen, um so eine Vergleichsgrundlage zu schaffen.
Der Sarcos Inverse Kinematics Datensatz hat 44484 Trainingsbeispiele mit 28 Dimensionen und 4449 Testbeispielen. Die Aufgabe für diesen Datensatz ist die Lösung des Inversen Kinematik Problems für einen Roboterarm, wobei die letzten 7 Attribute die vorherzusagenden Gelenkkräfte darstellen. In der Literatur wird üblicherweise nur das Inverse Kinematik Problem für die erste Gelenkachse gelöst, sodass die folgenden Experimente sich im Sinne einer Vergleichbarkeit auch auf diese Achse beschränken. Mit seinen 44484 Trainingsbeispielen ist die Berechnung eines vollständigen Gauß-Prozesses extrem zeitaufwendig, sodass hier Projection GP seine Stärken als Online-Algorithmus ausspielen kann.
Um die Skalierbarkeit der verschiedenen Systeme miteinander zu vergleichen sind die beiden Datensätze mit lediglich 13 bzw. 21 dimensionalen Beispielen nicht ausreichend. Um sowohl den Effekt der Dimensionalität, als auch den Einfluss der Menge der Basisvektoren zu untersuchen, wird zusätzlich ein einfacher Datengenerator genutzt. Dieser Datengenerator erzeugt zunächst eine lineare Funktion der gewünschten Dimensionalität mit zufälligen Koeffizienten. Anschließend wird diese lineare Funktion genutzt, um die Label für 50000 zufällige erzeugte Beobachtungen zu berechnen. Diese 50000 Beispiele werden anschließend für eine Modellberechnung benutzt, wobei hier der Durchsatz und Energieverbrauch gemessen wird.

7.2 Verwendete Architekturen

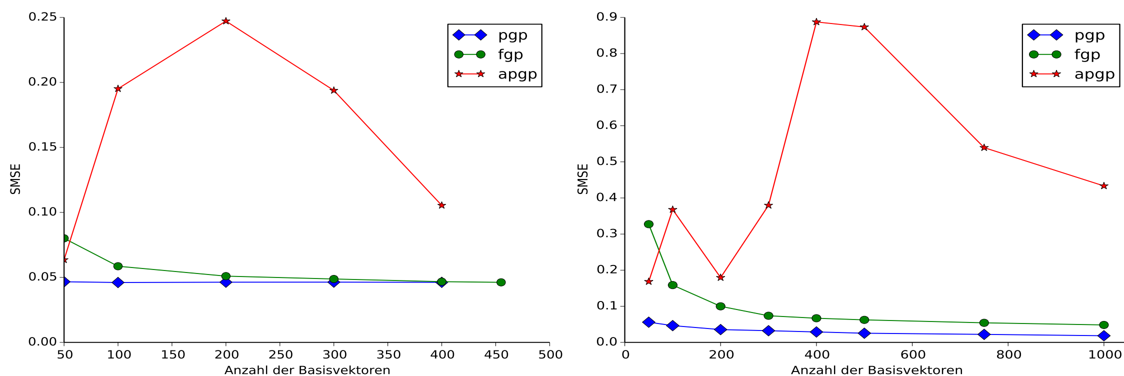
FPGAs bieten sich als stromsparende Alternative mit dem Potential zu ähnlichen Geschwindigkeiten wie handelsüblichen Desktop CPUs an. Aus diesem Grund kommen insgesamt drei Vergleichssysteme zum Einsatz.
Zum Einen repräsentiert ein `Intel i7-3700` mit 8 Kernen und 12 GB Hauptspeicher einen üblichen Desktop PC, der relativ viel Energie verbraucht und eine vergleichsweise hohe Rechenkapazität bietet. Als Kompromiss zwischen Rechenkapazität und Energieverbrauch wird zum Anderen ein Laptop mit einer `Intel i7-4600U Mobile` CPU mit 4 Kernen und 8 GB Hauptspeicher verwendet. Als Vertreter des eingebetteten Systembereiches wird das FPGA zu guter Letzt noch mit einem `Raspberry PI 2B` verglichen. Dieser beinhaltet eine `ARM v7` CPU mit 4 Kernen und 1 GB Hauptspeicher.
Die Laufzeitmessungen können dabei auf den drei Systemen direkt in den Quellcode eingebettet werden. Im Falle des FPGAs wird wie bereits angesprochen der theoretisch maximale Durchsatz angegeben.
Im Falle der Energiemessungen stellt sich zunächst die Frage nach der Vergleichbarkeit von Energiemessungen. Das FPGA bildet ein in sich geschlossenes System, welches voll-

ständig auf die Ausführung von Projection GP konfiguriert wurde. Dem Gegenüber ist sowohl der Desktop PC, als auch der Laptop für einen viel allgemeineren Gebrauch ausgelegt, sodass hier viele Komponenten wie z.B. die Grafikkarte oder WLAN enthalten sind, die für Projection GP nicht gebraucht werden. Des Weiteren ist es gerade im Bereich der Desktop PCs durch die große Auswahl verschiedener Hardwarekomponenten schwer, eine Vergleichbarkeit herzustellen.
Betrachtet man auf der anderen Seite lediglich den Energieverbrauch der CPU innerhalb dieser Architekturen, so wird man hier nicht den Energieverbrauch von Speicherzugriffen und Mainboardpheripherie wie Interrupt Controller oder Direct Memory Access mitmessen. In diesem Fall ist der gemessene Energieverbrauch geringer als der Tatsächliche.
Um den Gesamtverbrauch des Systems zu messen, wird zusätzliche Hardware benötigt, wohingegen die Messung des Energieverbrauchs der CPU per Software erfolgen kann. Hier bietet `Intel` unter [Veg] die `Intel Rapl` Bibliothek an, welche auf Basis der `Intel Performance Counter` eine aktuelle Schätzung über den Energieverbrauch einiger unterstützter `Intel` CPUs angibt. Die nachfolgenden Energiemessungen beschränken sich auf die Messung dieses CPU Verbrauchs.
Für den `Raspberry PI 2B` ist die Situation etwas anders. Dieser ist zunächst ein in sich geschlossenes System, welches sich lediglich durch USB Peripherie und GPIO Pins erweitern lässt. Betrachtet man also das `Raspberry PI 2B` einzeln, ohne jegliche Peripherie, so ist durchaus eine gewisse Vergleichbarkeit gegeben. Zunächst lässt sich der maximale Stromverbrauch des `Raspberry PI 2B` abschätzen. Der Hersteller empfiehlt das Gerät mit einem Netzteil mit $1,5$A Strom bei 5V Spannung zu betreiben [PI/b], was zu einer maximalen Leistungsaufnahme von $7,5$ W führt. Dieser Wert ist als maximale Obergrenze anzusehen, der lediglich bei voller Systemlast und entsprechender angeschlossener Peripherie erreicht wird. Es gibt eine Vielzahl verschiedener Messreihen für das `Raspberry PI 2B`, wobei diese sich zum Teil durch die ausgeführte Software, sowie angeschlossene Peripherie stark unterscheiden. In dem vorliegenden Anwendungsfall wird lediglich Zugriff auf die Ethernetschnittstelle, jedoch keine zusätzliche Peripherie am `Raspberry PI 2B` benötigt. Des Weiteren lastet die Projection GP Implementierung nur einen Kern des `ARM v7` Prozessors vollständig aus. Unter [PI/a] ist eine Messreihe dieses Anwendungsfalls zu finden, wobei hier eine Leistungsaufnahme von 2W gemessen wird.
Im Falle des FPGAs ist eine Abschätzung des Energieverbrauches ohne Generierung des Bitstroms schwierig, da dieser von der Taktrate, sowie der Ausnutzung der FPGA-Ressourcen abhängt. Aus diesem Grund wird jeweils auf den Energieverbrauch der beiden synthetisierten Systeme verwiesen.

7.3 Vorhersagegenauigkeit

Zur Messung der Vorhersagegenauigkeit wird der Standardized Mean Squared Error (SMSE) (siehe Abschnitt 4.1) verwendet. Dieser gibt den durchschnittlichen quadrierten Fehler im Bezug auf die Standardabweichung der Testbeispiele an. Wird also immer der Mittelwert der Label im Testdatensatz vorhergesagt, so ist der SMSE annähernd Eins.
Für beide Datensätze wird Projection GP mit einem vollständigen Gauß-Prozess und der in Abschnitt 6.2 skizzierten approximativen Variante von Projection GP ohne die erste Phase verglichen. In der Approximation werden α, C und die Menge der Basisvektoren als Nullvektor bzw. Nullmatrix und Q als Einheitsmatrix initialisiert.
Für beide Projection GP Varianten wird die Anzahl der Basisvektoren jeweils variiert. Im Falle eines vollständigen Gauß-Prozesses ist die Berechnung auf allen Daten nicht immer möglich. Hier wird entsprechend der Anzahl der Basisvektoren der Projection GP Varianten eine Stichprobe der Daten gezogen und der vollständige Gauß-Prozess auf dieser Teilmenge berechnet. Um günstige und ungünstige Stichproben zu vermeiden, wird dieses Vorgehen 10 mal wiederholt und der durchschnittliche SMSE angegeben.
Als Kernfunktion wird eine radiale Basisfunktion verwendet, wobei auch hier l und γ variiert werden. Es wird lediglich der beste SMSE aller Parameter angegeben.
Zu den Boston Housing Daten liegt kein Testdatensatz vor, sodass eine 10-fache Kreuzvalidierung durchgeführt wird. Für diesen Datensatz kann zusätzlich ein vollständiger Gauß-Prozess auf 451 Datenpunkten berechnet werden.
Im Falle der inversen Kinematik Daten liegt ein Testdatensatz vor, sodass dieser verwendet wird. Wegen der Größe des Traingsdatensatzes kann jedoch kein vollständiger Gauß-Prozess auf allen Daten berechnet werden, sodass hier nur die Stichproben verwendet werden.
Abbildung 7.1a zeigt den SMSE für die drei Algorithmen bei ihrer Anwendung auf dem Boston Housing Datensatz. Zunächst zeigt sich, dass ein approximierter Projection GP ein sehr gemischtes, schon fast zufälliges Verhalten aufweist. Bei einer geringen Anzahl von 50 Basisvektoren ist der Vorhersagefehler vergleichbar mit einem normalen ProjectionGP oder einem vollständigen Gauß-Prozess. Erhöht man die Anzahl der Basisvektoren, so verschlechtert sich die Vorhersagegenauigkeit jedoch zunehmend bis zu einem Wert von knapp unter $0,25$. Ab dann verbessert sich die Vorhersagegenauigkeit für noch mehr Basisvektoren wieder, wobei niemals ein zu Projection GP oder zu einem vollständigen Gauß-Prozess vergleichbarer Wert erreicht wird.
Dem Gegenüber zeigt sich, dass ProjectionGP bereits bei 50 Basisvektoren einen sehr guten Vorhersagefehler von ungefähr $0,05$ erreicht, der sich im Folgenden jedoch nur marginal verbessert. Dem Gegenüber verbessert sich ein vollständiger Gauß-Prozess bei einer größeren Strichprobe erwartungsgemäß stetig von ca. $0,075$ bis $0,05$. Schließlich schneidet ein echter vollständiger Gauß-Prozess, d.h. derjenige Prozess, der alle Daten verwendet,

mit einem Fehler von knapp unter $0,05$ am besten ab, wobei hier aufgrund der Achsenskalierung und der sehr geringen Unterschiede zwischen den Verfahren ein direkter Vergleich schwierig ist.
Abbildung 7.1b zeigt den SMSE für die drei Algorithmen bei ihrer Anwendung auf den Sarcos Inverse Kinematics Datensatz. Hier ergibt sich ein ähnliches Bild für den SMSE wie bereits für den Boston Housing Datensatz. Der approximative Projection GP verhält sich sehr gemischt und zeigt sprunghaftes Verhalten. Mit einer geringen Anzahl von 50 Basisvektoren ist sein Vorhesagefehler besser als der eines vollständigen Gauß-Prozesses, wobei auch hier analog eine deutliche Verschlechterung des Vorhersagefehlers bei einer höheren Anzahl von Basisvektoren zu sehen ist. Schließlich erreicht er einen Vorhersagefehler von fast 1 bei 400 Basisvektoren und hat damit im Wesentlichen den gleichen Fehler wie die Vorhersage des Mittelwertes für jeden Datenpunkt.
Dem Gegenüber bietet Projection GP wieder den geringsten Fehler mit unter $0,1$, wobei sich hier jetzt eine Verbesserung des Fehlers mit mehr Basisvektoren einstellt.
Der vollständige Gauß-Prozess kann ebenfalls von einer größeren Stichprobe profitieren und verbessert sich auch hier stetig. Er erreicht mit einem Vorhersagefehler von ca. $0,35$ bis $0,1$ jedoch nicht ganz die Güte von ProjectionGP.



(a) Standardized Mean Squared Error von Projection GP (PGP), einem vollständigen Gauß-Prozess (FGP) und der Approximation von Projection GP (APGP) für verschiedene Anzahlen von Basisvektoren auf dem Boston Housing Datensatz.

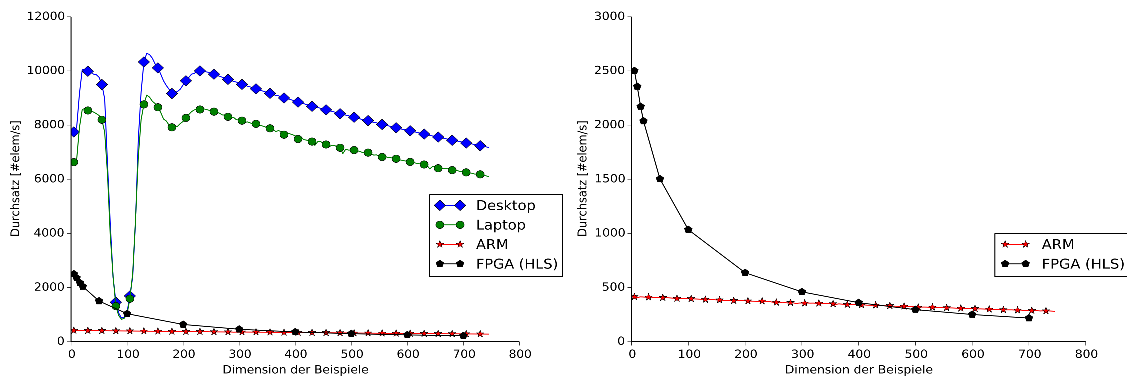
(b) Standardized Mean Squared Error von Projection GP (PGP), einem vollständigen Gauß-Prozess (FGP) und der Approximation von Projection GP (APGP) für verschiedene Anzahlen von Basisvektoren auf dem Sarcos Inverse Kinematics Datensatz.

Abbildung 7.1: Vorhersagegenauigkeit für die verwendeten Datensätze.

7.4 Durchsatz

Der Durchsatz des FPGA Systems wird mit dem Durchsatz der beiden `Intel` CPUs, sowie dem Durchsatz der `ARM` CPU verglichen. Hierzu wird der in Abschnitt 7.1 beschriebene Datengenerator verwendet, wobei im ersten Experiment die Beispieldimensionen und im

zweiten Experiment die Menge der Basisvektoren variiert werden.
Den Überlegungen am Ende von Abschnitt 6.3.2 folgenden, wird hier lediglich der Durchsatz für die zweite Phase von Projection GP gezeigt. Im Falle des FPGAs wird der maximale theoretische Durchsatz angegeben, wobei hier explizit darauf hingewiesen wird, dass die verwendete Implementierung für 21 Dimensionen und 100 Basisvektoren optimiert wurde. Für die beiden `Intel` Systeme, sowie die `ARM` CPU werden die Dimensionen von 5 bis 750 jeweils in Fünferschritten variiert. Im Falle des FPGAs wurde entsprechend der Markierungen die Dimensionen variiert, da hier die Kompilation von `C`-Code in Hardwarecode vergleichsweise zeitintensiv ist. Bei der Generierung des Hardwarecodes wurde darauf geachtet, dass der durch das HLS Werkzeug geschätzte Verbrauch an FPGA Ressourcen nicht die Ressourcen des verwendeten FPGAs überschreitet. Hierzu waren keine Änderungen an der Implementierung nötig, da sich im Wesentlichen die Menge des verwendeten Blockrams geändert hat.
Abbildung 7.2 zeigt den Durchsatz der vier Systeme bei Variation der Dimension und 100 Basisvektoren. Der Durchsatz ist in Elemente pro Sekunde angegeben und bezeichnet die Anzahl der Beispiele, die die Implementierung pro Sekunde abarbeiten konnte.
Zunächst fällt in Abbildung 7.2a auf, dass Desktop und Laptop einen sehr ähnlichen Verlauf zeigen. Der Durchsatz beginnt bei beinahe 8000 Elemente pro Sekunde auf dem Desktop und etwas über 6000 Elemente pro Sekunde auf dem Laptop. Dann erhöht sich für beide Architekturen der Durchsatz bei steigender Dimensionalität auf 10000 Elemente pro Sekunde auf dem Desktop bzw. etwas über 8000 Elemente pro Sekunde auf dem Laptop. Bei einer Dimension von ca. 100 fällt die Durchsatz auf beiden System auf ca. 1000 Elemente pro Sekunde stark ab, um sich dann danach auf einen etwas höheren Durchsatz als zuvor zu normalisieren. Bei einer Dimension von ca. 200 tritt ein weiterer kleinere Einbruch auf etwas unter 10000 bzw. 8000 Elemente pro Sekunde auf den Systemen auf, der sich auch danach wieder erholt. Ab einer Dimension von ca. 250 nimmt dann der Durchsatz mit steigender Dimension stetig ab und erreicht sein Minimum von etwas über 8000 Elemente pro Sekunde beim Desktop und etwas unter 7000 Elemente pro Sekunde auf dem Laptop bei einer Dimension von 750.
Der Durchsatz der `ARM` CPU und des FPGAs lässt sich aufgrund der Achsenskalierung nur schwierig erkennen, sodass dieser nochmals gesondert in Abbildung 7.2b zu sehen ist. Hier ist zunächst zu erkennen, dass die `ARM` CPU nur einen vergleichsweise geringen Durchsatz von etwas unter 500 Elementen pro Sekunde erreicht, dieser sich aber im Vergleich zu den `Intel` CPUs nur marginal mit steigender Dimension verschlechtert. Das FPGA hingegen schafft einen Durchsatz von ca. 2500 Elementen pro Sekunde und fällt dann mit steigender Dimension rapide ab. Schließlich erreicht das FPGA bei 400-dimensionalen Beispielen den gleichen Durchsatz von etwas unter 500 Elementen pro Sekunde wie die `ARM` CPU und fällt danach sogar noch unter diesen Durchsatz.



(a) Durchsatz der beiden `Intel` CPUs, der `ARM` CPU und des FPGAs für verschiedene Dimensionen bei 100 Basisvektoren.

(b) Durchsatz der `ARM` CPU und des FPGAs für verschiedene Dimensionen bei 100 Basisvektoren.

Abbildung 7.2: Durchsatz der verwendeten Architekturen bei verschiedenen Dimensionen.

Tabelle 7.1 zeigt der Vollständigkeit halber den Durchsatz für die beiden synthetisierten Systeme mit 100 Basisvektoren und für 13 bzw. 21-dimensionale Beispiele. Durch die geringe Taktrate von lediglich 25 MHz erreichen beide Systeme einen Durchsatz von knapp 357 bzw. 340 Elemente pro Sekunde. Betrachtet man zusätzlich die Zeit zur Netzwerkkommunikation, d.h. die Dauer zwischen dem Verschicken eines Beispiels bis zum Erhalt einer Bestätigung, so ergibt sich ein Durchsatz von knapp 100 Elementen pro Sekunde.

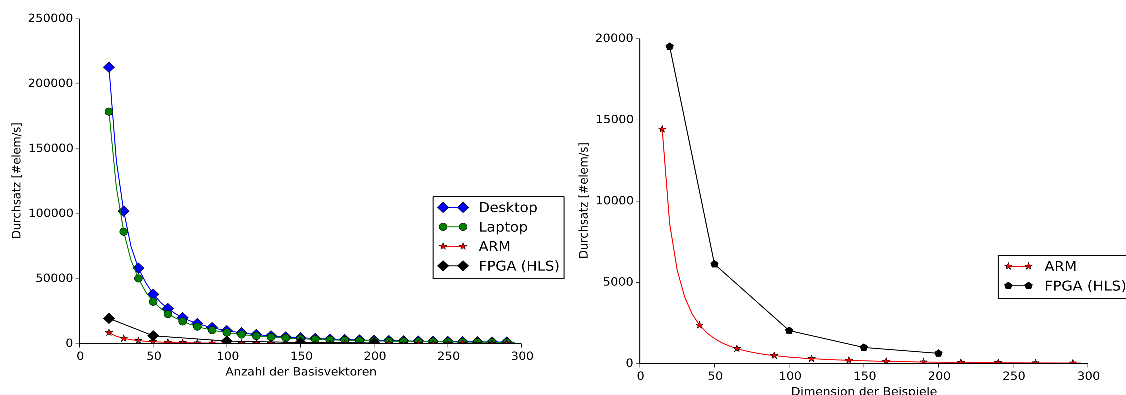
	Durchsatz ohne Netzwerkkommunikation [#elem/s]	Durchsatz mit Netzwerkkommunikation [#elem/s]
PGP,`dim` = 13	357	102,8
PGP,`dim` = 21	340	99,3

Tabelle 7.1: Durchsatz der real synthetisierten Systeme bei 25 MHz Takt für die Projection GP Implementierung mit 100 Basisvektoren.

In Abbildung 7.3 sind die Durchsatzmessungen für Projection GP bei 21 dimensionalen Beispielen und Variation der Anzahl der Basisvektoren zu sehen. Für den Desktop, Laptop und `Raspberry PI` wurden jeweils 20 bis 300 Basisvektoren in Fünferschritten auf dem Desktop variiert. Im Falle des FPGAs ist analog die Anzahl der Basisvektoren entsprechend der Markierungen variiert.Bei der Generierung des Hardwarecodes wurde darauf geachtet, dass der durch das HLS Werkzeug geschätzte Verbrauch an FPGA Ressourcen nicht die Ressourcen des verwendeten FPGAs überschreitet. Hierzu musste bei der Implementierung ab einer Anzahl von 150 Basisvektoren an einigen Stelle das Schleifenabrollen durch Pipelining ersetzt werden.

Zunächst ist in Abbildung 7.3a zu erkennen, dass alle Systeme einen ähnlichen Verlauf zeigen. Das Desktop System beginnt bei einem Durchsatz von ca. 225000 Elementen pro

Sekunde bei 20 Basisvektoren und fällt bereits bei 100 Basisvektoren rapide auf einen Durchsatz von unter 12500 Elementen pro Sekunde ab. Ein ähnliches Bild zeit sich bei dem Laptop System, wobei hier der Durchsatz von etwas über 175000 Elementen pro Sekunde bei 20 Basisvektoren ebenfalls auf einen Durchsatz von etwas von unter 12500 Elementen pro Sekunde abfällt. Das FPGA System, sowie der `Raspberry PI` zeigen auch hier einen deutlich geringeren Durchsatz als die `Intel` Systeme, sodass Abbildung 7.3b für eine bessere Vergleichbarkeit nur diese beiden Systeme zeigt. Hier zeigt sich, dass das FPGA mit einem Durchsatz von knapp 20000 Elementen pro Sekunde zunächst einen um fast 5000 Elemente höheren Durchsatz hat, als das `Raspberry PI` Modell, jedoch um einen Faktor von ca. 10 langsamer ist, als die `Intel` Systeme. Sowohl beim `ARM` System, als auch beim FPGA System fällt der Durchsatz ähnlich wie bei den `Intel` Systemen rapide ab. Hierbei bleibt der Durchsatz des FPGAs jedoch durchgehend über dem des `ARM` Systems. Ab einer Dimension von 200 verhalten sich alle Systeme annähernd gleich mit einem Durchsatz von knapp unter 1000 Elemente pro Sekunde.



(a) Durchsatz der beiden `Intel` CPUs, der `ARM` CPU und des FPGAs bei einer unterschiedlichen Anzahl an Basisvektoren mit 21 Dimensionen.

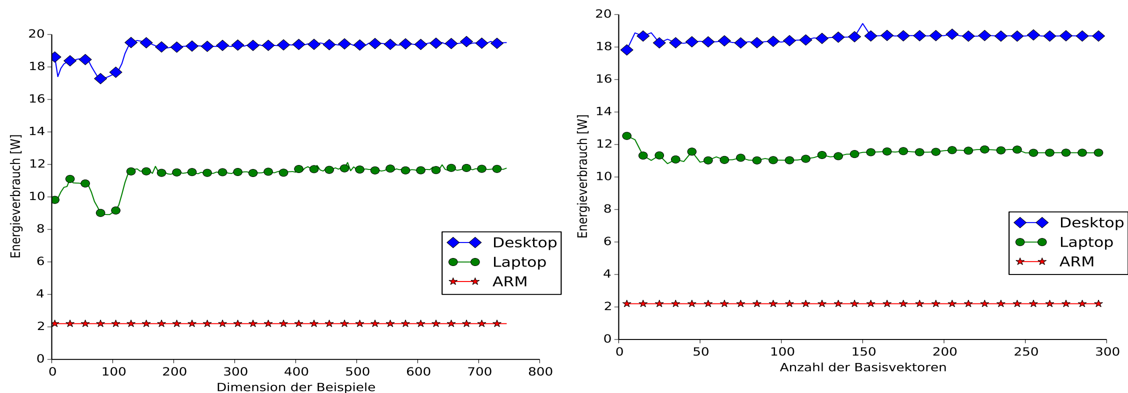
(b) Durchsatz der `ARM` CPU und des FPGAs bei einer unterschiedlichen Anzahl an Basisvektoren mit 21 Dimensionen.

Abbildung 7.3: Durchsatz der verwendeten Architekturen bei einer unterschiedlichen Anzahl an Basisvektoren.

7.5 Energieverbrauch und Energieeffizienz

In diesem Abschnitt wird der Energieverbrauch der `Intel` CPUs, sowie des `Raspberry PI`s der Durchsatzmessungen vorgestellt. Zusätzlich wird der Energieverbrauch der beiden synthetisierten FPGA Systeme angegeben. Wie im vorherigen Abschnitt bereits erläutert wird hierzu zum Einen die Anzahl der Basisvektoren für 21-dimensionale Beispiele, sowie die Dimensionalität der Beispiele für 100 Basisvektoren jeweils variiert.
Zur Messung der Energie kommt wie erwaähnt angedeutet die `Intel Rapl` Bibliothek

zum Einsatz. Um einen fairen Vergleich zu gewährleisten, wird zunächst der Verbrauch von Desktop und Laptop CPU ohne Last, d.h. im Leerlauf gemessen und von den Energiemessungen unter Last abgezogen. Die Desktop CPU verbraucht hier $3,63$ W im Leerlauf, wohingegen die Laptop CPU mit $3,45$ W im Leerlauf auskommt. Auf beiden Systemen lief ein `Ubuntu 14.04` in einer Standardkonfiguration, wobei das Laptop mit Netzteil und ohne Stromspartechniken betrieben wurde.
Als Referenz für die `ARM` CPU werden die in Abschnitt 7.2 diskutierten 2 W angegeben.
Abbildung 7.4a zeigt den Energieverbrauch von Projection GP auf verschiedenen Architekturen bei Änderung der Dimensionalität und 100 Basisvektoren. Hier fällt zunächst der große Unterschied zwischen dem Desktop und dem Laptop auf. Der Desktop PC hat einen Verbrauch von Rund 20W, wohingegen der Laptop mit nur 12W auskommt. Zusätzlich zeigt sich bei einer Dimensionalität von ca. 100 ein Einbruch im Energieverbrauch auf knapp 17W beim Desktop und ca. 9W beim Laptop. Der Energieverbrauch vom `ARM` System ist mit lediglich 2W um einen Faktor 6 bis 10 geringer als für den Desktop bzw. Laptop. Abbildung 7.4b zeigt den Energieverbrauch der drei Systeme bei Variation der Anzahl der Basisvektoren. Hier zeigt sich, dass das Desktop System einen relativ stabilen Energieverbrauch von ca. 18.5 W hat, wohingegen der Laptop relativ konstant 11 W verbraucht. Auch hier zeigt sich, dass der Verbrauch des `Raspberry PI` um einen Faktor von $5,5$ bzw. $9,25$ geringer ist als bei den `Intel` Systemen.



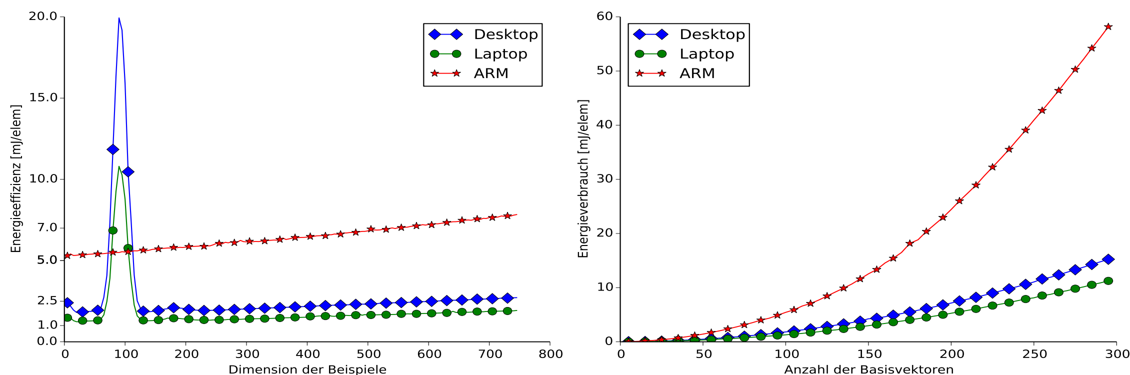
(a) Der Energieverbrauch der einzelnen Systeme für verschiedene Dimensionen mit 100 Basisvektoren.

(b) Der Energieverbrauch der einzelnen Systeme für eine unterschiedliche Anzahl der Basisvektoren bei 21-dimensionalen Beispielen.

Abbildung 7.4: Der Energieverbrauch der einzelnen Systeme.

Mit Hilfe der Energieverbrauches P und des Durchsatzes T lässt sich der Energieverbrauch $\varepsilon = P/T$ pro Element der einzelnen Architekturen bestimmen.
Abbildung 7.5a zeigt die Effizienz der drei Systeme bei Variation der Dimensionalität mit 100 Basisvektoren, wobei ein geringerer Wert eine höhere Effizienz bedeutet. Hier ist zunächst zu erkennen, dass das `Raspberry PI` mit 5mJ pro Element relativ viel Energie pro

Element benötigt. Dieser Wert steigt mit steigender Dimensionalität stetig an, bis er fast 7 mJ pro Element 700 dimensionalen Beispielen erreicht.
Dem Gegenüber sind Desktop und Laptop deutlich energieeffizienter. Es fällt auf, dass Laptop und Desktop mit einem Energieverbrauch von 1.5mJ bis 2mJ fast dieselbe Energieeffizienz haben. Auch hier sinkt die Effizienz mit zunehmender Dimension, wobei sich hier die Kurve weniger kritisch verhält im Vergleich zum ARM. Durch den Einbruch des Durchsatzes um eine Dimensionalität von 100 ist auch hier ein Einbruch der Effizienz auf über 10 MJ bzw. fast 20mJ pro Element für den Laptop bzw. Desktop zu erkennen. Anschließend normalisiert sich die Effizienz wieder und erreicht einen Wert von 2mJ und $2,5$mJ pro Element für den Laptop bzw. den Desktop bei 700-dimensionalen Beispielen.
Abbildung 7.5b zeigt die Effizienz der drei Systeme bei Variation der Anzahl der Basisvektoren und 21-dimensionalen Beispielen. Hier ist zunächst zu erkennen, dass die Kurven für alle drei Systeme einen qualitativ gleichen Verlauf haben. Alle Systeme verbrauchen bei weniger als 50 Basisvektoren sehr wenig Energie, sodass aufgrund der Achsenskalierung dieser nahe Null liegen. Mit steigender Anzahl der Basisvektoren hingegen steigt der Energieverbrauch pro Element rapide an, sodass die ARM CPU schließlich fast 60mJ pro Element bei 300 Basisvektoren verbraucht. Dem gegenüber ist der Laptop und der Desktop deutlich effizienter. Der Desktop verbraucht knapp 10mJ pro Element bei 300 Basisvektoren, wohingegen der Laptop mit knapp $7-8$mJ pro Element auskommt.



(a) Die Effizienz der einzelnen Systeme für verschiedene Dimensionen mit 100 Basisvektoren

(b) Die Effizienz der einzelnen Systeme für eine unterschiedliche Anzahl der Basisvektoren bei 21-dimensionalen Beispielen.

Abbildung 7.5: Die Effizienz der einzelnen Systeme.

Ergänzend zeigt Tabelle 7.2 den Energieverbrauch der synthetisierten Systeme mit 100 Basisvektoren für 13 bzw. 21 dimensionale Beispiele. Beide Systeme haben mit $1,06$ bzw. $1,135$ W den geringsten Energieverbrauch im Test. Die Effizienz ist mit ca. $3-3,6$ mJ besser als die des Raspberry PIs, kann jedoch nicht mit der Effizienz der Intel CPUs mithalten. Das HLS Tool schätzt eine Taktrate von 150 MHz für die beiden Systeme, sodass damit der Durchsatz um einen Faktor Sechs gesteigert werden kann. Bei dem Versuch

diese Taktraten zu synthetisieren (vgl. Abschnitt 6.3.3) wurde hier ein Energieverbrauch im Bereich von ca. $2,3$W ermittelt, was insgesamt zu einem Verbrauch von ca. 1 mJ pro Element führen würden. Damit wäre die Effizienz vergleichbar mit den `Intel` Systemen.

	∅ Energie [W]	Max Energie [W]	Effizienz [mJ]
PGP,`dim` = 13	1,06	1,163	2,98 - 3,26
PGP,`dim` = 21	1,135	1,235	3,3 - 3,6

Tabelle 7.2: Energieverbrauch und Effizienz der real synthetisierten Systeme bei 25 MHz Takt für die Projection GP Implementierung mit 100 Basisvektoren.

7.6 Diskussion

In den vorangegangen Abschnitten wurde die Vorhersagegüte von Projection GP auf zwei Testdatensätzen untersucht und anschließend der Durchsatz und der Energieverbrauch auf dem FPGA mit handelsüblichen CPUs verglichen.
Hier ist zunächst festzuhalten, dass Projection GP gute bis sehr gute Vorhersagen lieferte. Die approximierte Variante von Projection GP ohne die erste Phase hingegen lieferte eher gemischte bis schlechte Ergebnisse, wohingegen ein vollständiger Gauß-Prozess ebenfalls gute Ergebnisse lieferte. Es bleibt festzuhalten, dass Projection GP als Regressionsmethode eine gute Wahl darstellt, wobei ein Vergleich mit anderen Methoden und die Anwendungen auf anderen Datensätzen an dieser Stelle ausblieb, sodass sich hier Raum für weitere Arbeiten ergibt.
Im Falle des Durchsatzes ist zu erkennen, dass die Desktop und die Laptop Implementierung extrem schnell sind. Hier überrascht zunächst die Tatsache, dass der energiesparende Laptop eine vergleichbare Geschwindigkeit zum Desktop System erreicht, was sicherlich der neueren Prozessorgeneration geschuldet ist. Dennoch zeigen beide System bei einer Dimensionalität von ca. 100 bzw. ca. 200 einen starken Einbruch des Durchsatzes, der nicht auf dem FPGA oder dem `ARM` System auftritt. Untersucht man daher den Assemblercode auf diese beiden Systemen genauer, so fällt auf, dass hier viele Vektorisierungsbefehle zu finden sind. Vektorisierungsbfehle sind spezielle Assemblerbefehle, die große Register innerhalb der CPU ausnutzen, um Berechnungen auf Hardwareebene parallel ausführen zu können und so einen Performanzgewinn versprechen (vgl. [HP11] Anhang B). Um den Grad der Vektorisierung in Relation zu setzen, sei an dieser Stelle also der Anteil der Vektorisierung von Projection GP und anderen Programmen gezeigt.
Als Vergleichsprogramme werden zunächst die `gnu-coreutils` und `gzip` als Programme ausgewählt, die auf typischen Linuxsystemen zu finden sind und einen breiten Anwendungsbereich haben. Das `gnu-coreutils` Paket beinhaltet eine Menge verschiedener Programme zur Rechnerverwaltung und Arbeit mit Dateien, wohingegen `gzip` ein optimiertes

Programm zum packen und entpacken von Dateien ist. Als weitere, hochoptimierte, jedoch weniger im Alltag zu findende Softwareprojekte wird die Vektorisierung der Programme im `openssl` Projekt und im linearen Algebra Projekt `LAPACK` betrachtet. Das `openssl` Projekt stellt Implementierungen verschiedener kryptographischer Verfahren für eine sichere Netzwerkkommunikation bereit und ist daher sowohl auf Sicherheit, als auch Geschwindigkeit optimiert. Das `LAPACK` Projekt stellt hochoptimierte Programme und Implementierungen für Standardaufgaben der linearen Algebra, wie die Berechnung von Eigenwerten einer Matrix oder die Berechnung einer inversen Matrix bereit. Zu guter Letzt werden mit `bico`, `kmeans++` und `svmlight` bekannte Implementierungen im Bereich des Data Mining untersucht, um einen fairen Vergleich zu Projection GP zu erlauben. `bico` und `kmeans++` bieten eine Implementierung des k-means Clustering an, wohingegen `svmlight` eine schnelle und optimierte Implementierung der SVM ist.
Alle Programme wurde mit dem `gcc` bzw. `g++` in Version `4.9.3` auf dem Laptop kompiliert, wobei die Optionen `O3, mtune=native` sowie `march=native` gewählt wurden. Diese Optionen schalten die aggressivste, plattformspezifischen Optimierungen in Bezug auf die Ausführungsgeschwindigkeit ein. Im Falle von `LAPACK` und `openssl` werden die mitgeliefert Kompilierskripte und die dortigen Optimierungseinstellungen verwendet, da diese Projekte händisch geschriebenen Assemblercode architekturspezifisch einbinden.

Programm	Assemblerbefehle	Vektorisierungsbefehle	Vektorisierung [%]
`gnu-coreutils`	7447,1	115,8	1,56
`gzip`	17683	659	3,73
`openssl`	154734,39	9009,41	5,82
`lapack`	288896,29	31592,71	10,94
`bico`	14074	1893	13,45
`keans++`	2982	482	16,16
`svmlight`	23380	4956	21,2
`projectiongp`	3783	1377	36,4

Tabelle 7.3: Automatische Vektorisierung für verschiedene Programme. Die Kompilation von `openssl` und `LAPACK` erfolgt mit den mitgelieferten Kompilierskripten. Für die übrigen Programme wurden die Optimierungen `O3, mtune=native` und `march=native` aktiviert. Im Falle von `gnu-coreutils`, `openssl` und `LAPACK` wird die durchschnittliche Anzahl der Assemblerbefehle verwendet, da hier mehrere (Test-)Programme enthalten sind.

In Tabelle 7.3 ist zunächst zu erkennen, dass die `gnu-coreutils` und der `gzip` Quellcode nur zu $1,5$ % bzw. $3,7$ % vektorisiert werden können und damit kaum spezielle Vektorisierungsbefehle angewendet werden. Der Grund hierfür liegt in der Tatsache, dass beide Programme vergleichsweise wenig Berechnungen durchführen, sondern vor allem Dateien

und Zeichenketten manipulieren. Zusätzlich bieten beide Programme eine intensive Fehlerbehandlung, die entsprechende Eingaben des Benutzer überprüfen und gegebenenfalls Fehlermeldungen an den Benutzer zurückgeben. Diese steigern die Codegröße, wodurch sich das Verhältnis von Ein- und Ausgabeverwaltung zu der eigentlichen Funktionalität verschiebt.
Die händische Vektorisierung im `openssl` Projekt, sowie im `LAPACK` Projekt ist mit knapp 6 % und 10 % deutlich höher. Grund hierfür liegt sicherlich in den berechnungsintensiven Problemen, die in beiden Projekten gelöst werden. Dennoch ist auch hier anzumerken, dass der Grad der Vektorisierung durch Fehlerbehandlungen und Behandlung der Nutzer Ein- und Ausgabe nur eingeschränkt ist.
Die im Bereich des Data Mining entstandenen Implementierung `bico`, `kmeans++` und `svm-light` zeigen mit $13\% - 21\%$ einen hohen Vektorisierungsgrad. Dies liegt neben der berechnungsintensiven Natur der gelösten Probleme auch an einem geringeren Verwaltungsaufwand für Nutzereingaben und Fehlerbehandlung. Diese Programme entstanden vorallem zu Forschungszwecken und sind nicht notwendigerweise benutzerfreundlich im Sinne einer intensiven Fehlerbehandlung.
Projection GP zeigt mit 36 % den höchsten Grad der Vektorisierung, was zunächst den allgemein hohen Durchsatz erklärt. Der Grund für diesen hohen Grad der Vektorisierung liegt vor allem daran, dass die Implementierung für die High Level Synthese optimiert wurde. Damit sind Parameter statisch zur Übersetzungszeit bekannt und Speicherstrukturen bewusst einfach gehalten, sodass moderne Compiler eine Vielzahl von Optimierungen anwenden können. Tatsächlich schient sich hiermit auch den Einbruch des Durchsatzes bei 100 bzw. 200 dimensionalen Beispielen zu erklären. Schaltet man die Verwendung von Vektorisierungsbefehlen bei der Übersetzung aus, so ähnelt der resultierende Durchsatz über alle Dimensionen dem in Abbildung 7.2a gezeigten Durchsatz von 1000 Elemente pro Sekunde. Ein Fehler im Übersetzungsprogramm scheint hier zunächst unwahrscheinlich, da sich ein ähnliches Verhalten auch mit dem `gcc` in Version `4.8.4` und dem `clang` Compiler in Version `3.6.0` zeigte. Damit scheinen diese Performanzeinbrüche an den Vektorisierungsbfehlen der `x86` Architektur der `Intel` CPUs zu liegen, wobei hier eine mögliche Erklärung in den Registerbreiten liegt: Bei bestimmten Dimensionen um 100 und 200 lassen sich die resultierenden Beobachtungen nicht passend in die Vektorregister der CPU laden, sodass die CPU hier mehrere Speicherzugriffe braucht. Zusätzlich muss die CPU die Register bei unpassender Breite mit passenden Nullen auffüllen, sodass neben dem Warten auf den Speicher ein Mehraufwand durch das Auffüllen der Nullen entsteht, was effektiv den Effekt der Vektorisierung negiert.
Diese Theorie wird von der Tatsache unterstützt, dass bei Dimensionen von 100 bzw. 200 der Energieverbrauch sinkt, was durch das Warten der CPU auf den Speicher erklärt werden könnte.
Dennoch bietet diese Theorie keine abschließende Erklärung, da der beschrieben Effekt nur

bei Dimensionen um 100 bzw. 200 auftritt. Die Registerbreiten sind bei den `Intel` CPUs als 2er Potenzen angelegt, d.h. Register sind je nach Modell und Grad der Vektorisierung $32, 64, 128, 256$ oder 512 Bit breit. Damit müsste sich eine optimale Performanz bei Dimensionsgrößen zeigen, die zu diesen Registerbreiten passen. Im Umkehrschluss bedeutet dies, dass bei allen unpassenden Dimensionsgrößen ein Performanzeinbruch verzeichnet werden müsste, sodass dieser deutlich öfter als nur bei den Dimensionen um 100 bzw. 200 auftreten müsste. Des Weiteren betrachtet diese Erklärung weder Cachingeffekte, noch Mechanismen wie Branch-Prediction, sodass die Gründe für diesen Effekt nicht abschließend geklärt sind und weitere Arbeiten zu diesem Thema folgen müssen.

Die FPGA Implementierung zeigt sich insgesamt gemischt. Zunächst ist festzuhalten, dass der Durchsatz der Implementierung nicht mit dem Durchsatz auf dem Laptop bzw. dem Desktop zu vergleichen ist. Die Gründe liegen hier neben der bereits diskutierten starken Optimierung des Quellcodes in der Tatsache, dass beide `Intel` Systeme mit 2.1 GHz im Falle des Laptops bzw. 3.4 GHz im Falle der Desktopmaschine um einen Faktor von fast 14 bis 22 schneller getaktet sind als das FPGA.
Durch Ausnutzung von Parallelität beim Berechnen der Kernfunktionswerte, sowie beim Iterieren über die Matrixstrukturen ist es dem FPGA jedoch möglich einen höheren Durchsatz als der mit 900 MHz getakteten `ARM` Prozessor zu erreichen.
Der geringe Durchsatz beim Messen der Netzwerkkommunikation von ca. 100 Elementen pro Sekunde überrascht nicht, da hier die Paketumlaufzeiten mitgemessen werden. Da die Messung sowohl für 13-dimensionale Beispiele, als auch für 21-dimensionale Beispiele fast identisch ist, ist anzunehmen, dass die Netzwerkkommunikation hier der Flaschenhals bildet. Es zeigt sich also, dass die synthetisierten Hardwareblöcke trotz des geringen Durchsatzes von Runde 350 Elementen pro Sekunde fast drei mal schneller als die Netzwerkkommunikation sind und damit zunächst ausreichend schnell sind. An dieser Stelle bleibt die Frage offen, ob diese Realgeschwindigkeit für eine Anwendung ausreichend ist, da sie nur auf dem Hintergrund einer konkreten Anwendung diskutiert werden kann. Bei der Konfiguration des `MicroBlaze` Prozessors, sowie des `lwIP` Protokollstapels wurde der Fokus auf eine möglichst platz- und damit energiesparende Konfiguration gelegt. Schreibt die konkrete Anwendung einen höheren Netzwerkdurchsatz vor, so können diese Konfigurationen entsprechend angepasst werden.
Das FPGA System ist das energiesparendste System im Test und konnte im Vergleich zum `Raspberry PI` den Energieverbrauch nochmals halbieren. Damit eignet sich diese FPGA Implementierung gut für den Einsatz in eingebetteten Systemen.
Tatsächlich ist die Anwendbarkeit des Laptops und des Desktops in eingebetteten Systemen aufgrund des hohen Energieverbrauches fragwürdig. Hier ist zu beachten, dass neben der eigentlichen CPU, auch die angeschlossene Mainboardperipherie wie die Festplatte oder der Hauptspeicher Energie verbrauchen, sodass der reale Energieverbrauch dieser

beiden System höher liegt. Der `Raspberry PI` und das FPGA hingegen bieten bereits jeweils ein geschlossenes System, welches mit $1 - 2$W Energieverbrauch fast $10 - 20$ mal energiesparender als die `Intel` Systeme ist.
Betrachtet man die Effizienz der Plattformen genauer, so zeigt sich zunächst, dass die `ARM` CPU eine vergleichsweise schlechte Effizienz liefert. Der `Raspberry PI` ist zwar mit 2 W Energieverbrauch ein sehr sparsames System, doch hat es auch einen geringen Durchsatz, sodass die Effizienz vergleichsweise schlecht ist.
Das FPGA hingegen verbraucht mit nur 1W noch weniger Energie, sodass sich trotz eines geringeren Realdurchsatzes eine bessere Effizienz im Vergleich zum `Raspberry PI` ergibt. Diese Effizienz kann wie beschrieben nochmals gesteigert werden, wenn die Synthese höhere Taktraten durch geeignetes Floorplanning ermöglicht.
Alles in Allem zeigt sich, dass FPGAs eine sehr energiesparende Systemarchitektur anbieten und bei geeigneter Programmierung einen passenden Durchsatz liefern. Durch Verwendung der High Level Synthese kann der Hardwareblock zunächst in einer Hochsprache entwickelt werden, sodass die vorliegenden Implementierung von Projection GP die gleiche Vorhersagegüte wie eine Hochsprachenimplementierung hat.

8 | Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurden FPGAs als mögliche Systemarchitektur für maschinelle Lernverfahren in eingebetteten Systemen untersucht. Hierzu wurde zunächst ein Entwurfsvorgehen für technische System vorgestellt und die Funktionsweise von FPGAs erläutert. Daraufhin wurden Gauß-Prozesse beispielhaft als maschinelles Lernverfahren betrachtet, um diese anschließend auf einem FPGA zu implementieren.
Es zeigte sich, dass eine Implementierung für ein FPGA dem Entwurf eines Gesamtsystems nahe kommt. Daher wurde zunächst die Laufzeitumgebung auf dem FPGA entworfen, sodass Daten über TCP/IP zwischen dem FPGA und einem Host-PC ausgetauscht werden können und der maschinelle Lernalgorithmus passend ausgeführt wird.
Die hier vorgestellte FPGA Systemarchitektur ist durch die Verwendung der `AXI` Schnittstelle modular, wodurch sich der maschinelle Lernalgorithmus leicht austauschen lässt. Damit kann dieses Systemarchitektur als Grundlage für weitere Arbeiten genutzt werden. Zusätzlich wurde das Gesamtsystem möglichst ressourcenschonend implementiert.
Der Projection GP Algorithmus wurde als ein spezieller Gauß-Prozess mittels High Level Synthese auf dem FPGA implementiert, wobei hier einige Geschwindigkeitsoptimierungen diskutiert und ihr Einfluss kritisch evaluiert wurde.
Zunächst ist festzuhalten, dass der implementierte Gauß-Prozess durch die Verwendung von HLS die gleiche Vorhersagegüte wie übliche Implementierungen auf Standardhardware liefert, da der zugrundeliegende Quellcode semantisch äquivalent ist. Hier mussten lediglich die Struktur des Quellcodes und einige Präprozessordirektiven geändert werden. Es zeigt sich, dass das entworfene FPGA System um einen Faktor von fast 20 energiesparender als übliche Desktophardware ist, jedoch aufgrund der geringeren Taktrate um einen Faktor von bis zu 34 langsamer ist. Im Vergleich zu Hardware aus dem Bereich der eingebetteten Systeme ist die gezeigte Implementierung um einen Faktor um 2 energiesparender und erreicht einen ähnlichen Durchsatz.
Die Möglichkeiten zum Energiesparen ergeben sich vor Allem durch detaillierte Analyse des Speicherbedarfs des Algorithmus, sodass das umliegende System extrem energie- und platzsparend implementiert werden konnte. Hier zeigt sich, dass eine ganzheitlicher Systementwurf, der sowohl den maschinellen Lernalgorithmus, als auch die Ausführungsplattform berücksichtigt, tiefgreifendere Optimierungen ermöglicht.

Zu guter Letzt sei angemerkt, dass die hier vorgestellte Implementierung prinzipiell realzeitfähig ist. Für den maschinelle Lernalgorithmus lassen sich exakt die Taktrate und Taktverzögerung angeben, sodass an dieser Stelle Deadlines abgeleitet werden können. Die Datenkommunikation hingegen ist derzeit nicht realzeitfähig, da das TCP Protokoll dies nicht unterstützt. Hier fehlte für die vorliegende Arbeit eine geeignete Realzeitanwendung, um die Realzeitfähigkeit eingehender zu untersuchen, sodass ich auch hier Raum für weitere Arbeiten ergibt.
Die Synthese von Hardwarecode stellt sich als ein iterativer Arbeitsprozess dar, der aufgrund der lange Synthesezeit mehrere Tage bis Wochen in Anspruch nehmen kann. Um diesen zu beschleunigen, wurden im Laufe der vorliegenden Arbeit ein Großteil der vorhanden Syntheseoptimierungen im Synthesetool ausgeschaltet.
Eine theoretische Analyse des Durchsatzes zeigt, dass dieser durch Erhöhung der Taktrate nochmals deutlich gesteigert werden kann, wodurch sich der Unterschied zwischen FPGAs und handelsüblicher Desktophardware weniger stark darstellt. Zur Erhöhung der Taktrate müssen die Optimierungen im Synthesetool entsprechend konfiguriert werden. Die Auswahl der optimalen Konfiguration zeigt sich jedoch als herausfordernd, sodass auch hier ein Bedarf an weiteren Arbeiten, in denen Techniken wie Floorplanning oder die partielle Rekonfiguration untersucht werden besteht.
Aus Sicht des maschinellen Lernens zeigt sich, dass Projection GP durchaus gute bis sehr gute Vorhersageergebnisse liefert. Dennoch muss für eine erfolgreiche Anwendung vor allem die Anzahl der Basisvektoren gewählt werden. Hier steht aktuell keine Methode zur automatisierten Wahl der Anzahl der Basisvektoren bereit, sodass auch hier weitere Forschung erfolgen kann.
Zu guter Letzt berücksichtigt Projection GP keinerlei Concept-Drift bzw. Änderungen im Datenstrom, was auch hier weitere Arbeiten ermöglicht.

A | Eigenschaften der Normalverteilung

In diesem Abschnitt wird die Randverteilungseigenschaft und die konditionalisierte Form der Normalverteilung hergeleitet. Dazu sollen zunächst einige Eigenschaften aus der Lineare Algebra festgehalten werden. $I_n \in \mathbb{R}^{n \times n}$ bezeichne dazu im Folgenden die $n \times n$ Einheitsmatrix.

Theorem A.0.1 (Matrix-Inversions Lemma) Seien $A \in \mathbb{R}^{n \times n}$ eine invertierbare $n \times n$ Matrix, $B \in \mathbb{R}^{m \times m}$ eine invertierbare $m \times m$ Matrix. Seien ferner $C, D \in \mathbb{R}^{n \times m}$ $n \times m$ Matrizen, dann gilt folgende Formel:

$$(A + CBD)^{-1} = A^{-1} - A^{-1}C(B^{-1} + DA^{-1}C)^{-1}DA^{-1}$$

Beweis (Matrix-Inversion Lemma)
Zu zeigen: $(A + CBD)[A^{-1} - A^{-1}C(B^{-1} + DA^{-1}C)^{-1}DA^{-1}] = I_n$

$$\begin{aligned}
& (A + CBD)[A^{-1} - A^{-1}C(B^{-1} + DA^{-1}C)^{-1}DA^{-1}] \\
= \; & (A + CBD)A^{-1} - A(CBD)A^{-1}C(B^{-1} + DA^{-1}C)^{-1}DA^{-1} \\
= \; & I_n + CBDA^{-1} - (C + CBDA^{-1}C)(B^{-1} + DA^{-1}C)^{-1}DA^{-1} \\
= \; & I_n + CBDA^{-1} - CB(B^{-1} + DA^{-1}C)(B^{-1} + DA^{-1}C)^{-1}DA^{-1} \\
= \; & I_n + CBDA^{-1} - CBDA^{-1} = I_n
\end{aligned}$$

Theorem A.0.2 (Inverse einer symmetrischen Matrix) Sei $A \in \mathbb{R}^{n \times n}$ eine symmetrische, invertierbare Blockmatrix

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{12}^T & A_{22} \end{bmatrix}$$

mit $A_{11} \in \mathbb{R}^{p \times p}, A_{22} \in \mathbb{R}^{q \times q}, A_{21} = A_{12}^T \in \mathbb{R}^{p \times q}$ wobei $n = p + q$ gilt. Bezeichne weiterhin $B = A^{-1}$ die Inverse von A mit

$$B = A^{-1} = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} B_{11} & B_{12} \\ B_{12}^T & B_{22} \end{bmatrix}$$

wobei $B_{11} \in \mathbb{R}^{p \times p}, B_{22} \in \mathbb{R}^{q \times q}, B_{21} = B_{12}^T \in \mathbb{R}^{p \times q}$ gilt. Dann lassen sich die vier Blöcke von B wie folgt berechnen:

$$\begin{aligned}
B_{11} &= (A_{11} - A_{12}A_{22}^{-1}A_{12}^T)^{-1} = A_{11}^{-1} + A_{11}^{-1}A_{12}(A_{22} - A_{12}^TA_{11}^{-1}A_{12})^{-1}A_{12}^TA_{11}^{-1} \\
B_{22} &= (A_{22} - A_{12}^TA_{11}^{-1}A_{12})^{-1} = A_{22}^{-1} + A_{22}^{-1}A_{12}^T(A_{11} - A_{12}A_{22}^{-1}A_{12}^T)^{-1}A_{12}^TA_{22}^{-1} \\
B_{12}^T &= -A_{22}^TA_{12}^T(A_{11} - A_{12}A_{22}^{-1}A_{12}^T)^{-1} \\
B_{12}^T &= -A_{11}^TA_{12}(A_{22} - A_{12}^TA_{11}^{-1}A_{12}^T)^{-1}
\end{aligned}$$

Beweis (Inverse einer symmetrischen Matrix)

Zu zeigen:

$$\begin{aligned} I_n &= AA^{-1} = AB = \begin{bmatrix} A_{11} & A_{12} \\ A_{12}^T & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{12}^T & B_{22} \end{bmatrix} \\ &= \begin{bmatrix} A_{11}B_{11} + A_{12}B_{12}^T & A_{11}B_{12} + A_{12}B_{22} \\ A_{12}^T B_{11} + A_{22}B_{12}^T & A_{12}^T B_{12} + A_{22}B_{22} \end{bmatrix} = \begin{bmatrix} I_p & 0 \\ 0 & I_q \end{bmatrix} \end{aligned}$$

Es gilt:

$$\begin{aligned} A_{11}B_{11} + A_{12}B_{12}^T = I_p &\Rightarrow B_{11} = A_{11}^{-1} - A_{11}^{-1}A_{12}B_{12}^T \\ A_{12}^T B_{11} + A_{22}B_{12}^T = 0 &\Rightarrow B_{12}^T = -A_{22}^{-1}A_{12}^T B_{11} \end{aligned}$$

Durch Einsetzen folgt:

$$\begin{aligned} B_{11} &= A_{11}^{-1} + A_{11}^{-1}A_{12}A_{22}^{-1}A_{12}^T B_{11} \\ \Rightarrow A_{11}^{-1} &= (I_p - A_{11}^{-1}A_{12}A_{22}^{-1}A_{12}^T)B_{11} \\ \Rightarrow I_p &= (A_{11} - A_{12}A_{22}^{-1}A_{12}^T)B_{11} \\ \Rightarrow B_{11} &= (A_{11} - A_{12}A_{22}^{-1}A_{12}^T)^{-1} \end{aligned}$$

Durch Anwendung von Theorem A.0.1 ergibt sich weiterhin:

$$B_{11} = (A_{11} - A_{12}A_{22}^{-1}A_{12}^T)^{-1} = A_{11}^{-1} + A_{11}^{-1}A_{12}(A_{22} - A_{12}^T A_{11}^{-1}A_{12})^{-1}A_{12}^T A_{11}^{-1}$$

Analog lassen sich die gesuchten Ausdrücke für B_{22} und B_{12}^T herleiten.

Theorem A.0.3 (Determinante einer symmetrischen Matrix) Sei $A \in \mathbb{R}^{n\times n}$ eine symmetrische Blockmatrix mit

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{12}^T & A_{22} \end{bmatrix}$$

Dann gilt für die Determinante von A:

$$|A| = |A_{22}||A_{11} - A_{12}A_{22}^{-1}A_{12}^T| = |A_{11}||A_{22} - A_{12}^T A_{11}^{-1}A_{12}|$$

Beweis (Determinante einer symmetrischen Matrix) Zunächst wird die Matrix A umgeformt:

$$\begin{aligned} A &= \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{12}^T & A_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & 0 \\ A_{12}^T & I_p \end{bmatrix} \cdot \begin{bmatrix} I_p & A_{11}^{-1}A_{12} \\ 0 & A_{22} - A_{12}^T A_{11}^T A_{12} \end{bmatrix} \\ &= \begin{bmatrix} I_p & A_{12} \\ 0 & A_{22} \end{bmatrix} \cdot \begin{bmatrix} A_{11} - A_{12}A_{22}^{-1}A_{12}^T & 0 \\ A_{22}^{-1}A_{21} & 0 \end{bmatrix} \end{aligned}$$

Nach Determinantenproduktsatz gilt: $|BC| = |B||C|$ für zwei Matrizen $B, C \in \mathbb{R}^{n\times n}$ [PP+12]. Somit folgt, dass

$$|A| = \left|\begin{bmatrix} I_p & A_{12} \\ 0 & A_{22} \end{bmatrix}\right| \cdot \left|\begin{bmatrix} A_{11} - A_{12}A_{22}^{-1}A_{12}^T & 0 \\ A_{22}^{-1}A_{21} & 0 \end{bmatrix}\right|$$

Ferner gilt für 2×2 Blockmatrizen[PP+12]

$$\left|\begin{bmatrix} E & 0 \\ F & G \end{bmatrix}\right| = \left|\begin{bmatrix} E & F \\ 0 & G \end{bmatrix}\right| = |E||G|$$

worraus sich schließlich der Beweis ergibt.

Mit Hilfe dieser Theoreme lässt sich nun folgende Eigenschaft der Normalverteilung herleiten

Satz A.0.1 (Randverteilung der Normalverteilung)
Sei $\vec{x} = \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \end{bmatrix} \sim \mathcal{N}\left(\vec{m} = \begin{bmatrix} \vec{m}_1 \\ \vec{m}_2 \end{bmatrix}, \Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}\right)$ normalverteilt, dann sind die Randverteilungen für $\vec{x}_1$ und $\vec{x}_2$ ebenfalls Normalverteilungen, d.h.

$$\vec{x}_1 \sim \mathcal{N}(\vec{m}_1, \Sigma_{11}) \text{ bzw. } \mathcal{N}(\vec{x}_1, \vec{m}_1, \Sigma_{11})$$

$$\vec{x}_2 \sim \mathcal{N}(\vec{m}_2, \Sigma_{22}) \text{ bzw. } \mathcal{N}(\vec{x}_2, \vec{m}_2, \Sigma_{22})$$

Beweis (Randverteilungseigenschaft der Normalverteilung)
Der Beweis wird in Anlehnung an [VM64] Abschnitt 9.2 ff geführt. Es ist zu zeigen:

$$p(\vec{x}_1|\vec{m}_1, \Sigma_{11}) = \int p(\vec{x}|\vec{m}, \Sigma) d\vec{x}_2 = (2\pi)^{-\frac{p}{2}} |\Sigma_{11}|^{-\frac{1}{2}} exp\left(-\frac{1}{2}(\vec{x_1} - \vec{m_1})^T \Sigma_{11}^{-1} (\vec{x_1} - \vec{m_1})\right)$$

Für die Dichtefunktion gilt:

$$\begin{aligned} p(\vec{x}|\vec{m}, \Sigma) &= (2\pi)^{-\frac{N}{2}} |\Sigma|^{-\frac{1}{2}} exp\left(-\frac{1}{2}(\vec{x} - \vec{m})^T \Sigma^{-1} (\vec{x} - \vec{m})\right) \\ &= (2\pi)^{-\frac{N}{2}} |\Sigma|^{-\frac{1}{2}} exp\left(-\frac{1}{2} Q(\vec{x}_1, \vec{x}_2)\right) \end{aligned}$$

Sei hier

$$\Sigma^{-1} = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}^{-1} = \begin{bmatrix} \Sigma^{11} & \Sigma^{12} \\ \Sigma^{21} & \Sigma^{22} \end{bmatrix} = \begin{bmatrix} \Sigma^{11} & \Sigma^{12} \\ \Sigma^{12T} & \Sigma^{22} \end{bmatrix}$$

Dann lässt sich der Exponent $Q(\vec{x}_1, \vec{x}_2)$ wie folgt umschreiben:

$$\begin{aligned} Q(\vec{x}_1, \vec{x}_2) &= (\vec{x}-\vec{m})^T \Sigma^{-1} (\vec{x}-\vec{m}) = \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \end{bmatrix}^T \begin{bmatrix} \Sigma^{11} & \Sigma^{12} \\ \Sigma^{21} & \Sigma^{22} \end{bmatrix} \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \end{bmatrix} \\ &= (\vec{x}_1-\vec{m}_1)^T \Sigma^{11} (\vec{x}_1-\vec{m}_1) + 2(\vec{x}_1-\vec{m}_1)^T \\ &\quad \cdot \Sigma^{12} (\vec{x}_2-\vec{m}_2) + (\vec{x}_2-\vec{m}_2)^T \Sigma^{22} (\vec{x}_2-\vec{m}_2) \end{aligned}$$

Aus Theorem A.0.3 folgt:

$$\begin{aligned} \Sigma^{11} &= (\Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{12}^T)^{-1} = \Sigma_{11}^{-1} + \Sigma_{11}^{-1}\Sigma_{12}(\Sigma_{22} - \Sigma_{12}^T\Sigma_{11}^{-1}\Sigma_{12})^{-1}\Sigma_{12}^T\Sigma_{11}^{-1} \\ \Sigma^{22} &= (\Sigma_{22} - \Sigma_{12}^T\Sigma_{11}^{-1}\Sigma_{12})^{-1} = \Sigma_{22}^{-1} + \Sigma_{22}^{-1}\Sigma_{12}^T(\Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{12}^T)^{-1}\Sigma_{12}\Sigma_{22}^{-1} \\ \Sigma^{12} &= \Sigma^{21^T} = -\Sigma_{11}^{-1}\Sigma_{12}(\Sigma_{22} - \Sigma_{12}^T\Sigma_{11}^{-1}\Sigma_{12})^{-1} \end{aligned}$$

Durch Einsetzen in $Q(\vec{x}_1, \vec{x}_2)$ ergibt sich:

$$\begin{aligned} Q(\vec{x}_1, \vec{x}_2) &= (\vec{x}_1-\vec{m}_1)^T[\Sigma_{11}^{-1} + \Sigma_{11}^{-1}\Sigma_{12}(\Sigma_{22} - \Sigma_{12}^T\Sigma_{11}^{-1}\Sigma_{12})^{-1}\Sigma_{12}^T\Sigma_{11}^{-1}](\vec{x}_1-\vec{m}_1) \\ &\quad -2(\vec{x}_1-\vec{m}_1)^T[\Sigma_{11}^{-1}\Sigma_{12}(\Sigma_{22} - \Sigma_{12}^T\Sigma_{11}^{-1}\Sigma_{12})^{-1}](\vec{x}_2-\vec{m}_2) \\ &\quad +(\vec{x}_2-\vec{m}_2)^T[(\Sigma_{22} - \Sigma_{12}^T\Sigma_{11}^{-1}\Sigma_{12})^{-1})^{-1}](\vec{x}_2-\vec{m}_2) \\ &= (\vec{x}_1-\vec{m}_1)^T\Sigma_{11}^{-1}(\vec{x}_1-\vec{m}_1) \\ &\quad +(\vec{x}_1-\vec{m}_1)^T[\Sigma_{11}^{-1}\Sigma_{12}(\Sigma_{22} - \Sigma_{12}^T\Sigma_{11}^{-1}\Sigma_{12})^{-1}\Sigma_{12}^T\Sigma_{11}^{-1}](\vec{x}_1-\vec{m}_1) \\ &\quad -2(\vec{x}_1-\vec{m}_1)^T[\Sigma_{11}^{-1}\Sigma_{12}(\Sigma_{22} - \Sigma_{12}^T\Sigma_{11}\Sigma_{12})^{-1}](\vec{x}_2-\vec{m}_2) \\ &\quad +(\vec{x}_2-\vec{m}_2^T[(\Sigma_{22} - \Sigma_{12}^T\Sigma_{11}^{-1}\Sigma_{12})^{-1}](\vec{x}_2-\vec{m}_2) \\ &= (\vec{x}_1-\vec{m}_1)^T\Sigma_{11}^{-1}(\vec{x}_1-\vec{m}_1) \\ &\quad +[(\vec{x}_2-\vec{m}_2) - \Sigma_{12}^T\Sigma_{11}^{-1}(\vec{x}_1-\vec{m}_1)]^T(\Sigma_{22} - \Sigma_{12}^T\Sigma_{11}^{-1}\Sigma_{12})^{-1} \\ &\quad \cdot[(\vec{x}_2-\vec{m}_2) - \Sigma_{12}^T\Sigma_{11}^{-1}(\vec{x}_1-\vec{m}_1)] \end{aligned}$$

Sei $Q(\vec{x}_1, \vec{x}_2) = Q_1 + Q_2$ mit

$$\begin{aligned} Q_1 &= (\vec{x}_1-\vec{m}_1)^T\Sigma_{11}^{-1}(\vec{x}_1-\vec{m}_1) \\ Q_2 &= [(\vec{x}_2-\vec{m}_2) - \Sigma_{12}^T\Sigma_{11}^{-1}(\vec{x}_1-\vec{m}_1)]^T \\ &\quad \cdot(\Sigma_{22} - \Sigma_{12}^T\Sigma_{11}^{-1}\Sigma_{12})^{-1}[(\vec{x}_2-\vec{m}_2) - \Sigma_{12}^T\Sigma_{11}^{-1}(\vec{x}_1-\vec{m}_1)] \\ &= (\vec{x}_2-\vec{b})^T A^{-1}(\vec{x}_2-\vec{b}) \end{aligned}$$

und

$$\begin{aligned} \vec{b} &= \vec{m}_2 + \Sigma_{12}^T\Sigma_{11}^{-1}(\vec{x}_1-\vec{m}_1) \\ A &= \Sigma_{22} - \Sigma_{12}^T\Sigma_{11}^{-1}\Sigma_{12} \end{aligned}$$

Nun lässt sich die Dichtefunktion $p(\vec{x}|\vec{m},\Sigma)$ mit Hilfe von Theorem A.0.3 umformen:

$$\begin{aligned}
p(\vec{x}|\vec{m},\Sigma) &= (2\pi)^{-\frac{N}{2}}|\Sigma|^{-\frac{1}{2}} exp\left(-\frac{1}{2}Q(x_1,x_2)\right) \\
&= (2\pi)^{-\frac{N}{2}}|\Sigma_{11}|^{-\frac{1}{2}}|\Sigma_{22}-\Sigma_{12}^T\Sigma_{11}^{-1}\Sigma_{12}|^{-\frac{1}{2}} exp\left(-\frac{1}{2}Q(x_1,x_2)\right) \\
&= (2\pi)^{-\frac{p}{2}}|\Sigma_{11}|^{-\frac{1}{2}} exp\left(-\frac{1}{2}(\vec{x}_1-\vec{m}_1))^T\Sigma_{11}^{-1}(\vec{x}_1-\vec{m}_1)^T\right) \\
&\quad \cdot (2\pi)^{-\frac{q}{2}}|A|^{-\frac{1}{2}} exp\left(-\frac{1}{2}(\vec{x}_2-\vec{b}))^T A^{-1}(\vec{x}_2-\vec{b})^T\right) \\
&= \mathcal{N}(\vec{x}_1,\vec{m}_1,\Sigma_{11})\mathcal{N}(\vec{x}_2,\vec{b},A)
\end{aligned}$$

Durch Einsetzen in die Integralform ergibt sich schließlich:

$$p(\vec{x}_1|\vec{m}_1,\Sigma_{11}) = \int \mathcal{N}(\vec{x}_1,\vec{m}_1,\Sigma_{11})\mathcal{N}(\vec{x}_2,\vec{b},A)$$

Der erste Teil der Formel ist unabhängig von $\vec{x}_2$ sodass dieser konstant im Integral ist:

$$p(\vec{x}_1|\vec{m}_1,\Sigma_{11}) = \mathcal{N}(\vec{x}_1,\vec{m}_1,\Sigma_{11}) \int \mathcal{N}(\vec{x}_2,\vec{b},A)d\vec{x}_2$$

Dieses Integral hängt nur noch von $\vec{x}_2$ ab und beschreibt die Normalverteilung mit Mittelwertvektor $\vec{b}$ und Kovarianzmatrix A. Aus der Definition der Dichtefunktion folgt, dass dieses Integral immer 1 ergibt. Damit folgt:

$$p(\vec{x}_1|\vec{m}_1,\Sigma_{11}) = \mathcal{N}(\vec{x}_1,\vec{m}_1,\Sigma_{11}) = (2\pi)^{-\frac{p}{2}}|\Sigma_{11}|^{-\frac{1}{2}} exp\left(-\frac{1}{2}(\vec{x_1}-\vec{m_1})^T\Sigma_{11}^{-1}(\vec{x_1}-\vec{m_1})\right)$$

Satz A.0.2 (Konditionalisierung der Normalverteilung)
Sei $\vec{x} = \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \end{bmatrix} \sim \mathcal{N}\left(\vec{m} = \begin{bmatrix} \vec{m}_1 \\ \vec{m}_2 \end{bmatrix}, \Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}\right)$ normalverteilt, dann ist die bzgl. $\vec{x}_1$ konditionalisierte Verteilung $\vec{x}_2|\vec{x}_1$ von $\vec{x}_2$ eine Normalverteilung mit:

$$\vec{x}_2|\vec{x}_1 \sim \mathcal{N}(\vec{m}_2 + \Sigma_{21}^T \Sigma_{11}^{-1}(\vec{x}_1 - \vec{m}_1), \Sigma_{22} - \Sigma_{12}^T \Sigma_{11}^{-1} \Sigma_{12})$$

Beweis (Konditionalisierung der Normalverteilung) Zu zeigen:

$$p(\vec{x}_2|\vec{x}_1, \vec{m}, \Sigma) = \frac{p(\vec{x}|\vec{m}, \Sigma)}{p(\vec{x}_1|\vec{m}, \Sigma)} = \mathcal{N}(\vec{m}_2 + \Sigma_{21}\Sigma_{11}^{-1}(\vec{x}_1 - \vec{m}_1), \Sigma_{22} - \Sigma_{21}^T \Sigma_{11}^{-1} \Sigma_{21})$$

Ausgehend von Beweis A gilt:

$$p(\vec{x}|\vec{m}\Sigma) = \mathcal{N}(\vec{x}_1, \vec{m}_1, \Sigma_{11})\mathcal{N}(\vec{x}_2, \vec{b}, A)$$

Daraus folgt:

$$p(\vec{x}_1|\vec{x}_2, \vec{m}, \Sigma) = \frac{\mathcal{N}(\vec{x}_1, \vec{m}_1, \Sigma_{11})\mathcal{N}(\vec{x}_2, \vec{b}, A)}{\mathcal{N}(\vec{x}_1, \vec{m}_1, \Sigma_{11})} = \mathcal{N}(\vec{x}_2, \vec{b}, A)$$

mit

$$\begin{aligned} \vec{b} &= \vec{m}_2 + \Sigma_{12}^T \Sigma_{11}^{-1}(\vec{x}_1 - \vec{m}_1) \\ A &= \Sigma_{22} - \Sigma_{12}^T \Sigma_{11}^{-1} \Sigma_{12} \end{aligned}$$

was dem gesuchten Ausdruck entspricht.

Satz A.0.3 (Erwartungswert der Normalverteilung) Sei $\vec{x} \sim \mathcal{N}(\vec{m}, \Sigma)$ ein normalverteilter Vektor mit Mittelwert $\vec{m}$ und Varianz Σ. Dann ist der Erwartungswert von $\vec{x}$:

$$\mathbb{E}[\vec{x}] = \int_{\vec{x}} N(\vec{m}, \Sigma) \cdot \vec{x} dx = \vec{m}$$

Beweis (Erwartungswert der Normalverteilung) Der Satz wird für den eindimensionalen Fall, d.h. $\vec{x} = x \in \mathbb{R}^1, \vec{m} = m \in \mathbb{R}^1$ und $\Sigma = \sigma \in \mathbb{R}^1$ gezeigt. Für die Dichtefunktion gilt:

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-m)^2}{2\sigma^2}\right)$$

Daraus ergibt sich der Erwartungswert:

$$\begin{aligned}
E(X) &= \int_{-\infty}^{\infty} x \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-m)^2}{2\sigma^2}\right) dx \\
E(X) &= \int_{-\infty}^{\infty} (x+m) \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{x^2}{2\sigma^2}\right) dx \\
&= \int_{-\infty}^{\infty} x \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{x^2}{2\sigma^2}\right) dx \; + \; \int_{-\infty}^{\infty} m \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{x^2}{2\sigma^2}\right) dx
\end{aligned}$$

Für das erste Integral ergibt sich:

$$\begin{aligned}
I_1 &= \int_{-\infty}^{0} x \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{x^2}{2\sigma^2}\right) dx + \int_{0}^{\infty} x \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{x^2}{2\sigma^2}\right) dx \\
I_1 &= -\int_{0}^{-\infty} x \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{x^2}{2\sigma^2}\right) dx + \int_{0}^{\infty} x \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{x^2}{2\sigma^2}\right) dx \\
I_1 &= \int_{0}^{\infty} (-x) \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(-x)^2}{2\sigma^2}\right) dx + \int_{0}^{\infty} x \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{x^2}{2\sigma^2}\right) dx \\
\Rightarrow I_1 &= -\int_{0}^{\infty} x \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{x^2}{2\sigma^2}\right) dx + \int_{0}^{\infty} x \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{x^2}{2\sigma^2}\right) dx = 0 \\
\Rightarrow E(X) &= \int_{-\infty}^{\infty} m \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{x^2}{2\sigma^2}\right) dx
\end{aligned}$$

Multipliziere mit $\sigma\sqrt{2}$:

$$\Rightarrow E(X) = \int_{-\infty}^{\infty} m \frac{1}{\sqrt{\pi}} e^{-x^2} dx = m \frac{2}{\sqrt{\pi}} \int_{0}^{\infty} e^{-x^2} dx$$

$$\frac{2}{\sqrt{\pi}} \int_{0}^{\infty} e^{-x^2} dx = \lim_{t\to\infty} \frac{2}{\sqrt{\pi}} \int_{0}^{t} e^{-x^2} dx = 1$$

$$\Rightarrow E(X) = m$$

B | Blockschaltbild der FPGA Systemarchitektur

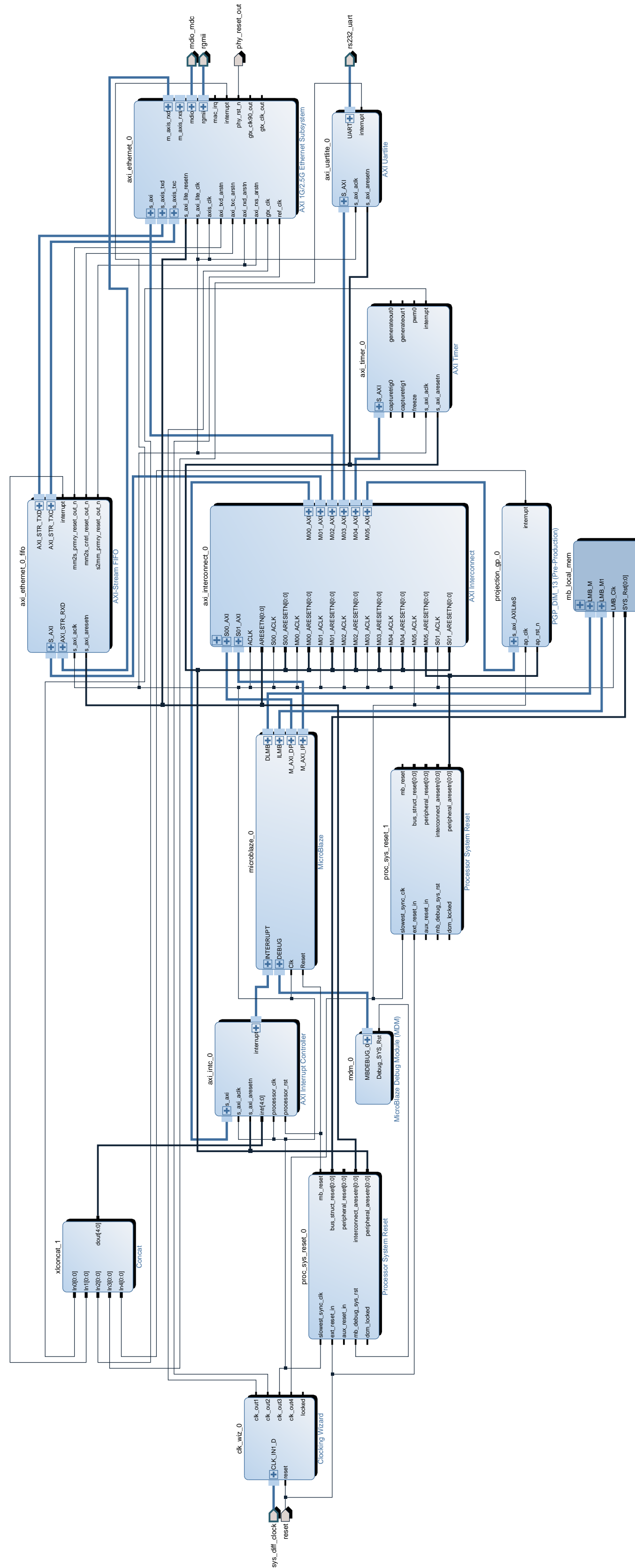


Abbildung B.1: Blockschaltbild der gesamten FPGA Systemarchitektur.

Abbildungsverzeichnis

2.1 Schematisches Vorgehen bei der Systementwicklung 6

3.1 Hardwareüberblick 10
3.2 FPGA: Signalpfadkonfiguration 12
3.3 FPGA: Konfigurierbare Logikblöcke 12
3.4 FPGA: Programmierung 13

6.1 Umsetzung einer Matrix als kontinuierliches Feld im Speicher. 40
6.2 Schematische Darstellung der FPGA Architektur. 45
6.3 TCP Paket zur FPGA Kommunikation 48
6.4 Speicherdefinition von Projection GP. 60
6.5 Kernfunktionsberechnung von Projection GP. 60
6.6 Aufruf von Projection GP. 61
6.7 Schematische Schaltung für die Berechnung der Kernfunktion. 63
6.8 Schematische Schaltung einer Pipeline für die Berechnung der Kernfunktion 64
6.9 Schematische Schaltung einer abgerollten Schleifen für die Berechnung der Kernfunktion. 66

7.1 Vorhersagegenauigkeit für die verwendeten Datensätze. 79
7.2 Durchsatz der verwendeten Architekturen bei verschiedenen Dimensionen. . 81
7.3 Durchsatz der verwendeten Architekturen bei einer unterschiedlichen Anzahl an Basisvektoren. 82
7.4 Der Energieverbrauch der einzelnen Systeme. 83
7.5 Die Effizienz der einzelnen Systeme. 84

B.1 Blockschaltbild der gesamten FPGA Systemarchitektur. 100

Tabellenverzeichnis

6.1 Der TCP/IP Protokollstapel im Vergleich zum ISO-OSI Modell. 47
6.2 Konfiguration von `lwIP`. 50
6.3 Konfiguration von `lwIP` (Fortsetzung). 51
6.4 Portkonfiguration des `AXI 1G/2.5G Ethernet Subsystem`. 53
6.5 FPGA Ausnutzung und Energieverbrauch für das System ohne maschinellen Lernalgorithmus. 56
6.6 Vergleich der Programmgrößen des TCP/IP Protokollstapels unter Minimalkonfiguration und Standardkonfiguration. 56
6.7 Geschätzter Verbrauch von Projection GP auf dem FPGA. 72
6.8 Geschätzter Durchsatz von Projection GP auf dem FPGA. 72
6.9 Realer Systemverbrauch für die zweite Phase von Projection GP 74

7.1 Durchsatz der real synthetisierten Systeme 81
7.2 Energieverbrauch und Effizienz der real synthetisierten Systeme 85
7.3 Automatische Vektorisierung für verschiedene Programme. 86

Literaturverzeichnis

[Alt] *Altera Support.* https://www.altera.com/support.html. – Zugriff Juli 2015

[AMY09] Asano, S. ; Maruyama, T. ; Yamaguchi, Y.: Performance comparison of FPGA, GPU and CPU in image processing. In: *International Conference on Field Programmable Logic and Applications, 2009. FPL 2009.*, 2009. – ISSN 1946–1488, S. 126–131

[Art] *Xilinx Artix-7 FPGA AC701 Evaluation Kit.* http://www.xilinx.com/products/boards-and-kits/ek-a7-ac701-g.html. – Zugriff Dezember 2015

[ASC] Anirudha Sarangi, Stephen M. ; Cherukupaly, Upender: *LightWeight IP Application Examples.* http://www.xilinx.com/support/documentation/application_notes/xapp1026.pdf. – Zugriff Dezember 2015

[Auv] *AuvizDNN: a library to create deep learning algorithms.* http://www.xilinx.com/products/intellectual-property/1-6unyml.html. – Zugriff Dezember 2015

[Boc15] Bockermann, Christian: *Mining Big Data Streams for Multiple Concepts*, TU Dortmund University, Diss., 2015

[BRS13] Bacon, D. ; Rabbah, R. ; Shukla, S.: FPGA programming for the masses. In: *Communications of the ACM* 56 (2013), Nr. 4, S. 56–63

[Bue11] Buede, D.: *The engineering design of systems: models and methods.* John Wiley & Sons, 2011

[C9905] *ISO/IEC 9899 C Standard DRAFT.* http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf. Version: 2005. – Zugriff Dezember 2015

[Can92] Cantor, G.: Über eine elementare Frage der Mannigfaltigkeitslehre. In: *Jahresbericht der Deutschen Mathematiker-Vereinigung* 1 (1892), S. 75–78

[CCL+13] Chen, J. ; Cao, N. ; Low, K. ; Ouyang, R. ; Tan, C. ; Jaillet, P.: Parallel Gaussian process regression with low-rank covariance matrix approximations. In: *arXiv preprint arXiv:1305.5826* (2013)

[CLN+11] Cong, J. ; Liu, B. ; Neuendorffer, S. ; Noguera, J. ; Vissers, K. ; Zhang, Z.: High-level synthesis for FPGAs: From prototyping to deployment. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30 (2011), Nr. 4, S. 473–491

[CLR10] Cormen, T. ; Leiserson, C. ; Rivest, R.: *Algorithmen - Eine Einführung.* Oldenbourg Wissensch.Vlg, 2010. – ISBN 3486275151

[CO02] Csató, L. ; Opper, M.: Sparse on-line Gaussian processes. In: *Neural computation* 14 (2002), Nr. 3, S. 641–668

[Csa02] Csató, L.: *Gaussian processes: iterative sparse approximations*, Aston University, Diss., 2002

[CT11] Cooper, K. ; Torczon, L.: *Engineering a compiler.* Elsevier, 2011

[Dun] Dunkels, A.: *Design and Implementation of the lwIP TCP/IP Stack.* `http://www.ece.ualberta.ca/~cmpe401/docs/lwip.pdf`. – Zugriff Dezember 2015

[Ern98] Ernst, R.: Codesign of embedded systems: status and trends. In: *Design Test of Computers, IEEE* 15 (1998), Apr, Nr. 2, S. 45–54. `http://dx.doi.org/10.1109/54.679207`. – DOI 10.1109/54.679207. – ISSN 0740–7475

[FFL07] Ferris, B. ; Fox, D. ; Lawrence, N.: WiFi-SLAM Using Gaussian Process Latent Variable Models. In: *IJCAI* Bd. 7, 2007, S. 2480–2485

[FHF06] Ferris, B. ; Haehnel, D. ; Fox, D.: Gaussian processes for signal strength-based location estimation. In: *In proc. of robotics science and systems*, 2006

[FO08] Fang, H. ; O'leary, D.: Modified Cholesky algorithms: a catalog with new approaches. In: *Mathematical Programming* 115 (2008), Nr. 2, S. 319–349

[GCC] *GCC 4.6 Release Series.* `https://gcc.gnu.org/gcc-4.6/changes.html#microblaze`. – Zugriff Dezember 2015

[GMHP04] Grochow, K. ; Martin, S. ; Hertzmann, A. ; Popović, Z.: Style-based inverse kinematics. In: *ACM Transactions on Graphics (TOG)* Bd. 23 ACM, 2004, S. 522–531

[HBSE11] Hussain, H.M. ; Benkrid, K. ; Seker, H. ; Erdogan, A.T.: FPGA implementation of K-means algorithm for bioinformatics application: An accelerated

approach to clustering Microarray data. In: *NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2011*, 2011, S. 248–255

[HD08] Hauck, S. ; DeHon, A.: *Reconfigurable computing: the theory and practice of FPGA-based computation.* Morgan Kaufmann, 2008

[HHL15] Hoang, T. ; Hoang, Q. ; Low, B.: A Unifying Framework of Anytime Sparse Gaussian Process Regression Models with Stochastic Variational Inference for Big Data. In: *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, 2015, S. 569–578

[HP11] Hennessy, J. ; Patterson, D.: *Computer architecture: a quantitative approach.* Elsevier, 2011

[HTF01] Hastie, T. ; Tibshirani, R. ; Friedman, J.: *The Elements of Statistical Learning.* Springer New York Inc., 2001 (Springer Series in Statistics)

[IDNG08] Irick, K.M. ; DeBole, M. ; Narayanan, V. ; Gayasen, A.: A Hardware Efficient Support Vector Machine Architecture for FPGA. In: *16th International Symposium on Field-Programmable Custom Computing Machines, 2008.*, 2008, S. 304–305

[IEE] *IEEE 802.3: ETHERNET.* http://standards.ieee.org/about/get/802/802.3.html, . – Zugriff Dezember 2015

[JR13] Jeffers, J. ; Reinders, J.: *Intel Xeon Phi coprocessor high-performance programming.* Morgan Kaufmann Publishers Inc., 2013. – ISBN 9780124104143, 9780124104945

[Kae08] Kaeslin, H.: *Digital integrated circuit design: From VLSI architectures to CMOS fabrication.* Cambridge University Press, 2008

[Kag] *ECML/PKDD 15: Taxi Trajectory Prediction.* `https://www.kaggle.com/c/pkdd-15-predict-taxi-service-trajectory-i`. – Zugriff Dezember 2015

[KCS+10] Kim, C. ; Chhugani, J. ; Satish, N. ; Sedlar, E. ; Nguyen, A. ; Kaldewey, T. ; Lee, V. ; Brandt, S. ; Dubey, P.: FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* ACM, 2010, S. 339–350

[KDW10] Kestur, S. ; Davis, J. ; Williams, O.: Blas comparison on FPGA, CPU and GPU. In: *IEEE computer society annual symposium on VLSI (ISVLSI), 2010* IEEE, 2010, S. 288–293

[KM11] Krishnamoorthy, A. ; Menon, D.: Matrix inversion using Cholesky decomposition. In: *arXiv preprint arXiv:1111.4144* (2011)

[KR08] Kurose, J. ; Ross, K.: *Computernetzwerke: Der Top-Down-Ansatz.* Pearson Deutschland GmbH, 2008

[LCH+14] Low, K. ; Chen, J. ; Hoang, T. ; Xu, N. ; Jaillet, P. ; Natarajan, P. ; Wong, Y. ; Kankanhalli, M.: Recent advances in scaling up Gaussian process predictive models for large spatiotemporal data. In: *Proc. Dy-DESS* (2014)

[LKC+10] Lee, V. ; Kim, C. ; Chhugani, J. ; Deisher, M. ; Kim, D. ; Nguyen, A. ; Satish, N. ; Smelyanskiy, M. ; Chennupaty, S. ; Hammarlund, P. u. a.: Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In: *ACM SIGARCH Computer Architecture News* Bd. 38 ACM, 2010, S. 451–460

[LRL+12] Liang, Y. ; Rupnow, K. ; Li, Y. ; Min, D. ; Do, M. ; Chen, D.: High-level Synthesis: Productivity, Performance, and Software Constraints. In: *JECE* 2012 (2012), Januar, 1:1–1:1. `http://dx.doi.org/10.1155/2012/649057`. – DOI 10.1155/2012/649057. – ISSN 2090–0147

[Mar10] Marwedel, P.: *Embedded system design: Embedded systems foundations of cyber-physical systems.* Springer Science & Business Media, 2010

[Meh] Mehta, Nick: Xilinx 7 Series FPGAs: The Logical Advantage. In: *White Paper: 7 Series FPGAs*

[Mil74] Miller, W.: Computational Complexity and Numerical Stability. In: *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, ACM, 1974 (STOC ’74), 317–322

[Moo07] Moore, A.: *FPGAs for Dummies Altera Special Edition.* Wiley Brand, 2007

[MS10] Misra, J. ; Saha, I.: Artificial neural networks in hardware: A survey of two decades of progress. In: *Neurocomputing* 74 (2010), Nr. 1, S. 239–255

[MVG+12] Meeus, W. ; Van Beeck, K. ; Goedemé, T. ; Meel, J. ; Stroobandt, D.: An overview of today’s high-level synthesis tools. In: *Design Automation for Embedded Systems* 16 (2012), Nr. 3, 31-51. `http://dx.doi.org/10.1007/s10617-012-9096-8`. – DOI 10.1007/s10617–012–9096–8. – ISSN 0929–5585

[MWH13] Monson, J. ; Wirthlin, M. ; Hutchings, B.L.: Implementing high-performance, low-power FPGA-based optical flow accelerators in C. In: *IEEE*

24th International Conference on Application-Specific Systems, Architectures and Processors (ASAP), 2013, 2013. – ISSN 2160–0511, S. 363–369

[NHM+07] Narayanan, R. ; Honbo, D. ; Memik, G. ; Choudhary, A. ; Zambreno, J.: An FPGA implementation of decision tree classification. In: *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07* IEEE, 2007, S. 1–6

[NLB+13] Navarro, D. ; Lucia, O. ; Barragan, L. ; Urriza, I. ; Jimenez, O.: High-level synthesis for accelerating the FPGA implementation of computationally demanding control algorithms for power converters. In: *IEEE Transactions on Industrial Informatics* 9 (2013), Nr. 3, S. 1371–1379

[nvi] *NVIDIA Cuda Programming guide.* http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3wH4EMj00. – Zugriff Dezember 2015

[OR06] Omondi, A. ; Rajapakse, J.: *FPGA implementations of neural networks.* Springer, 2006

[PB08a] Papadonikolakis, M. ; Bouganis, C: Efficient FPGA mapping of Gilbert's algorithm for SVM training on large-scale classification problems. In: *International Conference on Field Programmable Logic and Applications, 2008* IEEE, 2008, S. 385–390

[PB08b] Papadonikolakis, M. ; Bouganis, C.: A scalable FPGA architecture for non-linear SVM training. In: *International Conference on ICECE Technology, 2008*, 2008, S. 337–340

[PH05] Patterson, D. ; Hennessy, J.: *Computer organization and design: the hardware/software interface.* Morgan Kaufmann Publishers Inc., 2005. – ISBN 1–55860–281–X

[PI/a] *Raspberry Pi 2: Performance-Vergleich und Benchmarks.* http://jankarres.de/2015/03/raspberry-pi-2-performance-vergleich-und-benchmarks/. – Zugriff Januar 2016

[PI/b] *Raspberry PI FAQ: WHAT ARE THE POWER REQUIREMENTS?* https://www.raspberrypi.org/help/faqs/#power. – Zugriff Januar 2016

[PP+12] Petersen, K. ; Pedersen, M. u. a.: *The matrix cookbook.* http://www2.imm.dtu.dk/pubdb/p.php?3274, 2012

[QCR05] Quiñonero-Candela, J. ; Rasmussen, C.: A unifying view of sparse approximate Gaussian process regression. In: *The Journal of Machine Learning Research* 6 (2005), S. 1939–1959

[RAS08] Ramos, C. ; Augusto, J. ; Shapiro, D.: Ambient intelligence—The next step for artificial intelligence. In: *Intelligent Systems, IEEE* 23 (2008), Nr. 2, S. 15–18

[Raz02] Raz, R.: On the complexity of matrix product. In: *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing* ACM, 2002, S. 144–151

[Rob05] Robinson, S.: Towards an optimal algorithm for matrix multiplication. In: *SIAM news* 38 (2005), Nr. 9, S. 1–3

[RW] Rasmussen, Carl E. ; Williams, Christopher K. I.: *The SARCOS data.* http://www.gaussianprocess.org/gpml/data/. – Zugrif Dezember 2015

[RW06] Rasmussen, C. ; Williams, C.: *Gaussian processes for machine learning.* The MIT Press, 2006. – 95 S. – ISBN 0–262–18253–X.

[Sad11] Sadri, F.: Ambient intelligence: A survey. In: *ACM Computing Surveys (CSUR)* 43 (2011), Nr. 4, S. 36

[SG05] Snelson, E. ; Ghahramani, Z.: Sparse Gaussian processes using pseudo-inputs. In: *Advances in neural information processing systems*, 2005, S. 1257–1264

[Sil85] Silverman, B.: Some aspects of the spline smoothing approach to non-parametric regression curve fitting. In: *Journal of the Royal Statistical Society. Series B (Methodological)* (1985), S. 1–52

[SS00] Smola, A. ; Schökopf, B.: Sparse Greedy Matrix Approximation for Machine Learning. In: *Proceedings of the Seventeenth International Conference on Machine Learning*, Morgan Kaufmann Publishers Inc., 2000 (ICML ’00). – ISBN 1–55860–707–2, 911–918

[SS04] Smola, A. ; Schölkopf, B.: A tutorial on support vector regression. In: *Statistics and computing* 14 (2004), Nr. 3, S. 199–222

[Sun] Sundararajan, P.: *High Performance Computing Using FPGAs.* http://www.xilinx.com/support/documentation/white_papers/wp375_HPC_Using_FPGAs.pdf

[SWY+10] Shan, Y. ; Wang, B. ; Yan, J. ; Wang, Y. ; Xu, N. ; Yang, H.: FPMR: MapReduce Framework on FPGA. In: *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2010 (FPGA ’10). – ISBN 978–1–60558–911–4, 93–102

[TCL] TCL: *TCL Developer Exchange*. http://www.tcl-lang.org/. – Zugriff Dezember 2015

[Tei12] Teich, J.: Hardware/Software Codesign: The Past, the Present, and Predicting the Future. In: *Proceedings of the IEEE* 100 (2012), May, Nr. Special Centennial Issue, S. 1411–1430. http://dx.doi.org/10.1109/JPROC.2011.2182009. – DOI 10.1109/JPROC.2011.2182009. – ISSN 0018–9219

[THL09] Thomas, D. ; Howes, L. ; Luk, W.: A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In: *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays* ACM, 2009, S. 63–72

[UCI] UCI: *Housing Data Set*. https://archive.ics.uci.edu/ml/datasets/Housing. – Zugriff Dezember 2015

[Veg] Vega, Jun D.: *Intel Power Gadget*. https://software.intel.com/en-us/articles/intel-power-gadget-20. – Zugriff Dezember 2015

[Ver] *Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language*. http://www.eda.org/sv-ieee1800/. – Zugriff Dezember 2015

[VHD] *VHDL Analysis and Standardization Group (VASG)*. http://www.eda.org/twiki/bin/view.cgi/P1076/WebHome. – Zugriff Dezember 2015

[Viva] *Vivado Design Suite*. http://www.xilinx.com/products/design-tools/vivado.html. – Zugriff Dezember 2015

[Vivb] *Vivado High-Level Synthesis*. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html. – Zugriff Dezember 2015

[Vivc] *Vivado SDK*. http://www.xilinx.com/products/design-tools/embedded-software/sdk.html. – Januar Dezember 2015

[VM64] Von Mises, R.: *Mathematical theory of probability and statistics*. Academic Press, 1964

[Wei91] Weiser, M.: The computer for the 21st century. In: *Scientific american* 265 (1991), Nr. 3, S. 94–104

[WH00] Wirth, R. ; Hipp, J.: CRISP-DM: Towards a standard process model for data mining. In: *Proceedings of the 4th International Conference on the Practical Applications of Knowledge Discovery and Data Mining*, 2000, S. 29–39

[WHTS14] Wang, R. ; Hamilton, T.J. ; Tapson, J. ; Schaik, A. van: An FPGA design framework for large-scale spiking neural networks. In: *IEEE International Symposium onCircuits and Systems (ISCAS), 2014*, 2014, S. 457–460

[Wil14] Williams, V.: *Multiplying matrices in O(n2.373) time.* `http://theory.stanford.edu/~virgi/matrixmult-f.pdf`, 2014

[WL08] Wu, E. ; Liu, Y.: Emerging technology about GPGPU. In: *IEEE Asia Pacific Conference on Circuits and Systems, 2008.* IEEE, 2008, S. 618–622

[WLG+99] Wahba, G. ; Lin, X. ; Gao, F. ; Xiang, D. ; Klein, R. ; Klein, B.: The Bias-variance Tradeoff and the Randomized GACV. In: *Proceedings of the 1998 Conference on Advances in Neural Information Processing Systems II*, MIT Press, 1999. – ISBN 0–262–11245–0, 620–626

[WPB+09] Willhalm, T. ; Popovici, N. ; Boshmaf, Y. ; Plattner, H. ; Zeier, A. ; Schaffner, J.: SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. In: *Proceedings of the VLDB Endowment* 2 (2009), Nr. 1, S. 385–394

[Xila] *10 Gigabit Ethernet Subsystem v3.0.* `http://www.xilinx.com/support/documentation/ip_documentation/axi_10g_ethernet/v3_0/pg157-axi-10g-ethernet.pdf`. – Zugriff Januar 2016

[Xilb] *7 Series DSP48E1 Slice.* `http://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf`. – Zugriff Januar 2016

[Xilc] *AXI 1G/2.5G Ethernet Subsystem v7.0.* `http://www.xilinx.com/support/documentation/ip_documentation/axi_ethernet/v7_0/pg138-axi-ethernet.pdf`. – Zugriff Januar 2016

[Xild] *AXI Reference Guide UG1037 v3.0.* `http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf`. – Zugriff Januar 2016

[Xile] *MicroBlaze Processor Reference Guide.* `http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/ug984-vivado-microblaze-ref.pdf`. – Zugriff Januar 2016

[Xilf] *Tri-Mode Ethernet MAC v9.0.* `http://www.xilinx.com/support/documentation/ip_documentation/tri_mode_ethernet_mac/v9_0/pg051-tri-mode-eth-mac.pdf`. – Zugriff Januar 2016

[Xilg] *Vivado Design Suite User Guide - Design Analysis and Closure Techniques.* http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug906-vivado-design-analysis.pdf#nameddest=Floorplanning. – Zugriff Januar 2016

[Xilh] *Vivado High-Level Synthesis User Guide.* http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug902-vivado-high-level-synthesis.pdf. – Zugriff Dezember 2015

[Xili] *Xilinx: 7 Series FPGAs Overview.* http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf. – Zugegriffen Dezember 2015

[Xilj] *Xilinx: Smarter Data Center.* http://www.xilinx.com/applications/data-center.html. – Zugriff Dezember 2015

[Xilk] *Xilinx: Support.* http://www.xilinx.com/support.html. – Zugriff Dezember 2015

[Xill] *Xilinx: What is an FPGA?* http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm. – Zugriff Juli 2015

[Xilm] *Zynq-7000 AP SoC and 7 Series Devices Memory Interface Solutions v2.3.* http://www.xilinx.com/support/documentation/ip_documentation/mig_7series/v2_3/ug586_7Series_MIS.pdf. – Zugriff Januar 2016

[XLC+14] Xu, N. ; Low, K. ; Chen, J. ; Lim, K. ; Ozgul, E.: GP-Localize: Persistent Mobile Robot Localization using Online Sparse Gaussian Process Observation Model. In: *CoRR* abs/1404.5165 (2014). http://arxiv.org/abs/1404.5165

[ZS03] Zhu, J. ; Sutton, P.: FPGA implementations of neural networks–a survey of a decade of progress. In: *Field Programmable Logic and Application.* Springer, 2003, S. 1062–1066