HAW
HAMBURG

Bachelorarbeit

Jesse Stricker

Implementierung und Evaluation eines FPGA-Koprozessors für die Bilderkennung unter Verwendung von Convolutional Neural Networks

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

Jesse Stricker

Implementierung und Evaluation eines FPGA-Koprozessors für die Bilderkennung unter Verwendung von Convolutional Neural Networks

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Informatik Technischer Systeme*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Tim Tiedemann
Zweitgutachter: Prof. Dr. Michael Schäfers

Eingereicht am: 30. November 2022

Jesse Stricker

Thema der Arbeit

Implementierung und Evaluation eines FPGA-Koprozessors für die Bilderkennung unter Verwendung von Convolutional Neural Networks

Stichworte

FPGA, Koprozessor, Bilderkennung, CNN

Kurzzusammenfassung

Die Erkennung von Bildmerkmalen ist ein fester Bestandteil vieler neuen Anwendungen aus dem Bereich der Internet-of-Things-Geräte. Diese Arbeit stellt einen FPGA-Koprozessor vor, der auf Grundlage einer skalierbaren Tensor-Processing-Unit-Architektur entworfen wurde. Es wird sich mit Optimierungen auseinander gesetzt, die es ermöglichen, Convolutional Neural Networks selbst auf leistungsschwächeren Systemen zu verwenden.

Jesse Stricker

Title of Thesis

Implementation and evaluation of an FPGA coprocessor for image recognition using convolutional neural networks

Keywords

FPGA, Coprocessor, image recognition, CNN

Abstract

Image feature recognition is an integral part of many emerging applications from the field of Internet-of-Things devices. This work presents an FPGA coprocessor which was designed based on a scalable tensor processing unit architecture. Optimizations are addressed that allow convolutional neural networks to be used even on lower performance systems.

Inhaltsverzeichnis

Abbildungsverzeichnis **vi**

Tabellenverzeichnis **vii**

1 Einleitung **1**

1.1 Motivation 1

1.2 Zielsetzung 2

1.3 Aufbau der Arbeit 2

2 Grundlagen **3**

2.1 Maschinelles Lernen 3

2.1.1 Künstliche neuronale Netze 3

2.1.2 Faltungsnetzwerke 4

2.1.3 Batch-Normalization 5

2.1.4 Aktivierungsfunktion 7

2.2 Field Programmable Gate Array 8

2.2.1 Registertransferebene 9

2.2.2 Hardwarebeschreibungssprachen 9

2.2.3 Festkommazahlen 10

3 Verwandte Arbeiten **12**

3.1 Googles Tensor Processing Unit 12

3.2 tinyTPU 13

4 Konzeption **15**

4.1 Faltung durch Matrixmultiplikation 15

4.2 Depthwise separable Convolutions 15

4.3 Akkumulatoren 17

4.4 Batch Normalization 17

4.5 Softmax-Aktivierungsfunktion 18

4.6 Instruktionssatz 19

5 Implementierung **21**

5.1 Verwendete Technologien 21

5.2 Anpassung an die FPGA-Umgebung 21

5.2.1 Softmax-Aktivierungsfunktion 22

6 Evaluation **24**

6.1 Depthwise seperable Convolutions 24

7 Zusammenfassung und Fazit **26**

7.1 Ausblick auf Weiterentwicklung 26

7.1.1 Verschiedene Komponentengrößen 26

7.1.2 Unterstützung von dünnbesetzten Matrizen 27

7.1.3 Faltungsoperation mit Stride-Parameter 27

Literaturverzeichnis **28**

A Abbildungen **31**

Selbstständigkeitserklärung 33

Abbildungsverzeichnis

2.1 Ein einzelnes Perzeptron kombiniert mehrere Eingänge zu einer Ausgabe [18] 3
2.2 Ein mehrlagiges Perzeptron mit vielen Hidden Layers [18] 4
2.3 Der erste Schritt einer Faltungsoperation. Abgewandelt von [18] 5
2.4 Ein Faltungslayer mit M Eingaben, N Ausgaben und $M \times N$ Faltungskernen. Aktivierungsfunktion nicht dargestellt. 6
2.5 Graf der ReLU-Funktion 8
2.6 Ein digitale Schaltung mit einem Inverter (links) und einem taktflankengesteuerten D-Flip-Flop (rechts) 9

3.1 Blockdiagramm von Googles Tensor Processor Unit[13] 12

4.1 Eine depthwise Convolution mit M Eingaben, M Ausgaben und M Faltungskernen. 16
4.2 Eine pointwise Convolution mit M Eingaben, N Ausgaben und $M \times N$ Faltungskernen der Größe 1×1. 17
4.3 Innerer Aufbau der Batch-Normalization-Komponente. B bezeichnet die Busbreite 18
4.4 Die Exponentialfunktion e^x. Auf dem rechten Graphen lassen sich die Effekte der Quantisierung erkennen 19
4.5 Die absolute Abweichung zwischen der reellen und der quantisierten Exponentialfunktion 20

5.1 Block-Design mit Zynq UltraScale+ MPSoC und Custom IP 22

A.1 Schaltplan der max-Baumstruktur aus der RTL-Analyse von Xilinx Vivado 31
A.2 Block-Design mit Zynq UltraScale+ MPSoC und Custom IP (vergrößert) 32

Tabellenverzeichnis

4.1 Erweiterter Instruktionssatz 20

6.1 Leistungsvergleich von zwei MobileNet-Netzwerken[8] 25

1 Einleitung

Der Einsatz von maschinellen Lernverfahren (ML) zur Lösung komplexer Probleme ist mittlerweile weit verbreitet. Die hohe Vielfalt der praktischen Anwendungsmöglichkeiten reicht von der automatischen Erstellung medizinischer Diagnosen über die Verarbeitung natürlicher Sprachen bis hin zur Hinderniserkennung beim autonomen Fahren.

Dabei umfassen die Hardwaretypen, auf denen diese Verfahren umgesetzt werden, ein breites Spektrums. Auf der einen Seite wird es auf großen Rechenclustern mit über 100 Grafikprozessoren (GPU) eingesetzt, andererseits nutzen selbst kleinere eingebettete Systeme im Internet-of-Things-Umfeld dies beispielsweise für eine lokale Klassifikation von Sensordaten. Letztere stehen in dem Hauptfokus dieser Arbeit.

1.1 Motivation

Umsetzungen von ML-Modellen in eingebetteten Systemen bringen spezifische Anforderungen mit sich, welche z. B. bei Grafikprozessoren (GPUs) in privaten Anwendungsfällen zu vernachlässigen sind.

Da die Geräte häufig mit Batterien begrenzter Kapazität betrieben werden, gilt es, die Hardware möglichst energieeffizient zu gestalten. Dabei gibt es jedoch auch häufig harte Deadlines, die eingehalten werden müssen, da ansonsten im Extremfall lebensbedrohliche Situationen für die Anwender entstehen können.

Trotz all dieser Einschränkungen darf dies nicht zu Abstrichen bezüglich der Genauigkeit von Vorhersagen führen, damit die Verlässlichkeit der getroffenen Entscheidungen gewährleisten werden kann.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist es, den Prozess der Bilderkennung in der beschriebenen Umgebung eines eingebettenen Systems zu beschleunigen. Es soll hierfür ein Koprozessor konzipiert und auf einem Field Programmable Gate Array (FPGA) implementiert werden. Dabei wird auf eine Tensor-Processing-Unit (TPU) ähnliche Architektur zurückgegriffen, welche die flexible Ausführung unterschiedlichster künstlicher neuronaler Netze ermöglicht.

Der Hauptfokus dieser Arbeit liegt darauf, die vorhandene TPU um eine Unterstützung von Faltungsnetzwerken am Beispiel der MobileNets[8] zu erweitern.

1.3 Aufbau der Arbeit

In den nächsten zwei Kapiteln werden erst die themenspezifischen Grundlagen und danach einige verwandte Arbeiten vorgestellt. Darauf folgt die Konzeption, in der das entworfene System erläutert wird. Im Kapitel *Implementierung* wird auf die Umsetzung der Architektur auf einem FPGA eingegangen. Anschließend findet eine Auswertung des Systems und ein Vergleich zu Alternativen statt. Das letzte Kapitel enthält eine Zusammenfassung dieser Arbeit mit einem Ausblick auf Möglichkeiten der Weiterentwicklung.

2 Grundlagen

Dieses Kapitel erläutert die wichtigsten Grundlagen, die zum Verständnis dieser Arbeit hilfreich sind.

2.1 Maschinelles Lernen

Als maschinelles Lernen bezeichnet man den Prozess, bei dem Maschinen aus Daten „Wissen“ gewinnen können. Wenn Algorithmen aus einer Menge Trainingsdaten und Validierungsdaten selbstständig die Funktion zur Klassifikation erlernen, spricht man vom Teilgebiet des überwachten Lernens.

2.1.1 Künstliche neuronale Netze

Ein künstliches neuronales Netz beschreibt eine Struktur aus künstlichen Neuronen (Abb. 2.1), die auch als (einfache) Perzeptronen[16] bezeichnet werden. Entstanden sind sie aus der Idee, natürliche neuronale Netze nachzubilden, wie sie z. B. im menschlichen Gehirn existieren.

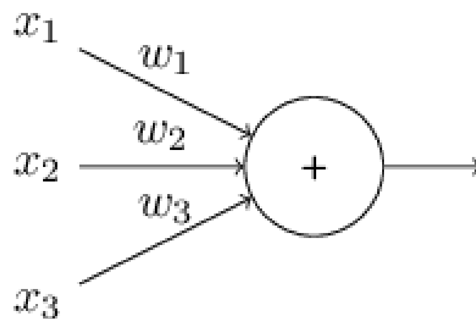

Abbildung 2.1: Ein einzelnes Perzeptron kombiniert mehrere Eingänge zu einer Ausgabe [18]

Mitlerweile existieren unzählige Weiterentwicklungen und Abwandlungen. Eine davon ist das mehrlagige Perzeptron (MLP). Bei steigender Anzahl an Layern spricht man auch von *Deep Learning*[6] (Abb. 2.2).

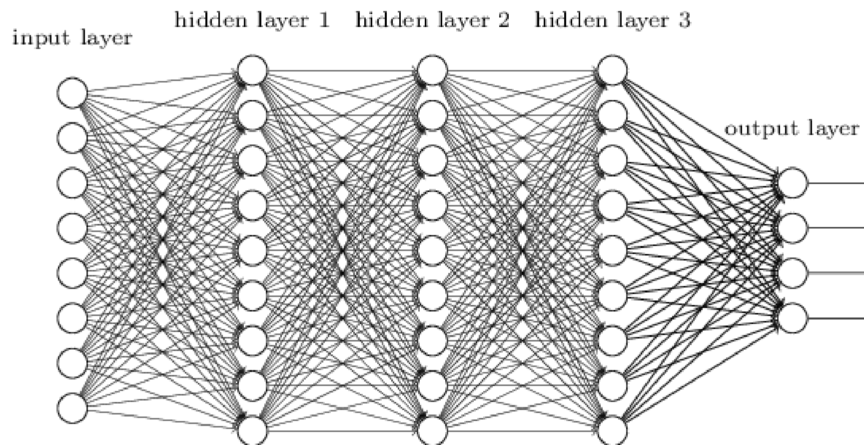

Abbildung 2.2: Ein mehrlagiges Perzeptron mit vielen Hidden Layers [18]

Bevor ein neuronales Netz in einer Produktionsumgebung z. B. zur Klassifikation eingesetzt werden kann, muss es *trainiert* werden. Dabei erlernt das Netz die Gewichtsverteilung der Neuronen, indem mit verschiedenen Optimierungsstrategien versucht wird, den Fehler zu minimieren. Der Fehler ist meist definiert als mittlere quadratische Abweichung (englisch MSE) zwischen gewollter und tatsächlicher Netzausgabe. Ist das Training abgeschlossen, werden die Gewichte nicht mehr verändert und das Netz kann zur *Inferenz*, also zur Schlussfolgerung von Entscheidungen, angewandt werden.

2.1.2 Faltungsnetzwerke

Faltungsnetzwerke[15] nutzen anstatt einfacher MLP-Layer die sogenannten Faltungslayer. Diese sind besonders für zwei-dimensionale Daten wie Bilder geeignet, da bei einer Faltung die Nachbarschaftbeziehungen der Bildpixel berücksichtigt werden.

In einem Faltungslayer sind die Neuroneingaben und -ausgaben keine skalaren Werte, sondern Featuremaps: zwei-dimensionale Pixelmatrizen. Die Neuronen wenden die mehrdimensionale diskrete Faltungsoperation mit den im Training erlernten Faltungskernen auf die Eingabe-Featuremaps an und addieren diese zusammen. Die Faltungskerne entsprechen hier den erlernten Gewichten eines MLP-Netzes.

Die folgende Berechnung wird pro Neuron ausgeführt:

$$(\boldsymbol{G})_{k,l} = \sum_{m=1}^{M} (\boldsymbol{K})_m * (\boldsymbol{F})_m \tag{2.1}$$

Hierbei sind $\boldsymbol{F}$ die Werte der M Eingabe-Featuremaps mit einer Größe von jeweils $D_F \times D_F$, $\boldsymbol{K}$ die M Falktungskerne mit einer Größe von $D_K \times D_K$ und $\boldsymbol{G}$ die Ausgabe-Featuremap mit einer Größe von $D_G \times D_G$.

Die Faltungsoperation ist in Gleichung (2.2) definiert und setzt einen Faltungskern mit ungerader Seitenlänge D_K vorraus. Es wird von einer Stride von 1 und der Verwendung von Zero-Padding ausgegangen.

$$(\boldsymbol{K} * \boldsymbol{F})_{k,l} = \sum_{i=1}^{D_K} \sum_{j=1}^{D_K} (\boldsymbol{K})_{i,j} \cdot (\boldsymbol{F})_{k-s+i,l-s+j} \text{ mit } s = \frac{D_K - 1}{2} \tag{2.2}$$

In der Abb. 2.3 ist diese Operation dargestellt. Die Falktungskerne kann man sich als gleitendes Fenster vorgestellen. Die Verknüpfung zwischen den Neuronen wird in Abb. 2.4 gezeigt.

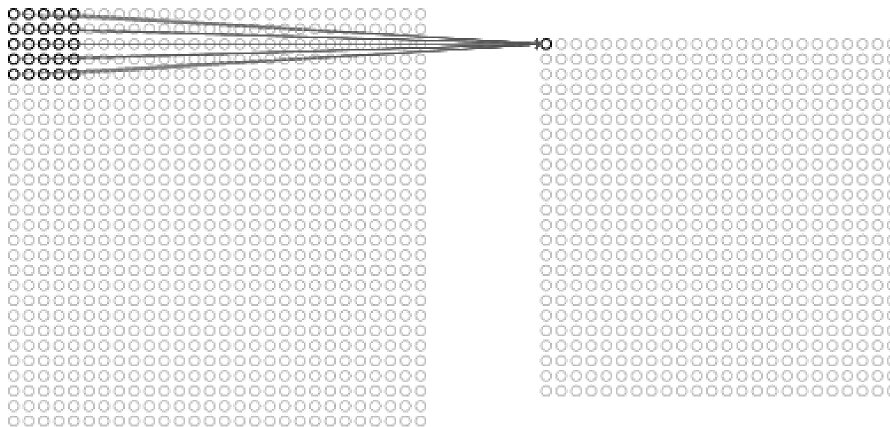

Abbildung 2.3: Der erste Schritt einer Faltungsoperation. Abgewandelt von [18]

2.1.3 Batch-Normalization

Bei der Batch-Normalization[12] werden die Ausgaben der Neuronen $\vec{x}$ pro Merkmal zwischen den Layern standardisiert (Gleichung (2.3)), sodass sie einen Mittelwert von $\mu = 0$ und eine Standardabweichung von $\sigma = 1$ besitzen.

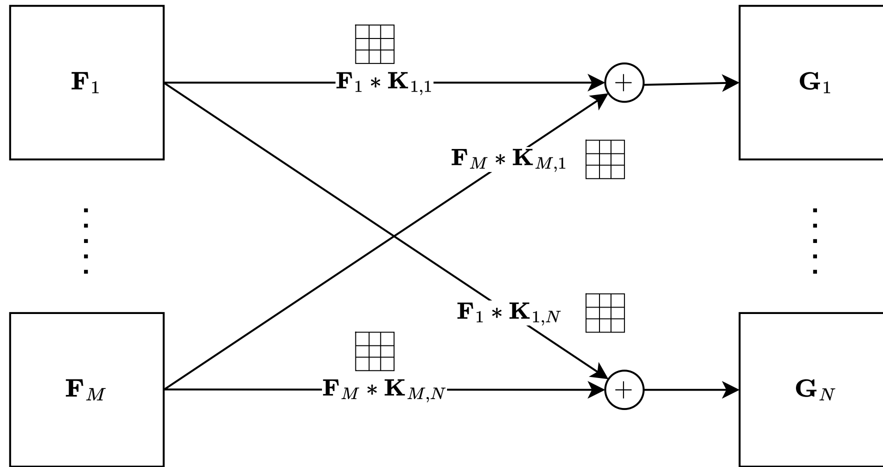

Abbildung 2.4: Ein Faltungslayer mit M Eingaben, N Ausgaben und $M \times N$ Faltungskernen. Aktivierungsfunktion nicht dargestellt.

$$\hat{x}_i = \frac{x_i - \mu}{\sigma} \tag{2.3}$$

$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{2.4}$$

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^2} \tag{2.5}$$

Dabei ist n die Größe des Mini-Batches, einer Untermenge der Trainingsdaten, für die die Standardisierung durchgeführt wird. Dies hilft unter anderem, die Traningsgeschwindigkeit zu erhöhen.

Während der Inferenz liegen die Eingaben des Netzes nicht in Batches, sondern einzeln vor, daher können keine Mittelwerte oder Standardabweichungen ermittelt werden. Als Ersatz werden sich während des Trainings schleifende Mittelwerte und Standardabweichungen gemerkt, die dann als Konstanten in der Inferenzphase verwendet werden können. Somit ist die Batch-Normalization eine einfache lineare Transformation.

Um den Wertebereich, den ein Layer repräsentieren kann, beizubehalten, werden die standardisierten Werte $\hat{x}$ mit den zwei zusätzlichen Parametern γ, β skaliert und verschoben (Gleichung (2.6)). Diese werden bei dem Training pro Batch-Normalization mitgelernt.

$$y_i = \gamma \hat{x} + \beta \tag{2.6}$$

Bei Faltungsnetzen werden die Mittelwerte und Standardabweichung pro Mini-Batch über alle Elemente der jeweiligen Feature-Map berechnet. Während der Inferenz verwenden ebenfalls alle Elemente einer Feature-Map die selben erlernten Parameter γ, β. Dies dient dazu, die Eigenschaften der Faltungsoperation beizubehalten.

Die Batch-Normalization wird meist als ein zusätzlicher Layer zwischen den Neuronen verstanden und kann sowohl vor als auch nach der Aktivierungsfunktion angesetzt werden[12].

2.1.4 Aktivierungsfunktion

Die Aktivierungsfunktion wird auf den Ausgaben eines Neurons angewandt. Durch die Verwendung einer nicht-linearen Funktion ist das Netz in der Lage, komplexe Kombinationen von Features zu erlernen. Es exitieren eine ganze Reihe an verschiedenen Aktivierungsfunktionen, von denen zwei relevante hier gezeigt werden.

Rectified Linear Unit

Die Rectified-Linear-Unit-Funktion[7] (ReLU) ist in Gleichung (2.7) definiert und in Abb. 2.5 zu sehen. Sie wird besonders häufig in den versteckten Layern tiefer Netzwerken[5] verwendet.

$$f(x) = \max(0, x) = \begin{cases} x & \text{für } x > 0 \\ 0 & \text{für } x \leq 0 \end{cases} \tag{2.7}$$

Softmax-Funktion

Wenn ein neuronales Netz zur Einordnung von Daten in mehr als zwei Klassen eingesetzt wird, hat sich für den letzten Layer die Softmax-Funktion als Aktivierungfunktion bewährt. Sie bildet einen Vektor von reelen Zahlen auf eine Wahrscheinlichkeitsverteilung

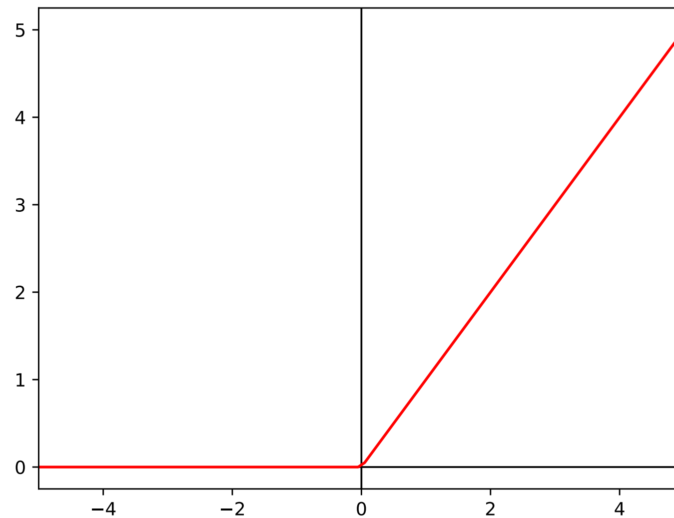

Abbildung 2.5: Graf der ReLU-Funktion

ab, das bedeutet, dass alle Werte im Interval $[0, 1]$ liegen und in Summe 1 ergeben. In Gleichung (2.8) ist die Softmax-Funktion in ihrer Variante ohne Parameter angegeben.

$$f(\vec{x})_i = \frac{\mathrm{e}^{x_i}}{\sum_k^K \mathrm{e}^{x_k}} \text{ mit } \vec{x} \in \mathbb{R}^K \tag{2.8}$$

Im folgenden ist ein Beispiel aufgeführt:

$$\begin{aligned} \vec{x} &= (\quad -2; \quad 4; \quad 3; \quad -1; \quad 0{,}5) \\ \vec{y} = f(\vec{x}) &\approx (0{,}002; \quad 0{,}711; \quad 0{,}261; \quad 0{,}005; \quad 0{,}021) \end{aligned}$$

2.2 Field Programmable Gate Array

Ein Field Programmable Gate Array (FPGA) ist ein integrierter Schaltkreis, dessen Verhalten nicht wie bei Mikrokontrollern von Software programmiert wird, sondern dessen digitale Schaltung nach der Herstellung erst von den Endkunden konfiguriert wird.

Sie setzen sich aus einem Gitter an programmierbaren Logikblöcken (*configurable logic block*, CLB) zusammen, welche mehrere D-Flip-Flops, Look-Up-Tabellen und Volladdierer

umfassen. Über diese CLBs lassen sich die benötigten Schaltungen beliebig realisieren. Für häufig genutze Operationen stellen die FPGA-Hersteller feste Blöcke bereit, die zwar nicht frei programmierbar sind, dafür aber deutlich schneller in ihrer Berechnung.

2.2.1 Registertransferebene

Die Registertransferebene (*register transfer level*, RTL) ist eine der Abstraktionsebenen, zwischen denen bei der Hardwaremodellierung unterschieden wird. Sie erlaubt die Definition von Signalflüssen zwischen Registern (sequentielle Logik) und die Anwendung von logischen Operationen (kombinatorische Logik) auf diesen. Auf dieser Ebene wird meist gearbeitet, wenn die Schaltung eines FPGAs beschrieben wird. In Abb. 2.6 ist ein simples Beispiel zu sehen.

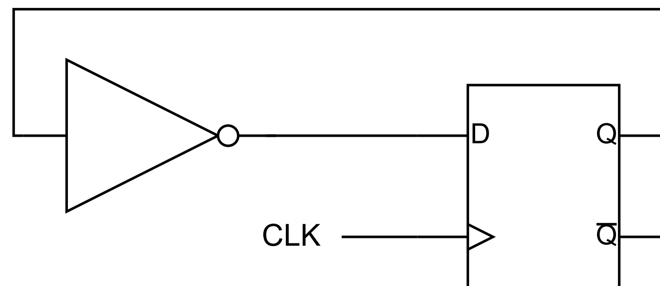

Abbildung 2.6: Ein digitale Schaltung mit einem Inverter (links) und einem taktflankengesteuerten D-Flip-Flop (rechts)

2.2.2 Hardwarebeschreibungssprachen

Zur Modellierung des Verhaltens von digitalen Schaltungen kommen die Hardwarebeschreibungssprachen zum Einsatz. Sie sind keine Programmiersprachen wie z. B. Java oder C++ im engeren Sinne, sondern ermöglichen es das Verhalten und die Struktur der digitalen Schaltungen zu definieren. Dieser Code wird als *synthetisierbar* bezeichnet. Dies gilt nicht für den Code, welcher für die Simulationslogik geschrieben wird.

Der Kompilierungsprozess ist in zwei Schritte aufgeteilt. Zuerst wird die *Synthese* durchgeführt: Hierbei leitet das Synthesetool aus dem Code eine Netzliste ab, welche die Verbindungen von Logikgattern und Speicherblöcken beschreibt. Danach kann die *Implementierung* stattfinden, welche die Netzliste in ein optimiertes physisches Layout der

verwendeten Komponenten und Drähte transformiert. Dieser Schritt wird deshalb auch *place and route* genannt.

Eine der am weitesten verbreiteten Hardwarebeschreibungssprachen ist die *VHSIC Hardware Description Language* (VHDL)[11], welche 1983 im Auftrag des Verteidigungsministeriums der Vereinigten Staaten entworfen wurde. Ihre Syntax und viele Konzepte (z. B. die starke Typisierung) wurden von der Programmiersprache Ada übernommen. VHDL trennt die Beschreibung einer Hardwarekomponente in die Außenansicht (`ENTITY`) und in den inneren Aufbau sowie Verhalten (`ARCHITECTURE`).

2.2.3 Festkommazahlen

Speicher- und Rechenkapazitäten sind auf FPGAs begrenzt. Die sonst auf CPUs und GPUs weit verbreiteten Gleitkommazahlen[10] werden aus diesen Gründen in ressourcenarmen Umgebungen je nach Anwendungsfall vermieden. Eine Alternative zur Kodierung und Berechnung von Zahlen sind die Festkommazahlen[9]. Im Vergleich zu den Gleitkommazahlen, welche nur für bestimmte Registergrößen standardisiert sind, lassen sich Festkommazahlen in jeder beliebigen Registergrößen verwenden. Außerdem sind die arithmetischen Operationen ein Vielfaches schneller zu berechnen, denn sie lassen sich mit wenigen Addierern und Multiplizierern umsetzen.

Ein bedeutender Nachteil ist jedoch, dass ein Festkommazahlentyp eine feste Genauigkeit in seinem Wertebereich hat, während Gleitkommazahlentypen sowohl sehr große als auch sehr kleine Zahlen mit einer guten relativen Genauigkeit repräsentieren können.

Mit den Festkommazahlen lassen sich rationalen Zahlen mit einer festgelegten Anzahl an Nachkommastellen repräsentieren. Der Wert einer solchen Zahl wird durch einen Quotienten aus Zähler und Nenner bestimmt. Der Nenner bleibt dabei für alle Festkommazahlen des selben Typs gleich, daher wird lediglich der Zähler als informationstragend betrachtet. Da der Nenner dennoch wichtig für den Wert ist, wird dieser üblicherweise im Typensystem der Programmiersprache oder in der Dokumentation festgehalten.

Bei Festkommazahlen wird der Zähler als einfache Bitreihenfolge gespeichert, genauso wie bei ganzen Zahlen. Ebenso findet auch hier das Zweierkomplement Verwendung bei der Kodierung der vorzeichenbehafteten Festkommazahlen.

Die verfügbaren Bits werden in m Ganzzahlstellen und n Nachkommastellen aufgeteilt. Der feste Nenner hat dann den Wert 2^n, was einer Auflösung (kleinster Abstand zwischen

benachbarten Werten) von $\frac{1}{2^n} = 2^{-n}$ entspricht. Der Zähler fällt in einen Bereich von $\left[0, 2^{m+n} - 1\right]$ oder $\left[-2^{m+n-1}, 2^{m+n-1} - 1\right]$, abhängig davon ob die Festkommazahl vorzeichenbehaftet ist oder nicht.

Der Wertebereich für Festkommazahlen mit (Gleichung (2.9)) und ohne (Gleichung (2.10)) Vorzeichen ist dann wie folgt:

$$\left[\frac{-2^{m+n-1}}{2^n}, \frac{2^{m+n-1} - 1}{2^n}\right] = \left[-2^{m-1}, 2^{m-1} - 2^{-n}\right]. \tag{2.9}$$

$$\left[\frac{0}{2^n}, \frac{2^{m+n} - 1}{2^n}\right] = \left[0, 2^m - 2^{-n}\right] \tag{2.10}$$

Die Parameter m und n eines Festkommazahlentyps werden üblicherweise mit der sogenannten Q-Notation angegeben. Es existieren zwei verschiedene Varianten, eine von Texas Instruments und eine von ARM. In diesem Dokument wird die von ARM definierte Variante verwendet, welchen von ARM selbst als „Q-format“[1] bezeichnet wird.

Vorzeichenbehaftete Festkommazahlen werden mit Q$m.n$ spezifiziert, vorzeichenlose mit UQ$m.n$.

Zum Beispiel liegen Zahlen des Typs Q3.5 in einem Bereich von $\left[-2^{3-1}, 2^{3-1} - 2^{-5}\right] = [-4, 3.96875]$ mit einer Auflösung von $2^{-5} = 0.03125$. Sie können in Registern mit einer Breite von $3 + 5 = 8$ Bit gespeichert werden.

3 Verwandte Arbeiten

In diesem Kapitel werden andere Arbeiten beschrieben, die sich mit demselben Thema befassen. Einige davon dienen als Grundlage für diese Arbeit.

3.1 Googles Tensor Processing Unit

Die ursprüngliche Tensor Processing Unit (TPU) wurde 2017 in einer Arbeit von Google[13] vorgestellt. Sie basiert auf einer eigens entworfenen ASIC (anwendungsspezifische integrierte Schaltung), die seit 2015 in den Rechenzentren zur Beschleunigung von kompletten Inferenz-Modellen in neuronalen Netzen eingesetzt wird.

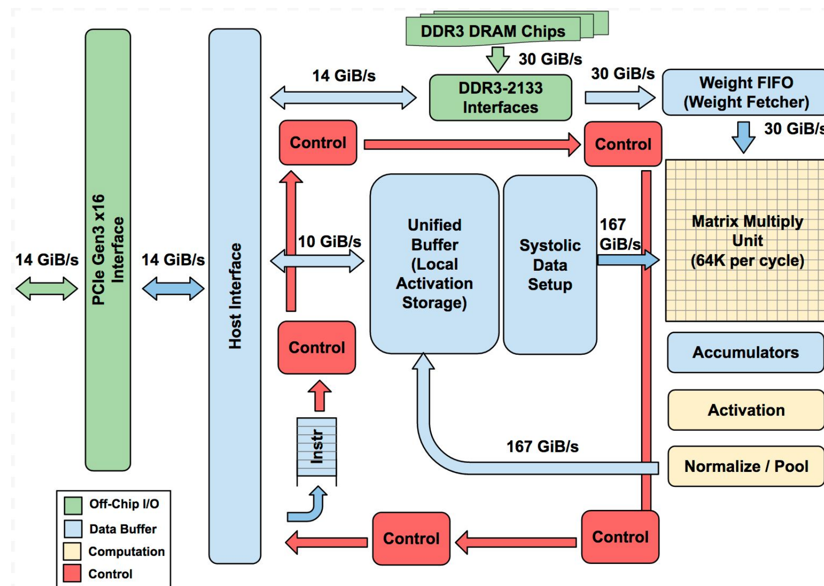

Abbildung 3.1: Blockdiagramm von Googles Tensor Processor Unit[13]

Abb. 3.1 zeigt eine Übersicht über die verschiedenen Komponenten der Architektur. Die TPU ist mit einem PCIe-Gen3-Bus an den Host angeschlossen, worüber die Instruktionen

empfangen werden, bevor sie in einem Puffer gespeichert werden. Ebenfalls werden die Netzeingangswerte (8 Bit breit) über diesen Bus in den Unified Buffer geladen. Die erlernten Gewichte (8 Bit breit) des Modells werden aus einem Speicher außerhalb des Chips geladen und in dem Weight FIFO zwischengespeichert.

Die Matrix Multiply Unit führt die eigentliche Berechnung der Netzwerk-Layer durch, die Matrizenmultiplikation der Werte aus dem Unified Buffer mit den Gewichten aus dem Weight FIFO. Dafür nutzt sie 256×256 MAC-Einheiten (Multiply-Accumulate), deren Ergebnisse in den Accumulator-Speichern (32 Bit breit pro Eintrag) gesammelt werden. Die Activation-Einheit führt die nicht-lineare Aktivierungsfunktion auf den Daten des Accumulators aus und speichert sie wieder im Unified Buffer.

Zwischen dem Unified Buffer und der Matrix Multiply Unit liegt das Systolic Data Setup. Dieses sorgt dafür, dass die Zeilen der Matrix mit den Eingabewerten jeweils um einen Takt versetzt ankommen. Durch dieses Pipeline-Verfahren müssen die Werte aus dem Unified Buffer pro Matrizenmultiplikation nur einmal gelesen werden. Das erhört einerseits die Latenz, jedoch sind arithmetische Operationen weitaus ernergieeffizienter als Speicherzugriffe.

3.2 tinyTPU

Die *tinyTPU*[4], welche als Bachelorarbeit von Jonas Fuhrmann entstanden ist, ist eine quell-offene Implementierung einer Tensor Processing Unit für FPGAs. Sie umfasst die Hardwarebeschreibung in VHDL und den Code für ein Board Support Package in C.

Die Architektur ist stark an die von Google beschriebene TPU angelehnt worden, der generelle Aufbau der verschiedenen Blöcke und ihre Verbindungen zueinander sind im Wesentlichen die selben, weshalb im folgenden vor allem auf die Unterschiede eingegangen wird.

Der Speicher für die Gewichte ist mit Dual-Port Block-RAM auf dem Chip realisert worden, anstatt auf einen Off-Chip DDR3-Speicher zuzugreifen. Das führt zu einer Erhöhung des Durchsatzes und zusätzlich entfällt der Bedarf für den Weight FIFO. Das Weight Memory ist direkt mit der Matrix Multiply Unit verdrahtet. Die Gewichte sind als 8-Bit-Festkommazahlen quantisiert.

Der Unified Buffer wird ebenfalls mit einem Block-RAM umgesetzt. Er besitzt drei Ports: Von einem kann nur gelesen werden, hier ist die Matrix Multiply Unit angeschlossen. Die Activation-Einheit schreibt ihre Ausgaben in den zweiten Port. Der letzte ist der Master-Port, welcher bei Benutzung die anderen beiden Ports deaktiviert. Über diesen Port schreibt und liest das Host-System indirekt die Ein- und Ausgangswerte des neuronalen Netzes. Diese sind, so wie die Gewichte, 8-Bit-Festkommazahlen.

Die Größe der Matrix Multiply Unit ist bei der tinyTPU frei konfigurierbar, standardmäßig werden Matrizen in der Größe 14×14 verrechnet. Eine Faltungsoperation wird nicht direkt unterstützt.

Die Akkumulatoren sind Register, in denen die 32 Bit breiten Ergbnisse der Matrizenmultiplikation zwischengespeichert werden. Hierfür wird wieder ein Block-RAM verwendet. Die aktuelle Instruktion gibt dabei an, ob die gespeicherten Werte mit den Akkumulator-Eingaben überschrieben werden, oder ob die Eingaben zu den vorhandenenn Werten hinzuaddiert werden.

Für die Aktivierung werden zwei Funktionen unterstützt: die ReLU-Funktion (Abschnitt 2.1.4), die durch ein Begrenzen des Wertebereichs realisiert ist, und die Sigmoid-Funktion, für die ein Look-Up-Table-RAM existiert. Bevor die per Instruktion gewählte Aktivierungsfunktion ausgeführt wird, werden die Werte gerundet und von 32-Bit auf 8-Bit skaliert.

Der Instruktionssatz umfasst sechs spezialisierte Instruktionen, die von den verschiedenen Steuerwerken ausgeführt werden.

Umgesetzt wurde tinyTPU auf einem Zynq 7020 System-on-a-Chip[20], welcher neben dem FPGA auch mit einer Dual-Core ARM Coretx-A9 CPU bestückt ist. Die Kommunikation zwischen der TPU und dem Host verläuft über eine AXI4-Lite-Schnittstelle (Advanced eXtensible Interface), definiert in der Advanced Microcontroller Bus Architecture von ARM[2].

4 Konzeption

Dieses Kapitel beschreibt die Erweiterungen zu den TPU-Architekturen, die in den Abschnitten 3.1 bis 3.2 vorgestellt wurden. Am Ende soll es möglich sein, Googles ImageNet-V1-Netzwerk[8] auf einem Koprozessor für eingebette Systeme umzusetzen.

4.1 Faltung durch Matrixmultiplikation

Die Faltungsoperation lässt sich mit einigen Anpassungen auf eine Matrixmultiplikation abbilden, welche dann von der Matrix Mulitply Unit berechnet werden kann. Dafür müssen die Faltungskerne und die Featuremaps in eine andere Form gebracht werden[17].

Dafür werden die Faltungskerne mit den Gewichten spaltenweise angeordnet. Dies geschiet bereits bei dem Export aus dem Trainigsmodell.

Es wird eine neue Komponente eingeführt, welche jeweils eine Featuremap in eine neue Matrix transformiert. Ein spaltenweise gleitendes Fenster in der Größe des Faltungskerns bestimmt den Ausschnitt der Featuremap, der ebenfalls spaltenweise in die Spalten der transformierten Featuremap übertragen wird.

Dann werden sie in die Systolic-Data-Setup-Komponente geladen, welche sie in die Matrix Multiply Unit führt. Nach der Matrixmultiplikation liegen die Ausgabe-Featuremaps spaltenweise in den Akkumulatoren vor.

4.2 Depthwise separable Convolutions

Die MobileNet-Architektur[8] verwendet für ihre Layer anstatt den regulären Faltungslayern (Abschnitt 2.1.2) solche mit sogenannten *depthwise separable Convolutions* („tiefenweise trennbare Faltungsoperationen“). Ihre Verwendung in neuronalen Netzwerken

wurde 2014 in [19] erstmals ausführlich beschrieben und getestet. Diese Layer setzen sich aus zwei hintereinander ausgeführten Operationen zusammen: *der depthwise Convolution* und der *pointwise Convolution*.

Ein Faltungslayer, der eine depthwise Convolution abbildet, führt die Faltungsoperation, separat für jede Eingangs-Featuremap aus (Abb. 4.1). Daraus folgt, dass ein depthwise Convolution-Layer die selbe Anzahl an Eingangs- und Ausgangs-Featuremaps besitzt. Damit bildet er die Funktion eines einfachen Filters ab.

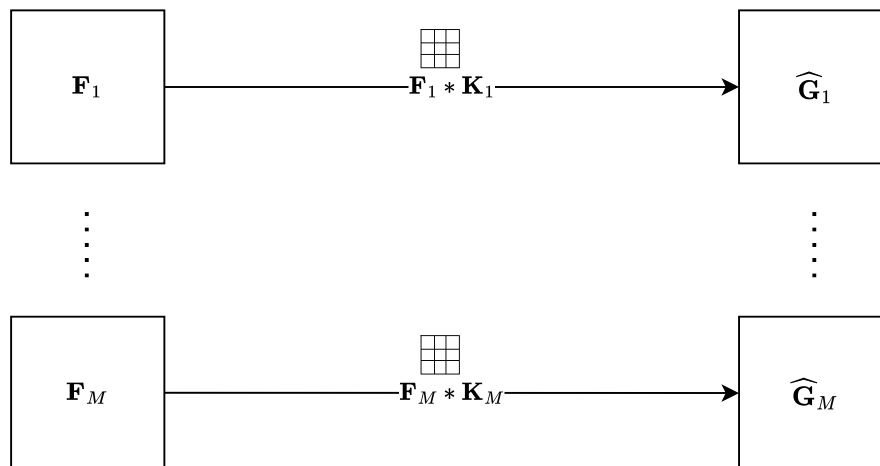

Abbildung 4.1: Eine depthwise Convolution mit M Eingaben, M Ausgaben und M Faltungskernen.

Eine pointwise Convolution beschreibt eine Faltung mit einem Filterkern der Größe 1×1. Pro Ausgangs-Featuremap verknüpft sie alle Eingangs-Featuremap miteinander und generiert dadurch neue FeaturesAbb. 4.2.

Die Faltungsoperationen werden wegen den Anpassungen in Abschnitt 4.1 weiterhin mit der Matrix Multiply Unit berechnet.

In den MobileNets werden die normalen Faltungen durch depthwise separable Convolutions ersetzt. Nach einer depthwise oder pointwise Convolution folgt immer jeweils ein Batch-Normalization-Layer und eine ReLU-Aktivierungsfunktion[8].

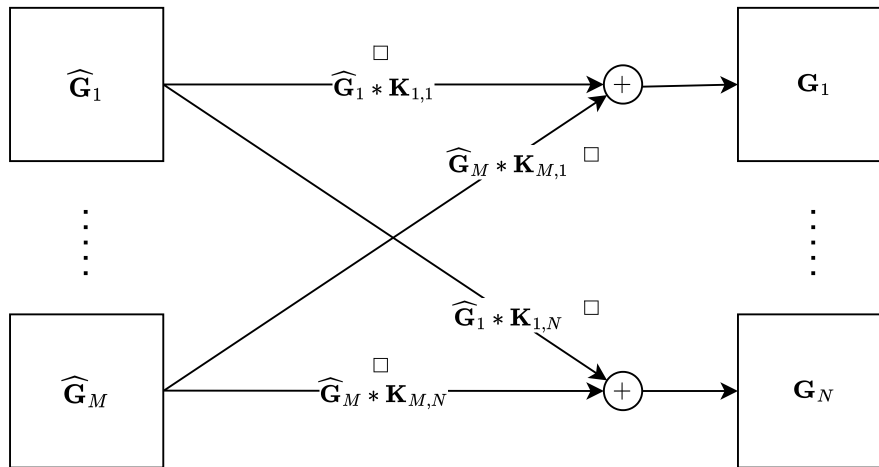

Abbildung 4.2: Eine pointwise Convolution mit M Eingaben, N Ausgaben und $M \times N$ Faltungskernen der Größe 1×1.

4.3 Akkumulatoren

Die vorhandenen Akkumulatoren werden um einen weiteren schreibenden Port ergänzt. Über diesen können die Werte, die von der Batch-Normalization-Komponente normalisiert wurden, wieder an ihre ursprüngliche Position geschrieben werden. Somit muss der Zugriff der Aktivierungs-Komponente auf die Ausgaben der Matrix-Multiply-Unit nicht erweitert werden.

4.4 Batch Normalization

Der Batch-Normalization-Layer (Abschnitt 2.1.3), welcher bei MobileNet zwischen den Layern für depthwise separable Convolutions und der ReLU-Aktivierungsfunktion existiert, wird als neuer Block in die TPU-Architektur eingefügt.

Die neue Batch-Normalization-Komponente bekommt als Eingänge die Ergebnisse der Matrix-Multiply-Unit aus den Akkumulatoren und die zugehörigen erlernten Skalierungsparameter γ, β sowie aus den Trainingsdaten ermittelten Mittelwert μ und Standardabweichung σ, allesamt als 32 Bit breite $Q16.16$-Festkommazahlen. Der Aufbau ist in Abb. 4.3 geschildert.

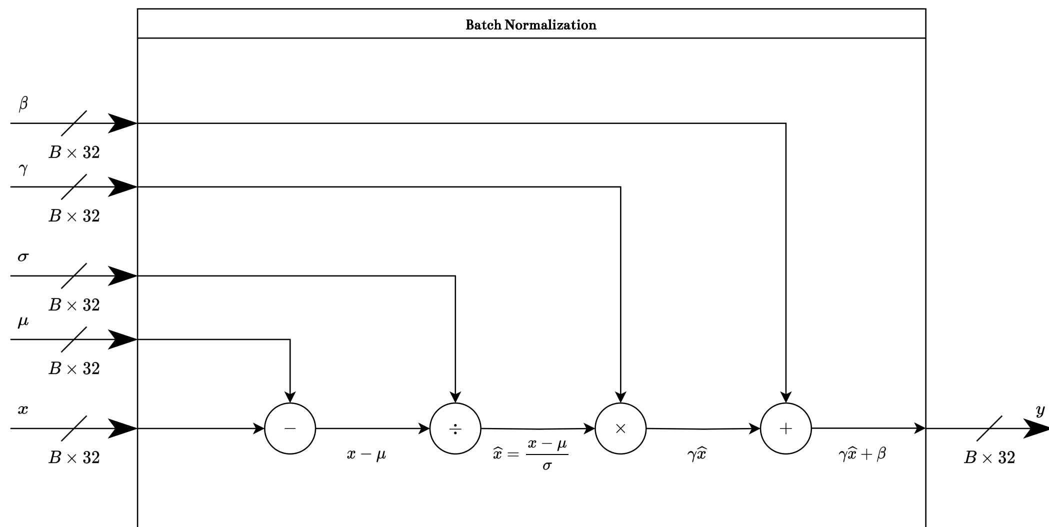

Abbildung 4.3: Innerer Aufbau der Batch-Normalization-Komponente. B bezeichnet die Busbreite

Ähnlich wie beim Weight Buffer wird für die Batch-Normalization-Parameter $\gamma, \beta, \mu, \sigma$ ein Speicher mit einer Breite von 4×32 Bit und einer zur Synthesezeit konfigurierbaren Tiefe als Block-RAM realisiert. Dieser kann während der Initialisierung vom Host-System über die AXI befüllt werden. Danach wird er nur von dem Batch-Normalization-Block gelesen, es reicht also ein Port. Ein Satz an Parametern einer Batch-Normalization wird nebeneinander gespeichert, sodass sie mit nur einer Adresse adressiert werden können.

Eine zusätzliche Steuereinheit dekodiert eine neu eingeführte Instruktion `batch_norm`, die die Adresse der Parameter, die Adresse der zu normalisierenden Werte in den Akkumulatoren und deren Anzahl enthält. Sie koordiniert den Zugriff auf die Speicher und steuert den Batch-Normalization-Block.

4.5 Softmax-Aktivierungsfunktion

Die softmax-Funktion (Abschnitt 2.1.4), welche als Aktivierungsfunktion in dem letzten Layer der Klassifikation des MobileNet[8] eingesetzt wird, wird in die vorhandene Activation-Einheit integriert.

Der Instruktionssatz der TPU-Architektur wird so angepasst, dass die `activate`-Instruktion neben den vorhandenen ReLU- und Sigmoid-Funktionen die Auswahl der softmax-Funktion erlaubt.

Für die Berechnung der Annäherung (siehe Abschnitt 5.2.1) wird das größte Element $\max_k(x_k)$ des Eingangsvektors benötigt. Dieses wird mit einer Baumstruktur aus Größer-Als-Vergleichen ermittelt. Sie besitzt eine asymptotische Laufzeit von $\mathcal{O}(\log n)$ und ist somit in weniger Taktzyklen durchlaufen als eine lineare Suche. In Abb. A.1 in Anhang A ist der Schaltplan aus der RTL-Analyse von Xilinx Vivado zu sehen.

Um die Exponentialfunktion mit den begrenzten Eingangswerten umzusetzen, wird das Ergebnes aus einer Look-Up-Tabelle geladen. Die Ausgabewerte sind im Festkommazahlenformat UQ0.8. Die folgenden Graphen zeigen die reellen Werte für e^x für $x \in [0, 1]$, die quantisierten Werte (Abb. 4.4) und die dadurch entstandenen absolutene Abweichung (Abb. 4.5).

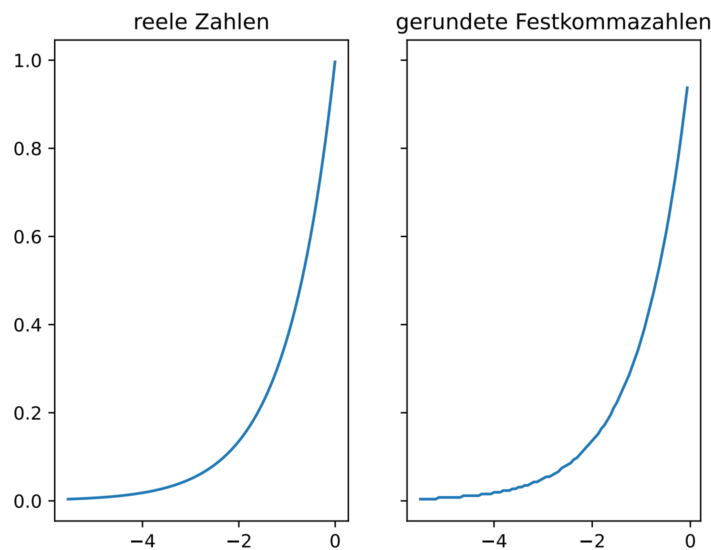

Abbildung 4.4: Die Exponentialfunktion e^x. Auf dem rechten Graphen lassen sich die Effekte der Quantisierung erkennen

4.6 Instruktionssatz

Der erweiterte Instruktionssatz der Tensor Processing Unit ist in Tabelle 4.1 angegeben.

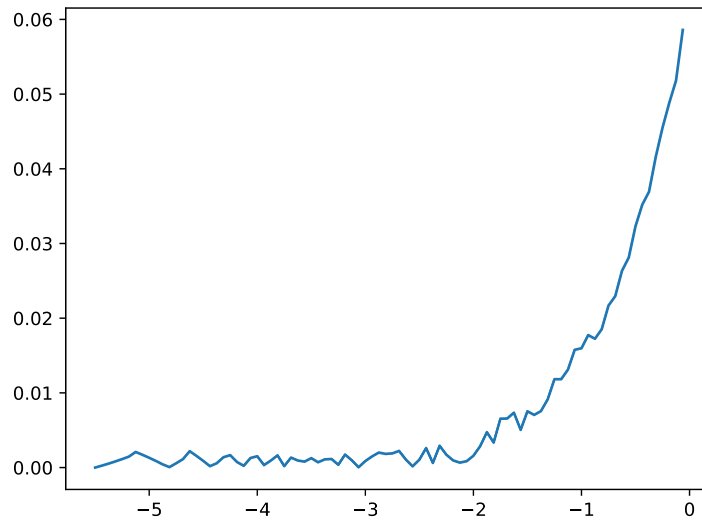

Abbildung 4.5: Die absolute Abweichung zwischen der reellen und der quantisierten Exponentialfunktion

Instruktion	OP-Code	24 Bit	16 Bit	32 Bit
`nop`	00000000	don't care		
`halt`	0000001x	don't care		
`read_weights`	00001xxx	Weight Buffer Address		Length
`matrix_multiply`	001xxxxx	Uniform Buffer Address	Accumulator Address	Length
`batch_norm`	01xxxxxx	BN-Parameter Buffer Address	Accumulator Address	Length
`activate`	1xxx1xxx	Uniform Buffer Address	Accumulator Address	Length
`synchronize`	11111111	don't care		

Tabelle 4.1: Erweiterter Instruktionssatz

5 Implementierung

In diesem Kapitel wird die Implementierung des Konzeptes aus Kapitel 4 vorgestellt. Es wird auf die verwendete Hardware und damit verbundene Entwicklungsumgebung eingegangen.

5.1 Verwendete Technologien

Die Implementierung wird entwickelt auf dem *Kria KR260 Robotics Starter Kit*[22] von Xilinx, Inc., einem Evaluation Kit für das umsetzen und testen von Anwendungen im Bereich der Robotik und Computer Vision. Es umfässt das *K26 System-on-a-Module* auf einer Trägerplatine, welches mit einem *Zynq UltraScale+ MPSoC*[21] ausgestattet ist. Dieser stellt unter anderem eine *Arm Coretx-A53* Accelerated Processing Unit (APU) und ein Field Programmable Gate Array zu Verfügung.

Als Entwicklungsumgebung wird *Vivado Design Suite*[23] verwendet. Dieses ebenfalls von Xilinx, Inc. bereitgestellte Komplettlösung wird für den gesamten Prozess der Hardwaremodellierung eingesetzt. Mit ihr wird der VHDL-Code geschrieben und synthetisiert.

Die Konfiguration der Hardware wird in einem Block-Design (Abb. 5.1, vergrößerte in Abb. A.2 dargestellt) zusammengefasst. Darin sind die verwendeten Intellectual-Property-Blöcke (auch IP Cores gennant) und deren Verbindungen spezifiziert. Die erweiterte TPU ist darin ebenfalls als selbst-erstellter IP-Block (Custom IP-Core) zu finden.

5.2 Anpassung an die FPGA-Umgebung

Oft können bekannte Algorithmen nicht so direkt wie in Software auf Hardware realisiert werden. Dieser Abschnitt beschreibt die Alternativen oder angepasste Algorithmen, die stattdessen umgesetzt werden.

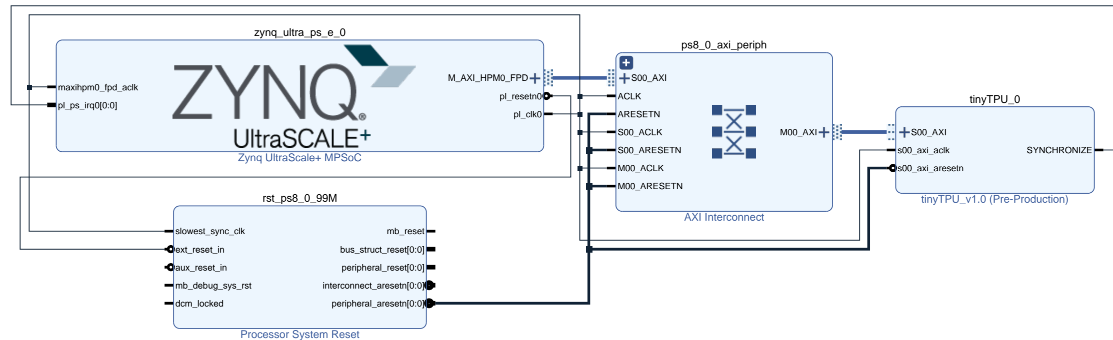

Abbildung 5.1: Block-Design mit Zynq UltraScale+ MPSoC und Custom IP

5.2.1 Softmax-Aktivierungsfunktion

Die softmax-Funktion (Abschnitt 2.1.4) ist wegen der Exponentialkomponente und der Division durch möglicherweise große Zahlen nicht trivial in Hardware umzusetzen. Die Verwendung von Festkommazahlen schränkt den Wertebereich von Zwischenergebnissen ein, weshalb sie in einigen Fällen über-/unterlaufen könnten.

Aus diesem Grund gibt es verschiedene Annäherungen und alternative Funktionen[3], von denen folgende[14] hier umgesetzt wird:

$$\hat{f}(\vec{x})_i = \mathrm{e}^{x_i - \max_k(x_k)} \tag{5.1}$$

Durch die Verwendung als Aktivierungsfunktion ergibt sich an die Annäherung die Anforderung, dass die vom Ausganslayer des CNNs ausgewählte Klasse diesselbe wie bei der echten softmax-Funktion ist. Seien also $q = \mathrm{argmax}_x(f(\vec{x})_i)$ und $r = \mathrm{argmax}_x(\hat{f}(\vec{x})_i)$ die Indizes der jeweils ausgewählten Klassen, dann muss $q = r$ gelten:

$$\begin{gathered} \max(f(\vec{x})_i) = \max\left(\frac{\mathrm{e}^{x_q}}{\sum_k^K \mathrm{e}^{x_k}}\right) \implies \\ x_q = \max_k(x_k) \end{gathered} \tag{5.2}$$

$$\begin{gathered} \max(\hat{f}(\vec{x})_i) = \max\left(\mathrm{e}^{x_r - \max_k(x_k)}\right) \implies \\ x_r = \max_k(x_k) \end{gathered} \tag{5.3}$$

Da $x_q = x_r$, gilt $\mathrm{argmax}_x(f(x_q)) = \mathrm{argmax}_x(\hat{f}(x_r)) \implies q = r$.

Es sei angemerkt, dass das Abbild der vorgestellten Annäherung $\hat{f}(\vec{x})_i$ im Gegensatz zu dem Abbild der echten softmax-Funktion keine Wahrscheinlichkeitsfunktion darstellt, da $\sum_i \hat{f}(\vec{x})_i > 1$. Dies ist aufgrund der spezifischen Anforderungen aber zu vernachlässigen.

Die in der Annäherung verwendete Exponentialkomponente lässt sich einfacher als eine allgemeine Exponentialfunktion umsetzen, da die Wertebereiche der Ein- und Ausgabe beschränkt sind:

$$\begin{aligned} x_i \leq \max_k(x_k) &\implies \\ x_i - \max_k(x_k) \leq 0 &\implies \\ \mathrm{e}^{x_i - \max_k(x_k)} &\leq 1. \end{aligned} \tag{5.4}$$

6 Evaluation

Im folgenden werden die in den vorherigen Kapiteln aufgeführten Konzepte untersucht, ob sie eine Verbesserung hinsichtlich der Performance erreichen können.

6.1 Depthwise seperable Convolutions

Die Faltungsoperationen eines regulären Faltungslayers (Abschnitt 2.1.2) benötigen folgende Anzahl an Multiplikationen:

$$D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F \tag{6.1}$$

Diese Berechnungskosten setzen sich zusammen aus der Größe des Faltungskerns $D_K \times D_K$, der Anzahl der Eingangs-Featuremaps M, die Anzahl der Ausgangs-Featuremaps N und der Größe der Eingangs-Featuremap $D_F \cdot D_F$.

Die Berechnungskosten der depthwise separable Convolutions (Abschnitt 4.2) des MobileNet hingegen setzen sich zusammen aus den depthwise Convolutions und den pointwise Convolutions und ergeben in Summe[8]:

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F \tag{6.2}$$

Die depthwise Convolutions werden im Gegensatz zu den regulären Faltungslayerns nicht pro Neuron und Eingangs-Featuremap ausgeführt, sondern lediglich pro Eingangs-Featuremap. Die pointwise Convolutions werden wie reguläre Faltungen berechnet, allerdings mit einem Faltungskern der Größe 1×1, somit entfällt der Term $D_K \times D_K$ in den Berechnungskosten.

Setzt man beide Berechnungskosten ins Verhältnis – Gleichung (6.2) ÷ Gleichung (6.1) – ergibt sich der Leistungsunterschied:

$$\frac{D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F}{D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F} = \frac{1}{{D_K}^2} + \frac{1}{N} \tag{6.3}$$

Bei Verwendung von depthwise Convolutions mit einer Faltungskern-Größe von 3×3, wie es bei MobileNet der Fall ist, benötigen die depthwise separable Convolutions nur ca. $1/9$ der Berechnungszeit der regulären Faltungsoperationen.

Im Gegenzug sinkt dafür die Genauigkeit der Klassifikation, allerdings nur geringfügig. In Tabelle 6.1 ist zu erkennen, dass bei zwei gleichen MobileNet-Netzwerken, welche sich nur in der Operation der Faltungslayer unterscheiden, die Genauigkeit der depthwise separable Convolutions nur ca. 1 % verliert. Auch die Parameteranzahl ist niedriger, was diese Art von Netzwerken besonders geeignet für Geräte mit wenig Speicherkapazität macht.

Faltungslayer-Operation	Genauigkeit	Operationen	Parameter
reguläre Faltung	71,7 %	$4866 \cdot 10^6$	$29{,}3 \cdot 10^6$
depthwise separable Convolution	70,6 %	$569 \cdot 10^6$	$4{,}2 \cdot 10^6$

Tabelle 6.1: Leistungsvergleich von zwei MobileNet-Netzwerken[8]

7 Zusammenfassung und Fazit

Im Rahmen dieser Arbeit wurde gezeigt, wie ein möglicher Koprozessor für die Beschleunigung maschinellen Lernens konzipiert werden kann. Dabei wurde eine vorhandene Tensor-Processing-Unit-Architektur aufgegriffen, welche für die Ausführung auf einem Field Programmable Gate Array eine solide Grundlage bildet[4]. Um die Inferenz mit Faltungsnetzwerken am Beispiel von MobileNet weiter für den Anwendungsfall eines ernergieeffizienten eingebetteten Systems zu optmieren, wurde eine Reihe von Erweiterungen und Änderungen an tinyTPU dargestellt.

Ein Teil dieser Konzepte wurde implementiert und in der Simulation getestet. Auf Grund einer fehlenden Gesamtumsetzung auf der Testhardware, konnte keine allumfassende Auswertung stattfinden. Trotzdem wurde im Evaluationskapitel ein theoretischer Leistungsvorteil aufgezeigt. Dieser liegt in einer erheblichen Reduktion der Berechnungskosten (Zeit und Hardwareelemente) bei nur geringfügigem Verlust an Klassifikationsgenauigkeit.

7.1 Ausblick auf Weiterentwicklung

In diesen Abschnitt werden Ideen für Weiterentwicklungen und mögliche Verbesserung der Architektur vorgestellt.

7.1.1 Verschiedene Komponentengrößen

Eine Möglichkeit zur besseren Nutzung der FPGA-Ressourcen könnte die Aufteilung der einzelnen TPU-Komponenten in unterschiedliche Größen sein. Für die späteren Layer tiefer Netze, in denen eine höhere Anzahl kleinerer Featuremaps existiert, könnte gegebenenfalls mit mehreren kleinen Matrix Mulitply Units ein höherer Grad der Parallelität erreicht werden.

7.1.2 Unterstützung von dünnbesetzten Matrizen

Auf leistungsfähigeren FPGAs, auf denen die Größe der Matrx Multiply Unit erhöht werden soll, mag eine Erwägung der Umsetzung von dünnbesetzten Matrizen sinvoll sein[13]. So könnte die Performance noch weiter optimiert werden, indem bei der Berechnung der Matrixmultiplikation die Elemente mit dem Wert Null nicht berücksichtigt werden.

7.1.3 Faltungsoperation mit Stride-Parameter

Die entworfene Faltungsoperation rechnet immer mit einem Stride-Parameter von 1. Neuronale Netze nutzen Layer mit höheren Stride-Werten, wenn die Ausgangs-Featuremap zusätzlich zur Faltung verkleinert werden soll. Ist das Ziel, dass die Tensor Processsing Unit diese neuronalen Netze zusätzlich unterstützt, dann kann die aktuelle Faltungsoperation um einen Stride-Parameter erweitert werden.

Literaturverzeichnis

[1] Arm Limited: Q-format. In: *RealView Development Suite AXD and armsd Debuggers Guide*. Version 3.0. Arm Limited, März 2006, Kap. 4.7.9. – URL https://developer.arm.com/documentation/dui0066/g. – Zugriffsdatum: 2023-07-08

[2] Arm Limited: AMBA AXI4-Lite Interface Specification. In: *AMBA AXI and ACE Protocol Specification*. Issue H.c. Arm Limited, Januar 2021, Kap. B, S. 119–126. – URL https://developer.arm.com/documentation/ihi0022/hc. – Zugriffsdatum: 2023-07-08

[3] Cardarilli, Gian C. ; Di Nunzio, Luca ; Fazzolari, Rocco ; Giardino, Daniele ; Nannarelli, Alberto ; Re, Marco ; Spanò, Sergio: A pseudo-softmax function for hardware-based high speed image classification. In: *Scientific Reports* 11 (2021), Juli, Nr. 1, S. 15307. – URL https://doi.org/10.1038/s41598-021-94691-7. – ISSN 2045-2322

[4] Fuhrmann, Jonas: *Implementierung einer Tensor Processing Unit mit dem Fokus auf Embedded Systems und das Internet of Things*. 2018. – URL http://hdl.handle.net/20.500.12738/8527

[5] Glorot, Xavier ; Bordes, Antoine ; Bengio, Yoshua: Deep Sparse Rectifier Neural Networks. In: Gordon, Geoffrey (Hrsg.) ; Dunson, David (Hrsg.) ; Dudík, Miroslav (Hrsg.): *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* Bd. 15. Fort Lauderdale, FL, USA : PMLR, April 2011, S. 315–323. – URL https://proceedings.mlr.press/v15/glorot11a.html

[6] Goodfellow, Ian ; Bengio, Yoshua ; Courville, Aaron: *Deep Learning*. MIT Press, 2016. – URL http://www.deeplearningbook.org

[7] Hahnloser, Richard H. R. ; Sarpeshkar, Rahul ; Mahowald, Misha A. ; Douglas, Rodney J. ; Seung, H. S.: Digital selection and analogue amplification coexist

in a cortex-inspired silicon circuit. In: *Nature* 405 (2000), Juni, Nr. 6789, S. 947–951. – URL https://doi.org/10.1038/35016072. – ISSN 1476-4687

[8] Howard, Andrew G. ; Zhu, Menglong ; Chen, Bo ; Kalenichenko, Dmitry ; Wang, Weijun ; Weyand, Tobias ; Andreetto, Marco ; Adam, Hartwig: MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. In: *CoRR* abs/1704.04861 (2017), April. – URL http://arxiv.org/abs/1704.04861

[9] Hyvarinen, A.: Fast and robust fixed-point algorithms for independent component analysis. In: *IEEE Transactions on Neural Networks* 10 (1999), Mai, Nr. 3, S. 626–634. – ISSN 1941-0093

[10] IEEE Standard for Floating-Point Arithmetic. In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019)

[11] IEEE Standard for VHDL Language Reference Manual. In: *IEEE Std 1076-2019* (2019)

[12] Ioffe, Sergey ; Szegedy, Christian: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In: *CoRR* abs/1502.03167 (2015), März. – URL http://arxiv.org/abs/1502.03167

[13] Jouppi, Norman P. ; Young, Cliff ; Patil, Nishant ; Patterson, David A. ; Agrawal, Gaurav ; Bajwa, Raminder ; Bates, Sarah ; Bhatia, Suresh ; Boden, Nan ; Borchers, Al ; Boyle, Rick ; Cantin, Pierre-luc ; Chao, Clifford ; Clark, Chris ; Coriell, Jeremy ; Daley, Mike ; Dau, Matt ; Dean, Jeffrey ; Gelb, Ben ; Ghaemmaghami, Tara V. ; Gottipati, Rajendra ; Gulland, William ; Hagmann, Robert ; Ho, C. R. ; Hogberg, Doug ; Hu, John ; Hundt, Robert ; Hurt, Dan ; Ibarz, Julian ; Jaffey, Aaron ; Jaworski, Alek ; Kaplan, Alexander ; Khaitan, Harshit ; Koch, Andy ; Kumar, Naveen ; Lacy, Steve ; Laudon, James ; Law, James ; Le, Diemthu ; Leary, Chris ; Liu, Zhuyuan ; Lucke, Kyle ; Lundin, Alan ; MacKean, Gordon ; Maggiore, Adriana ; Mahony, Maire ; Miller, Kieran ; Nagarajan, Rahul ; Narayanaswami, Ravi ; Ni, Ray ; Nix, Kathy ; Norrie, Thomas ; Omernick, Mark ; Penukonda, Narayana ; Phelps, Andy ; Ross, Jonathan ; Salek, Amir ; Samadiani, Emad ; Severn, Chris ; Sizikov, Gregory ; Snelham, Matthew ; Souter, Jed ; Steinberg, Dan ; Swing, Andy ; Tan, Mercedes ; Thorson, Gregory ; Tian, Bo ; Toma, Horia ; Tuttle, Erick ; Vasudevan, Vijay ; Walter, Richard ; Wang, Walter ; Wilcox, Eric ; Yoon,

Doe H.: In-Datacenter Performance Analysis of a Tensor Processing Unit. In: *CoRR* abs/1704.04760 (2017). – URL http://arxiv.org/abs/1704.04760

[14] Kouretas, Ioannis ; Paliouras, Vassilis: Hardware Implementation of a Softmax-Like Function for Deep Learning. In: *Technologies* 8 (2020), August, Nr. 3. – URL https://www.mdpi.com/2227-7080/8/3/46. – ISSN 2227-7080

[15] LeCun, Y. ; Boser, B. ; Denker, J. S. ; Henderson, D. ; Howard, R. E. ; Hubbard, W. ; Jackel, L. D.: Backpropagation Applied to Handwritten Zip Code Recognition. In: *Neural Computation* 1 (1989), Dezember, Nr. 4, S. 541–551. – URL https://doi.org/10.1162/neco.1989.1.4.541. – ISSN 0899-7667

[16] Mcculloch, Warren ; Pitts, Walter: A Logical Calculus of Ideas Immanent in Nervous Activity. In: *Bulletin of Mathematical Biophysics* 5 (1943), S. 127–147

[17] Meisel, Andreas: *Mustererkennung und Machine Learning (Vorlesungsmanuskript)*. Januar 2021. – Hochschule für Angewandte Wissenschaften Hamburg

[18] Nielsen, Michael A.: *Neural Networks and Deep Learning*. Determination Press, 2015. – URL http://neuralnetworksanddeeplearning.com/. – Zugriffsdatum: 2023-07-08

[19] Sifre, Laurent ; Mallat, Stéphane: Rigid-Motion Scattering for Texture Classification. In: *CoRR* abs/1403.1687 (2014), März. – URL http://arxiv.org/abs/1403.1687

[20] Xilinx, Inc.: *Zynq-7000 SoC Data Sheet: Overview (DS190)*. v1.11.1. San Jose, California, U.S.A.: , Juli 2018. – URL https://docs.xilinx.com/v/u/en-US/ds190-Zynq-7000-Overview. – Zugriffsdatum: 2023-07-08

[21] Xilinx, Inc.: *Zynq UltraScale+ MPSoC Data Sheet: Overview (DS891)*. v1.9. San Jose, California, U.S.A.: , Mai 2020. – URL https://docs.xilinx.com/v/u/en-US/ds891-zynq-ultrascale-plus-overview. – Zugriffsdatum: 2023-07-08

[22] Xilinx, Inc.: *Kria KR260 Robotics Starter Kit Data Sheet (DS988)*. v1.0. San Jose, California, U.S.A.: , Mai 2022. – URL https://docs.xilinx.com/r/en-US/ds988-kr260-starter-kit. – Zugriffsdatum: 2023-07-08

[23] Xilinx, Inc.: *Vivado Design Suite User Guide: Synthesis (UG901)*. 2023.1. San Jose, California, U.S.A.: , Juni 2023. – URL https://docs.xilinx.com/r/en-US/ug901-vivado-synthesis. – Zugriffsdatum: 2023-07-08

A Abbildungen

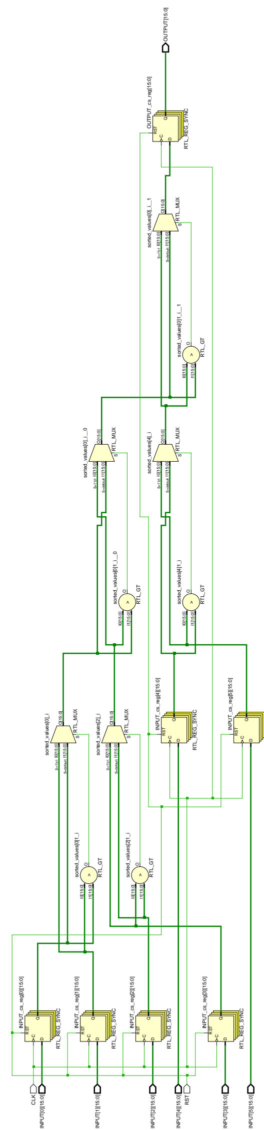

Abbildung A.1: Schaltplan der max-Baumstruktur aus der RTL-Analyse von Xilinx Vivado

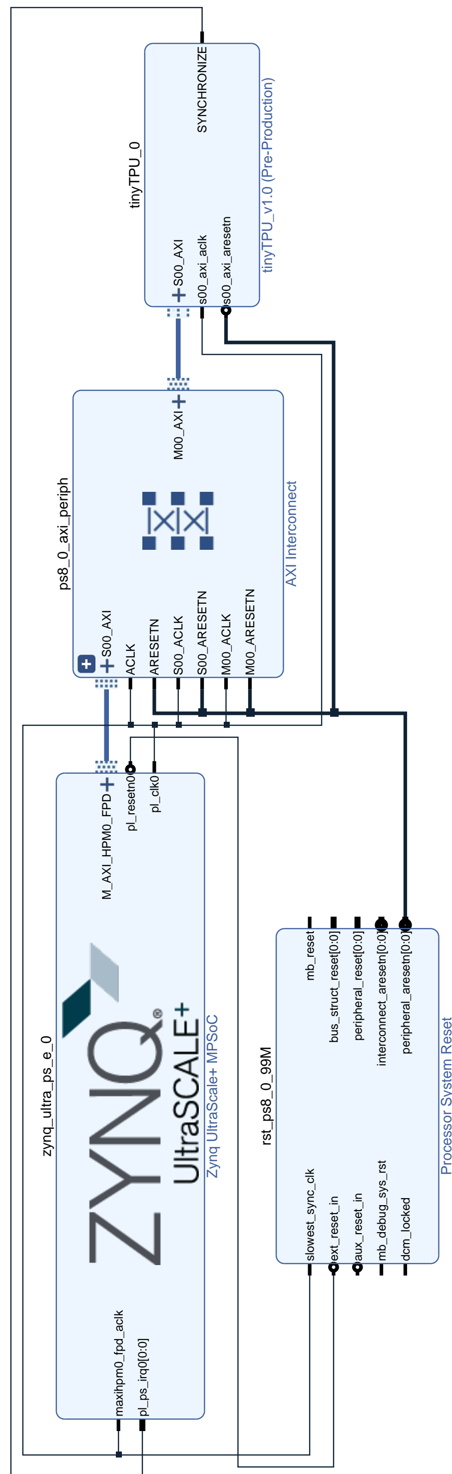

Abbildung A.2: Block-Design mit Zynq UltraScale+ MPSoC und Custom IP (vergrößert)

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original