Bericht zum Projekt
„Neuronale Netze auf strahlungstoleranten FPGAs für die Raumfahrt“

Institution
Hochschule Bremen
Flughafenallee 10
28199 Bremen

Autoren
Colin von Huth
Marvin Soldin
Ngwa Tingoh Kingsley
Sebastian von Minden
Tim Wieborg

Betreuer
Prof. Dr. Jan Bredereke

Semester
Wintersemester 2019/2020

14. Februar 2020

Zusammenfassung

Autor: Tim Wieborg, Colin von Huth

In diesem Projektbericht wird untersucht, ob es möglich ist, neuronale Netze auf leistungsschwachen, beziehungsweise strahlungstoleranten FPGAs auszuführen. Solche FPGAs werden im Bereich der Raumfahrt verwendet. Dazu werden verschiedene Werkzeuge zur Entwicklung von neuronalen Netzen auf FPGAs untersucht und evaluiert. Anschließend wird eine geeignete Toolchain entworfen, mit der es möglich ist, neuronale Netze auf FPGAs zu implementieren. Mit Hilfe eines Python-Skripts können sowohl die Eingaben zum neuronalen Netz gesendet, als auch die Ausgaben abgefragt werden. Machbarkeit und Funktionalität der entworfenen Toolchain werden anhand eines Beispiel-Projekts bewiesen.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Autor: Jan Bredereke

Onboard-Rechner von Raumfahrzeugen sind für die grundlegende Steuerung zuständig, aber auch für die Datenverarbeitung für die Nutzlasten. Die Steuerungsaufgaben sind dabei höchst sicherheitsrelevant, da zum Beispiel ein Satellit oft hunderte von Millionen Euro kostet und nicht verloren gehen soll. Aufgrund der hohen Belastung durch Weltraumstrahlung würden übliche heutige Prozessoren sehr schnell ausfallen. Daher verwendet man Spezialrechner, deren Chips Strukturbreiten von mindestens 65 nm aufweisen. Diese Spezialrechner sind robust genug, aber entsprechend weniger leistungsfähig als solche, die mit aktuellen 16 nm oder 10 nm hergestellt sind. Aufgrund der äußerst geringen Stückzahlen dieser Art von Rechnern werden sie oft nicht mit speziell entwickelten Chips (ASICs), sondern mit programmierbarer Standard-Hardware (FPGAs) realisiert. Strahlungsfeste Versionen bestimmter FPGAs mit entsprechend großer Strukturbreite sind hierfür verfügbar.

Zunehmend besteht Bedarf an mehr Onboard-Rechenleistung, zum Beispiel für Bildverarbeitung an Bord, etwa für autonome Rover auf anderen Himmelskörpern oder für Schwärme von Kleinsatelliten mit jeweils nur wenig Bandbreite zu einer Bodenstation. Neuronale Netze sind ein Verfahren, das z.B. das autonome Klassifizieren von Bildern ermöglicht. Allerdings benötigen sie sehr viel Rechenleistung und werden daher oft auf besonders leistungsfähiger Spezialhardware ausgeführt. Diese ist für die Weltraumumgebung leider völlig ungeeignet.

Das Ausführen eines bereits trainierten neuronalen Netzes benötigt Größenordnungen weniger an Rechenleistung, als es zu trainieren. Indem das Trainieren vorab am Boden erledigt wird, rücken jedenfalls einige Anwendungen der Bildverarbeitung in greifbarere Nähe. Weiterhin ist das Ausführen eines neuronalenNetzes eine inhärent hochparallele Aufgabe, für die ein FPGA sehr gut geeignet ist. Dazu kommt außerdem, dass ein neuronales Netz bei einer Bildverarbeitungsaufgabe eine gewisse Rate an transienten Fehlern (wie sie durch Weltraumstrahlung entstehen) ohne wesentliche Minderung des Ergebnisses erträgt. Während eine sequenziell arbeitende CPU durch ein einzelnes Störsignal in vielen Fällen komplett aus dem Tritt gebracht wird, bleibt eine Störung in einem neuronalen Netz oft nur lokal und beeinträchtigt die Qualität des Gesamtergebnisses dann nur graduell.

Bei der Datenverarbeitung für die Nutzlasten sind die Zuverlässigkeitsanforderungen oft nicht ganz so hoch wie für die grundlegende Steuerung. Hier werden daher statt strahlungsfester Chips zum Teil auch nur strahlungstolerante Chips eingesetzt. Diese erleiden zwar häufiger mal kurzzeitige Ausfälle, haben aber eine im Vergleich höhere Rechenleistung.

Aufgabenstellung

Ziel dieses Projekts soll es sein zu erkunden, inwieweit sich trainierte neuronale Netze auf einem FPGA ausführen lassen, dessen Leistungsfähigkeit dem eines strahlungstoleranten FPGAs entspricht. Das Spannungsfeld zwischen der nötigen Rechenleistung und der Rechenkapazität der geeigneten FPGAs soll ausgelotet werden. Dabei sind viele Ansätze zur Optimierung denkbar. Die schon genannte Senkung der Zuverlässigkeit einzelner Neuronen durch eine eigentlich zu kleine Strukturbreite ist nur eine davon.

Da neuronale Netze im Embedded-Bereich zunehmend an Bedeutung gewinnen, zum Beispiel auf Smartphones, gibt es bereits etliche weitere Ansätze, um trainierte neuronale Netze auf vergleichsweise rechenschwachen CPUs auszuführen.

Im Projekt soll zunächst ein möglichst einfaches neuronales Netz auf einem FPGA der genannten Klasse ausgeführt werden, als „Hello world“, um ein grundlegendes Verständnis von neuronalen Netzen zu erreichen, und um eine Toolchain zur Verfügung zu haben. Im allerersten Schritt reichen als Eingabemedium dafür die Schiebeschalter eines Experimental-FPGA-Boards.

Anschließend sollen die Teilnehmer das Thema je nach Neigung weiter vertiefen und dabei die Anwendung ausbauen. Hierfür sind z.B. ein schneller Eingabedatenstrom, leistungsfähigere Hardware und strukturelle Optimierungen der neuronalen Netze interessant.

Statt des Klassifizierens von Bildern können erst einmal auch andere, einfachere Aufgaben angegangen werden. Gute Ideen für anderes autonomes Klassifizieren oder für autonomes Regeln an Bord von Satelliten oder Rovern hätten sogar einen ganz eigenen Wert als Projektergebnis.

Eine mögliche einfache Aufgabe für den Anfang wäre das Unterscheiden einiger weniger kurzer Tonsignale. Ein denkbarer Einsatz für eine (ganz wesentlich) erweiterte Version wäre das Erkennen kritischer Veränderungen an der mechanischen Struktur des Satelliten über ein Mikrofon, z.B. eines beginnenden Defekts eines Nachstellmotors an den Solarzellen, eines beginnenden Defekts an einem Gyroskop (motorbetriebener Kreisel zur Lageregelung), eines Einschlags eines Mikrometeoriten usw.

Kapitel 2

Grundlagen

2.1 Auswahl der Hardware

Autor: Ngwa Tingoh Kingsley

Für das Projekt ist die Auswahl der korrekten Hardware sehr wichtig, da diese den speziellen Vorgaben zur Strahlungsresistenz und -toleranz nachkommen muss. Dieses Kapitel zeigt die Rechercheergebnisse zu einigen der wichtigsten technischen Aspekte, welche für das Projekt relevant sind. Dazu wird im Folgenden kurz erläutert, was Field Programmable Gate Array (FPGAs) sind. Anschließend werden einige für das Projekt relevante FPGAs und System-on-Chips (SoCs) vorgestellt und verglichen, um die Entscheidung für das im Projekt verwendete Board nachvollziehbar zu machen.

2.1.1 Field Programmable Gate Array (FPGA)

Ein Field Programmable Gate Array (FPGA) ist ein integrierter Schaltkreis (IC) der Digitaltechnik, auf dem eine logische Schaltung programmiert werden kann. Auf einem FPGA können nicht nur simple Schaltnetze, sondern auch sehr komplexe Schaltwerke aufgebaut werden. Darüber hinaus können FPGAs meist mehrfach rekonfiguriert werden. Der Konfigurationsvorgang beginnt mit einer Hardwarebeschreibungssprache (HDL), einer bestimmten digitalen Schaltung, die zu einem Bitstream kompiliert und auf das FPGA heruntergeladen wird [Cro+19, S.18 ff].

FPGA Architektur

Die grundlegende Architektur eines FPGAs besteht aus einer zweidimensionalen Array von einfachen digitalen Logikelementen, die in sogenannten konfigurierbaren Logikblöcken (CLBs) gruppiert sind. Jedes CLB besteht aus einer kleinen Anzahl von Flip-Flops (FFs) und Lookup-Tabellen (LUTs), wobei die LUTs in der Lage sind, boolesche Logikfunktionen sowie kleine Speicher und Schieberegister zu implementieren.

Ein CLB der Xilinx Serie 7 enthält beispielsweise acht LUTs mit 6 Eingängen, 16 FFs, 2 arithmetischen Carry-Logiken und Multiplexern [XIL16, S.17]. Einige CLBs enthalten Elemente, die die Fähigkeit zur Implementierung von Schieberegistern und distributed RAM unterstützen. Die CLBs sind mit programmierbaren Interconnects und Schaltmatrizen miteinander verbunden. Die Array-Struktur von CLBs, Schaltmatrizen und programmierbaren Interconnects ist am unteren Rand von Abbildung 2.1 auf der nächsten Seite ersichtlich.

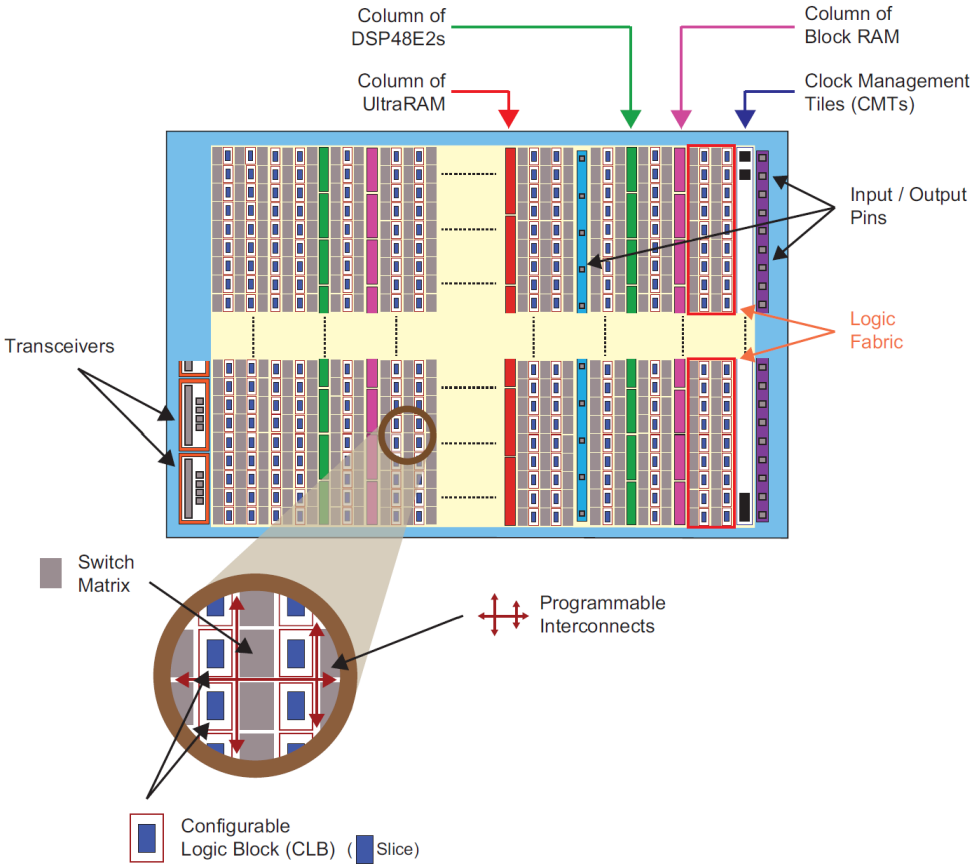

Abbildung 2.1: Beispiel für ein Ressourcen-Layout eines UltraScale+ [Cro+19, S.21]

.

Abbildung 2.1 auf der vorherigen Seite ist ein Beispiel für ein Ressourcen-Layout eines UltraScale+ FPGAs. Auf einer hohen Abstraktionsebene umfasst das FPGA-Baustein-Layout vertikale Bereiche, die verschiedene Arten von Ressourcen enthalten. Der Großteil des Bausteins ist mit allgemeiner Logik ausgestattet, d.h. CLBs, die hauptsächlich aus LUTs und FFs bestehen. Block-RAM- und Ultra-RAM-Speicherblöcke sowie DSP48x-Arithmetikscheiben sind in einzelnen oder doppelten Spalten auf dem Baustein angeordnet und bilden dünne vertikale Streifen.

Hinsichtlich der Anbindung sind die Input/Output-Blöcke (IOBs) in Gruppen gebündelt und in Spalten innerhalb der Hauptressourcen angeordnet. Die IOBs können eine Vielzahl von Schnittstellenstandards unterstützen. Zusätzliche Konnektivität wird in Form von schnellen seriellen Sendern/Empfängern bereitgestellt, die sich normalerweise in Vierergruppen an den Rändern des FPGAs befinden. In der Nähe dieser Blöcke sind separate Blöcke zur Unterstützung ausgewählter Kommunikationsstandards vorgesehen. In der FPGA-Struktur sind auch zusätzliche Ressourcen für die Konfiguration, das Taktmanagement und die Systemüberwachung vorhanden [Cro+19, S.22].

2.1.2 System-On-Chip (SoC)

FPGAs, die einen Prozessor und eine FPGA-Struktur auf einem Chip kombinieren, bieten eine einzigartige Alternative, um den vielfältigen Anforderungen der Entwickler nachzukommen. Diese werden System-on-Chip genannt[Cro+19, S.2].

SoC hat auch die Integration anderer Bausteine erlebt, sodass FPGA-Hersteller jetzt Bausteine herstellen, die nicht nur aus programmierbarer Logik bestehen, sondern aus programmierbarer Logik in Kombination mit Prozessoren, Speichern, Schnittstellen und weiteren Bausteinen.

Intellectual Property Core (IP-Core)

Die meisten SoCs werden anhand eines Intellectual Property Core konzipiert [Cro+19, S.18 ff]. Mit einem IP-Core wird das geistigte Eigentum des Enwicklers vom Modul bezeichnet. IP-Cores sind wiederverwendbare Hardwarebausteine und realisieren eine bestimmte Aufgabe. Dabei werden IP-Cores in Soft und Hard IP-Core Kategorien unterteilt.

1. Soft IP-Cores sind mit einer Hardware-Beschreibungssprache programmiert und in einer Bibliothek als Quelltext oder als vorsynthetisierte Netzliste verfügbar.
2. Hard IP-Cores sind bereits als individuelle Chips oder in sogenannten SoC-FPGAs im gleichen Silizium-Chip des FPGAs integriert.

2.1.3 Relevante FPGAs und SoCs

In diesem Unterkapitel werden eine Auswahl an strahlungsharten FPGAs, SoCs und normalen kommerziellen Field Programmable Gate Arrays kurz vorgestellt und verglichen, um ein für unser Projekt sinnvolles FPGA oder SoC zu finden.

Virtex-5QV

Der FPGA-Hersteller Xilinx bietet für Weltraumanwendungen das strahlungsharte FPGA Virtex-5QV an. Es ist in 65nm-Technologie gefertigt und nach eigener Werbung das einzige industriell verfügbare strahlungsharte FPGA. Es wird zusätzlich die besondere Silicon-On-Insulator (SOI) Technologie zum Schutz gegen „Durchbrennen“ durch Weltraumstrahlung verwendet. Im Vergleich dazu werden aktuelle FPGAs von Xilinx in 16 nm-Technologie gefertigt [Bre19, S.1].

7 Series

Im Jahr 2011 wurde die FPGA-Serie 7 [XIL18a, S.1] vorgestellt, die auf einer 28 nm High-K Metal Gate (HKMG)-Prozesstechnologie basiert. Die Serie ist in vier Familien unterteilt: Artix-7, Kintex-7, Spartan-7 und Virtex-7. Alle FPGAs der Serie 7 beinhalten unter anderem Logik-Zellen, Block-RAMs, DSP (Digital Signal Processing) Slices, Takt-Steuereinheiten (Clock Management Tiles, CMT) und Blöcke für PCI-Express. In FPGAs der Serie 7 existieren in jeder Slice vier voneinander unabhängige LUTs (A, B, C und D) mit sechs Eingängen (A1 bis A6) und zwei Ausgängen (O5 und O6). Neben Logikblöcken existieren in den FPGAs der Serie 7 eine Reihe von Block- RAMs bis zur einer Größe von 36Kb. Dabei lassen sich jedes dieser Speicherelemente entweder als einzelne 36Kb RAM oder als zwei unabhängige 18Kb RAMs benutzen. Zum Zwischenspeichern der Ein- und Ausgangssignale in den Input/Output-Blöcken (IOBs) sind Flip-Flops vorhanden.

ZYNQ-7020

Der ZYNQ-7020 basiert auf der Xilinx All programmierbaren SoC-Architektur. Dieses SoC ist in 28nm-Technologie gefertigt, also nicht strahlungsfest. Durch eine strahlungsfeste Rücksetzschaltung und weiteren Maßnahmen wird aber ein Weltraumeinsatz für nicht sicherheitsrelevante Aufgaben ermöglicht. Der ZYNQ-7020 enthält, außer einer ARM Cortex-A9 CPU, auch ein Artix-7-FPGA. Das Artix-7-FPGA im Zynq-7020 ist ganz grob zweidrittel mal so leistungsfähig wie das strahlungsharte FPGA Virtex-5QV [Bre19, S.3].

Die ARM Cortex-A9 CPUs beinhalten On-Chip-Speicher, externe Speicherschnittstellen und Verbindungsschnittstellen der Periphäriegeräte. Der schematische Aufbau eines ZYNQs sieht in etwa wie folgt aus:

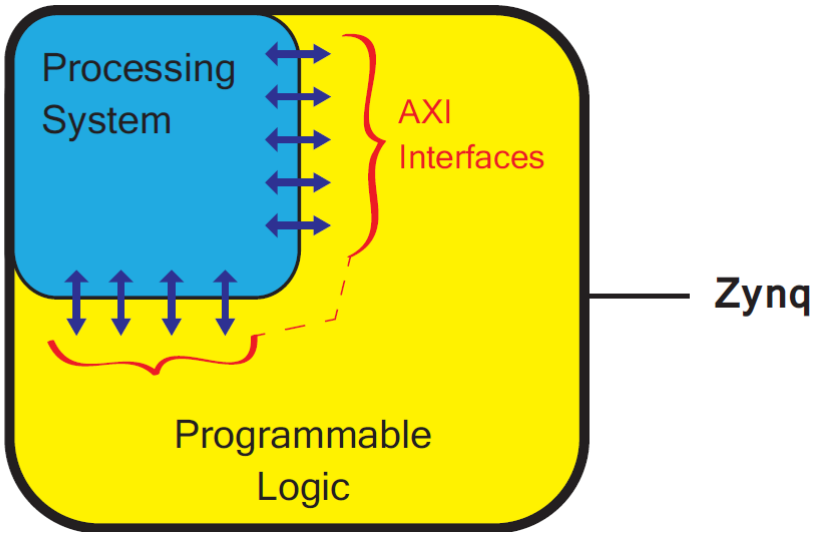

Abbildung 2.2: Architektur Aufbau eines Zynqs. [Cro+19, S.13]

Um einen Gesamtüberblick über die Architektur auf hoher Ebene zu verschaffen, besteht die ZYNQ-Architektur aus zwei Teilen. Dem Programmable System (PS) und der Programmable Logic (PL) mit einer Reihe von Verbindungen zwischen ihnen. Diese Verbindungen basieren auf dem Advanced eXtensible Interface (AXI) Standard, einem von ARM entwickelten On-Chip-Kommunikationsstandard [Cro+19, S.12].

2.2 Künstliche neuronale Netzwerke (KNN)

Autor: Marvin Soldin

2.2.1 Warum neuronale Netze?

Computer sind im Grunde nichts weiter als simple Rechenmaschinen, das heißt arithmetische Aufgaben stellen für sie kein Problem dar. Es gibt aber Kategorien von Problemen, welche sich nicht durch einfache Algorithmen lösen lassen. Hierzu gehören Probleme, die von einer großen Menge subtiler Faktoren abhängen [Ras17].

Ein sehr populäres Beispiel ist die Bilderkennung, welches ein Teilgebiet der Mustererkennung und Bildverarbeitung ist. Ein simpler Algorithmus tut sich schwer verschiedene Muster zu erkennen. Der Mensch erkennt diese Muster jedoch sehr schnell, da er diese in Bezug bringen kann [Pri13].

Der Unterschied besteht hier also in der Lernfähigkeit des Menschen, an welche es dem Computer mangelt. Der Computer ermöglicht zwar in kürzester Zeit komplizierte numerische Berechnungen, bietet jedoch keine direkte Lernfähigkeit [Ras17, S.1 ff.].

Doch genau diese Lernfähigkeit möchte man auf den Computer übertragen, um komplexere Aufgaben zu lösen. Das Studium der künstlichen neuronalen Netze ist also motiviert durch bereits bestehende biologische Systeme [Ras17, S.1 ff.].

2.2.2 Bestandteile neuronaler Netze

Ein künstliches neuronales Netz, häufiger auch KNN, besteht aus mehreren simplen Recheneinheiten, den Neuronen, sowie gerichteten, gewichteten Verbindungen zwischen diesen [OR06, S.4].

Ein künstliches Neuron *j* kann in der Regel durch vier Basiselemente beschrieben werden, siehe dazu [OR06, S.4 ff.].

Daraus ergeben sich die Basiselemente:

1. Gewichtung: Die Gewichte w_{ij} bestimmen den Grad des Einflusses, den die Eingaben des Neurons in der Berechnung der späteren Aktivierung einnehmen. Abhängig von dem Vorzeichen kann eine Eingabe hemmend oder erregend wirken.
2. Übertragungsfunktion: Die Übertragungsfunktion berechnet anhand der Gewichtung der Eingabe die Netzeingabe des Neurons.
3. Aktivierungsfunktion: Die Ausgabe des Neurons wird schließlich durch die Aktivierungsfunktion bestimmt.
4. Schwellenwert: Das Addieren eines Schwellenwerts zur Netzeingabe verschiebt die gewichteten Eingaben. Dies geschieht auch oft in Form einer zusätzlichen Konstanten Eingabe, dem Bias.

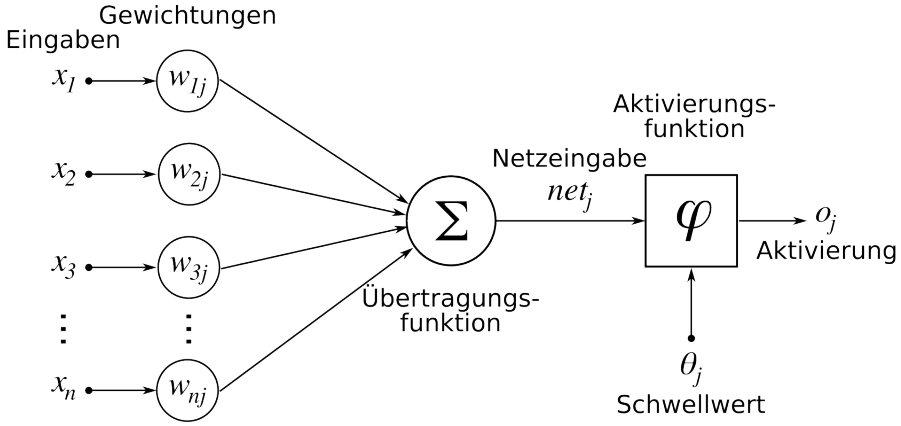

Abbildung 2.3: Schema eines künstlichen Neurons. [Per10]

Die gesamte Netzeingabe net_j ergibt sich durch das Skalarprodukt der Eingaben und Gewichtungen [OR06, S.4]:

$$\sum_{i=1}^{n} x_n * w_{nj} \tag{2.1}$$

Die Ausgabe des Neurons o_j ergibt sich durch die Aktivierungsfunktion [OR06, S.4]:

$$o_j = \varphi(net_j) \tag{2.2}$$

Die folgenden Aktivierungsfunktionen beziehen sich auf häufig genannten Beispiele in der Literatur, siehe dazu [Boh, S.56] und [OR06, S.5].

1. Identity:
$$\varphi(net_j) = net_j \tag{2.3}$$
2. Binary Step:
$$f(net_j) = \begin{cases} 1.0 & net_j \geq 0 \\ 0 & net_j < 0 \end{cases} \tag{2.4}$$
3. Soft Step:
$$\varphi(net_j) = \frac{1}{1 + e^{-net_j}} \tag{2.5}$$
4. TanH:
$$\varphi(net_j) = tahn(net_j) \tag{2.6}$$
5. ArcTan:
$$\varphi(net_j) = tan^{-1}(net_j) \tag{2.7}$$
6. Rectified Linear Unit (ReLU):
$$\varphi(net_j) = \begin{cases} 0 & net_j < 0 \\ net_j & net_j \geq 0 \end{cases} \tag{2.8}$$
7. Parameteric Rectified Linear Unit (PReLU):
$$\varphi(net_j) = \begin{cases} a * net_j & net_j < 0 \\ net_j & net_j \geq 0 \end{cases} \tag{2.9}$$
8. Exponential Linear Unit (RLU):
$$\varphi(net_j) = \begin{cases} a * (e^{net_j} - 1) & net_j < 0 \\ net_j & net_j \geq 0 \end{cases} \tag{2.10}$$
9. Soft Plus:
$$\varphi(net_j) = log_e(1 + e^{net_j}) \tag{2.11}$$

2.2.3 Aufbau eines Neuronalen Netzes

Neuronale Netze bestehen, wie bereits erwähnt, aus mehreren Recheneinheiten, den Neuronen, siehe 2.2.2 auf Seite 8. Wie genau die Neuronen zusammengesetzt werden, bestimmt die Topologie eines neuronalen Netzes. Es gibt eine Vielzahl von unterschiedlichen Topologien, welche für unterschiedliche Zwecke eingesetzt werden [Boh].

Die einfachste und häufigste Art eines neuronalen Netzes ist das sogenannte „Vorwärst gerichtete Netz“. Hier werden die Informationen von der Eingabeschicht bis hin zur Ausgabeschicht in eine Richtung weitergereicht [Ras17].

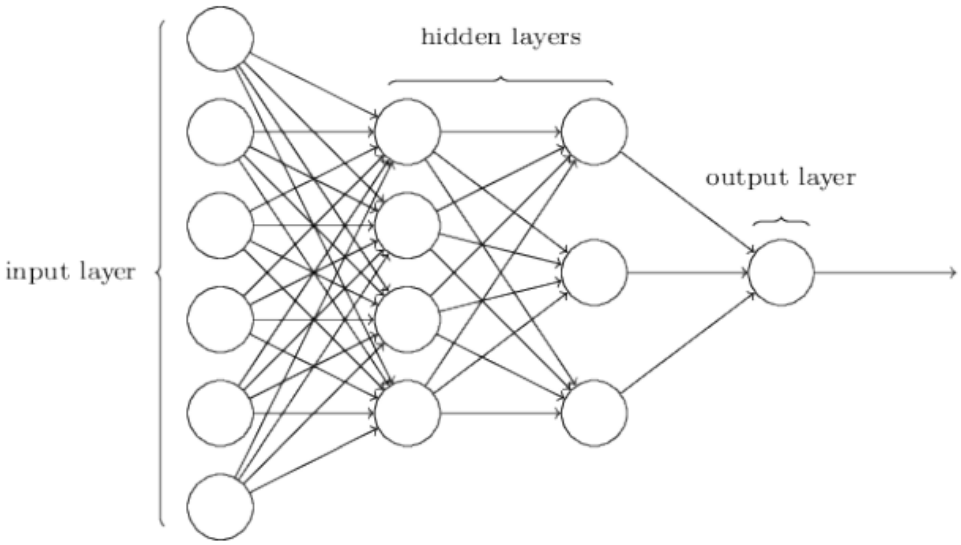

Abbildung 2.4: Architektur eines "Vorwärts gerichteten Netzes"[Nie15]

Eine einzelne Schicht eines „Vorwärts gerichteten Netzes“ kann durch eine einfache Matrizenmultiplikation und -addition berechnet werden. Die Elemente von Ausgabevektor $\overrightarrow{o}$ sind wiederum die Eingabe für die nächste Schicht, sodass sich das neuronale Netz durch eine Verkettung von Matrizenmultiplikation und -addition berechnen lässt [Ras17].

$$\overrightarrow{o} = \left(\begin{pmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \cdots & w_{k,n} \end{pmatrix} * \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{pmatrix} \right)$$

Im Weiteren werden noch häufig genutzte neuronale Netze benannt, aber nicht ausführlich erläutert:

- Rekurrente Netze: Zwar sind fast alle neuronale Netze Vorwärts gerichtete Netze, jedoch gibt es auch Varianten, bei denen zusätzliche Verbindungen existieren und bestimmte Bereiche des Netzwerkes auch rückwärts oder erneut durchlaufen werden können. Diese Netzwerke werden als rekurrente Netzwerke bezeichnet [SMH11].
- Convolutional Neural Network, zu Deutsch etwa „faltendes neuronales Netz“: Dieses Netz wird insbesondere im Bereich der Bild- und Audioverarbeitung häufig eingesetzt. Innerhalb jeder Schicht wird eine Mustererkennung durchgeführt, dabei präzisiert die darauf folgende Schicht die Mustererkennung. Grundsätzlich besteht die Struktur aus einem oder mehreren „Convolutional Layern“, gefolgt von einem „Pooling Layer“. Diese Abfolge kann beliebig oft wiederholt werden, siehe dazu [Boh, S.39ff] und [Umu+17].

2.2.4 Parallelismus in neuronalen Netzen

Neuronale Netze weisen mehrere Arten von Parallelität auf. Ein zuvor bereits genanntes Beispiel ist das Skalarprodukt eines einzelnen Neurons, siehe dazu 2.2.2 auf Seite 8. Folgende Arten der Parallelität lassen sich laut [OR06, S.12-13] innerhalb eines neuronalen Netzes finden.

1. Trainings Parallelismus: Neuronale Netze können auf unterschiedlichen beziehungsweise mehreren Geräten trainiert werden. Diese Ergebnisse können am Ende zusammengetragen werden um die bestmöglichen Parameter zu erhalten.

2. Schichten Parallelismus: In einem mehrschichtigen Netzwerk können die unterschiedlichen Schichten manchmal gleichzeitig verarbeitet werden. Das Potenzial ist hier jedoch sehr gering und daher von geringer Bedeutung.

3. Knotenparallelität: Dies ist womöglich die wichtigste Ebene der parallelen Bearbeitung. Sie bezieht sich auf die Unabhängigkeit der Neuronen in einer Schicht, denn jeder Knoten kann einzeln verarbeitet werden. Dies ist jedoch nicht immer möglich, da die Anzahl der Neuronen so groß werden kann, dass diese nicht mehr abgebildet werden können. Trotzdem lässt sich dieses Problem beispielsweise durch Hardware sehr gut lösen, denn die Berechnung von einer Schicht ist eine einfache Berechnung von Matrizen Operationen. Diese Operationen können auch sehr gut parallel ausgeführt werden.

2.2.5 Quantisierte neuronale Netze

Die folgenden Darstellungen beziehen sich auf die Ausformulierung der Werke [Umu+17] und [GMG16].

Neuronale Netze sind sehr leistungsfähig und werden schon in vielen Anwendungsbereichen, wie beispielsweise der Mustererkennung eingesetzt. Dahinter steckt aber eine sehr aufwendige Berechnung, daher gab es in den letzten Jahren eine Entwicklung, um die Anwendung neuronaler Netze zu beschleunigen. Eine Methode ist die Quantisierung der Gewichte, sodass die Berechnung mit niedrigen Fließkomma- oder Integer- Genauigkeit durchgeführt werden. Diese Berechnungen sind deutlich schneller.

Zu Deutsch kann man Quantisierung auch als Diskretisierung bezeichnen. Hierbei werden kontinuierliche Zahlenräume auf diskrete Zahlenräume abgebildet. Daraus folgt, dass sich die reelle Zahl 2,978 quantisiert auch als 2 darstellen lässt, dies geschieht lediglich indem man alle Zahlen nach dem Komma entfernt.

Eine Steigerung hiervon sind die binären neuronalen Netze, bei denen die Gewichte lediglich die Werte -1 und +1 annehmen können. Eine 1 stellt normalerweise eine +1 und eine 0 eine -1 dar. Dadurch, dass jetzt auf binärer Ebene gearbeitet wird, können Bitoperationen durchgeführt werden, welche deutlich schneller als die normale Berechnung sind.

Kapitel 3

Projektumgebung

Autor: Colin von Huth

Um die Umsetzung des Projektes durchführen zu können, wurde eine Toolchain aufgebaut. Diese wird im diesem Kapitel beschrieben.

3.1 PYNQ-Z1

Autor: Ngwa Tingoh Kingsley, Colin von Huth

PYNQ ist die Abkürzung für „Python Productivity for ZYNQ“. Aus Sicht der Hardware-Architektur ist der Kernchip von einem PYNQ ein ZYNQ, also eine FPGA SoC-Plattform, die CPU mit FPGA kombiniert, um die Abtastung, Verarbeitung und Ausgabe von Signalen durchzuführen. Softwareseitig sind die Programmiersprache Python und angepasste Programmierbibliotheken ein fester Bestandteil von PYNQ. Somit ist es einfach eingebettete Systeme auf FPGA-Basis zu entwickeln. Die programmierbaren Logikschaltungen werden als Hardwarebibliotheken importiert und über ihre APIs im Wesentlichen so programmiert, wie die Softwarebibliotheken importiert und programmiert werden [Bre19, S.2]. Ein PYNQ-Z1 Board sieht wie folgt aus:

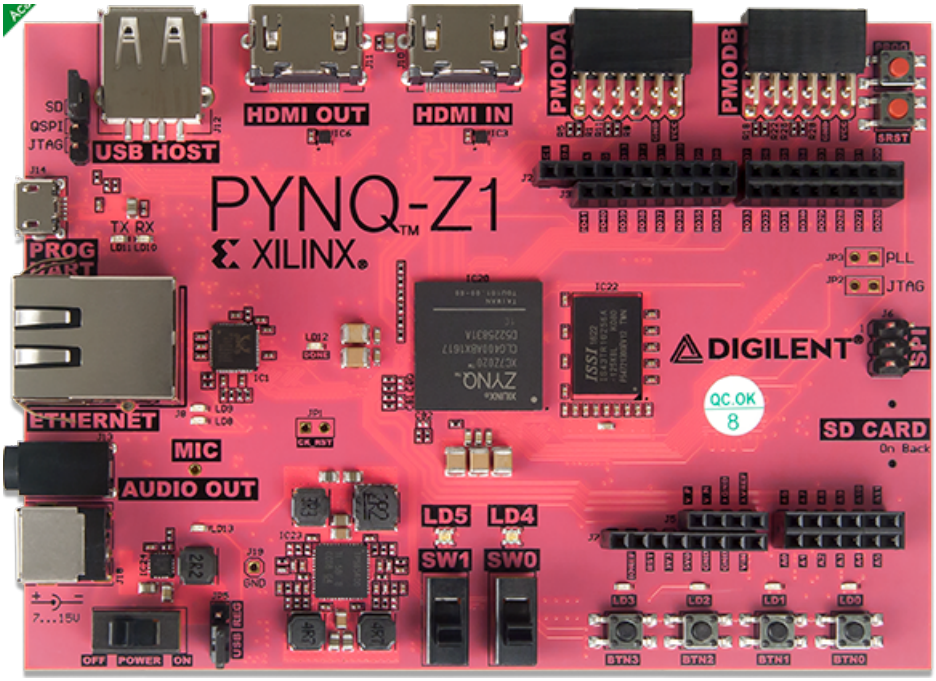

Abbildung 3.1: Das PYNQ-Z1 Board [DIG17, S. 1]

Das PYNQ-Z1 Board ist ein Produkt der Firma Digilent, welches die FPGA-Chips der Xilinx Artix-7 Familie xcz7020clg400-1 verwendet. Dieses SoC ist in 28nm-Technologie gefertigt, also auch nicht strahlungsfest und verfügt über 53200 LUTs, 4,9 MB (140 36KB Blöcke) BRAMs und 220 DSPs. Laut Aufgabenstellung muss die Leistung des FPGAs in der Größenordnung der Leistung strahlungstoleranter FPGAs bleiben, welches das PYNQ-Z1 Board mit 28nm-Technologie erfüllt.[Bre19, S.2]

Dem „PYNQ-Z1 Board Reference Manual“ [DIG17, S. 1] nach stellt die Software, welche auf dem Board vorinstalliert ist, einige grundlegende Funktionen bereit. Einige davon sind:

- Linux-Betriebssystem
- Webserver, auf dem die Jupyter Notebook Entwicklungsumgebung läuft
- Den IPython-Kernel und zugehörige Pakete
- Libaries für den FPGA

Es wurden zwei PYNQ-Z1-Boards für dieses Projekt angeschafft, da diese Entwicklungsplattform dafür entwickelt wurde, Hardware mit Python ansprechen zu können.

3.1.1 Inbetriebnahme

Autor: Colin von Huth

Installation

Bevor das PYNQ-Z1 in Betrieb genommen werden kann, muss das aktuelle PYNQ-Image auf eine Micro SD Speicherkarte installiert werden. Das aktuelle Image kann auf der PYNQ-Webseite [PYNQc] in der Sektion „Boards“ [PYNQa] heruntergeladen werden. Das heruntergeladene Image kann zum Beispiel mit der Software „balenaEtcher“ [BAL] auf die Micro SD Speicherkarte installiert werden.

Stromversorgung

Das PYNQ-Z1 kann wahlweise über den Micro-USB-Anschluss (J14) oder über die Hohlbuchse (J18) mit Strom versorgt werden. Über die Hohlbuchse lässt sich das PYNQ-Z1 mit 7 bis 15 Volt betreiben. [DIG17, S.2]

Bootvorgang

Der Bootvorgang des PYNQ-Z1 ist vergleichbar mit dem eines Mikrocontrollers. Er ist aufgeteilt in drei Stufen. Wenn das Board eingeschaltet oder zurückgesetzt wird, dann wird in der ersten Stufe (Stage 0) ein Programm vom BootROM ausgeführt. Das BootROM ist ein Speicher, von dem nur gelesen werden kann. Wenn das Board eingeschaltet und nicht zurückgesetzt wurde, wird der Status an den Modus-Pins (JP4) ausgelesen und in das Modus-Register geschrieben. Wenn das Board zurückgesetzt wurde, wird der vorherige Zustand aus dem Modus-Register benutzt. Das hat zur Folge, dass das Board einmal ausgeschaltet werden muss, um den Modus zu wechseln. Als nächstes wird, je nach Modus, ein „First Stage Bootloader“ (FSBL) vom Festspeicher in den internen RAM der „Application Processing Unit“ (APU) geladen. Als letztes wird in dieser Stufe die Ausführung an den FSBL abgegeben. In der zweiten Stufe (Stage 1) werden die Konfigurationen für die Komponenten des „Processing System“ (PS) vorgenommen. Wenn ein Bitstream im Boot Image vorhanden ist, wird dieser gelesen und dafür genutzt, die „Programmable Logic“ (PL) zu konfigurieren. Zum Schluss wird das Benutzerprogramm in den Speicher des Boot Images geladen und die Ausführung an dieses abgegeben. In der dritten und letzten Stufe (Stage 2) werden Benutzerprogramme ausgeführt. Diese Programme können z.B. Bootloader für

Linux-Betriebssysteme sein. Genauere Informationen sind im „Zynq Reference Manual“ [XIL18c] zu finden. [DIG17, S.6ff]

Das PYNQ-Z1 unterstützt drei verschiedene Modi: microSD, Quad-SPI Flash und JTAG. Der Modus wird, wie bereits beschrieben, über den Jumper JP4 eingestellt. Eine Änderung des Modi ist nur durch ein Aus- und wieder Anschalten möglich, nicht durch das zurücksetzen. Im Folgenden werden nun die unterschiedlichen Modi beschrieben. [DIG17, S.6ff]

microSD Boot Mode
Um den microSD Boot Mode nutzen zu können, muss eine microSD Karte, wie im Kapitel Installation beschrieben, vorbereitet werden. Anschließend wird die Karte in den microSD-Karten-Anschluss (J9) gesteckt. [DIG17, S.7]

Quad SPI Boot Mode
Der PYNQ-Z1 verfügt über einen 16MB Quad-SPI Flash-Speicher, wovon ebenfalls gebooted werden kann. In der Xilinx SDK Dokumentation [XIL19b] kann nachgelesen werden, wie ein Zynq Boot Image erstellt [XIL19a] werden kann. [DIG17, S.8]

JTAG Boot Mode
Im JTAG Boot Mode wartet der Prozessor, bis Software über die Xilinx Tools geladen wurde. Im Anschluss ist es möglich, diese Software laufen zu lassen oder mit dem Xilinx SDK Schritt für Schritt auszuführen. Mit dem Vivado Hardware Server ist es sogar möglich die PL unabhängig vom Prozessor zu konfigurieren. [DIG17, S.8]

3.1.2 Netzwerkkonfiguration

Autor: Colin von Huth

Da auf dem PYNQ-Z1 die webbasierte Anwendung Jupyter Notebook läuft, über die interaktiv Python auf dem PYNQ entwickelt werden kann, musste eine Möglichkeit für den Netzwerkzugriff auf das Board geschaffen werden. Zudem läuft auf dem PYNQ ein Ubuntu-basiertes Betriebssystem, welches sich für einen Zugriff über das Netzwerk gut eignet. Damit wäre es möglich zum Beispiel per SSH Befehle auszuführen oder per SCP oder Samba Dateien auf das Board zu kopieren. Für den Zugriff auf das Board werden nach dem „Getting Started“ [PYNQ18a] die folgenden Benutzerdaten verwendet:

Benutzername:	xilinx
Passwort:	xilinx

Zunächst wurde die Standard-Netzwerkkonfiguration auf dem PYNQ betrachtet. Hierzu wurde das PYNQ über den USB-Anschluss an einen Computer Verbunden. Damit ist der Zugriff über eine Serielle Schnittstelle zum Betriebssystem möglich. Nach der Dokumentation [PYNQ18b] müssen folgende Parameter für eine Kommunikation über die serielle Konsole beachtet werden:

Parameter	Wert
Baudrate	115200
Datenbits	8
Stopbits	1
Parität	Nein
Flusskontrolle	Nein

Aus der Konfigurationsdatei */etc/network/interfaces* lässt sich erkennen, dass das PYNQ standardmäßig die IP-Adresse 192.168.2.99 besitzt und zusätzlich eine IP-Adresse über DHCP abruft, sofern ein DHCP-Server im Netzwerk vorhanden ist.

Um Zugriff auf das PYNQ-Board zu bekommen, könnten beispielsweise alle Computer der Projektteilnehmer und die PYNQ-Boards über einen Switch verbunden werden. Dafür müssten dann alle Projektteilnehmer ihre Netzwerkkonfiguration anpassen und die IP-Adresse eines PYNQ-Boards müsste geändert werden, um Kollisionen zu vermeiden. Der größte Nachteil dieser Lösung besteht darin, dass

die PYNQ-Boards keinen Internet-Zugriff haben, um beispielsweise Updates oder weitere Pakete aus der Paketverwaltung zu installieren. Um den Boards einen Internetzugriff zu ermöglichen, könnte ein Computer als Gateway eingerichtet werden, um für alle an den Switch angeschlossenen Geräte einen Internetzugriff zu ermöglichen.

Bei der Praktischen Arbeit am PYNQ hat sich bewährt, das PYNQ per Netzwerkkabel direkt mit einem Computer zu verbinden und die Netzwerkkonfiguration am Computer anzupassen. Ein Internetzugriff wurde in der Regel nicht benötigt.

3.1.3 Häufige Probleme

Login im Jupyter Notebook nicht möglich

Autor: Colin von Huth

Mit dem Versionsstand des PYNQ-Images, mit dem die PYNQ-Boards ausgeliefert wurden, gab es Probleme bei der Verwendung des Jupyter Notebooks mit einigen Webbrowsern. Beispielsweise war der Webbrowser Firefox [FF] betroffen. Das Fehlerbild bestand darin, dass eine Anmeldung am Jupyter Notebook nicht möglich war. Aufgrund eines Beitrags in einem Issue [GHa] auf GitHub [GHb], welcher darauf hindeutet, dass dies in einem zukünftigen Release behoben wird, wurde als Versuch das aktuelle PYNQ-Image installiert. Hiermit wurde das Problem erfolgreich behoben.

Keine Netzwerkverbindung möglich

Autor: Tim Wieborg

Immer mal wieder kam es vor, dass nach dem Einschalten des PYNQs, dieser nicht via SSH erreicht werden konnte. Auch ein kompletter Neustart des Geräts hat das Problem nicht behoben. Warum dies so ist, konnte nicht herausgefunden werden, jedoch konnte das Problem durch das Neustarten des **networking** Services des PYNQs behoben werden. Da SSH keine Option war, muss der PYNQ über USB an einen Computer angeschlossen werden, sodass mittels serieller Schnittstelle und Telnet eine Verbindung aufgebaut werden kann. Anschließend muss man sich, wie bei SSH, anmelden und mittels *sudo systemctl restart networking* der Service neu gestartet werden. Nun kann man sich wieder über SSH verbinden. Benötigte Benutzerdaten sowie Baudrate und Co können unter 3.1.2 auf der vorherigen Seite gefunden werden.

3.2 Tensorflow

Autor: Marvin Soldin

TensorFlow, beschrieben durch [Mar+15], ist eine frei verfügbare Software, die speziell für das maschinelle lernen von Google konzipiert wurde. Mithilfe von TensorFlow können Datenflussgraphen erzeugt werden, deren Knoten aus Operationen bestehen und an deren Kanten Tensoren weiter durch den Graphen gegeben werden. Ein Tensor ist ein n-dimensionales Array. Ein eindimensionaler Tensor wird auch Vektor und ein zweidimensionaler Tensor auch Matrix genannt.

Die Datentypen innerhalb eines Tensors sind alle gleich, zum Beispiel *int32*, *string* oder *float32*.

Der Hauptteil von Tensorflow ist in C++ implementiert und hat verschiedene Aufgaben. Dazu zählen unter anderem die Abstraktion von verschiedenen Schnittstellen wie zum Beispiel der CPU, GPU oder einer TPU (Tensor Processing Unit).

Eine TPU ist eine von Google entwickelte Beschleunigerkarte, im Allgemeinen auch als anwendungsspezifische integrierte Schaltung (ASIC) bekannt.

Auf den Hauptteil von Tensorflow setzt eine C-Schnittstelle, welche aus Kompatibilitätsgründen verwendet wird. Dadurch wird eine vereinfachte Nutzung in unterschiedlichen Programmiersprachen ermöglicht. Auf diese Schnittstelle setzen wiederum eine C++ und Python Schnittstelle auf. Auf die Python Schnittstelle folgt dann die High-Level-Bibliothek Keras.

3.3 Keras

Autor: Marvin Soldin

Die folgende Darstellung nimmt [Cho+15] zur Grundlage der allgemeinen Beschreibung.

Keras ist eine High-Level-Bibliothek zum Beschreiben von neuronalen Netzen. Die Bibliothek ist in Python geschrieben und läuft auf der Basis von TensorFlow, CNTK oder Theano. Keras wurde entwickelt, um schnelles Experimentieren und einen einfachen Einstieg in das Thema neuronale Netze zu ermöglichen. Keras folgt dabei folgenden Prinzipien:

1. Benutzerfreundlichkeit: Keras ist eine API, die für Menschen und nicht für Maschinen entwickelt wurde. Es stellt die Benutzererfahrung in den Mittelpunkt.
2. Modularität: Unter einem Modell wird eine Sequenz oder ein Diagramm von eigenständigen, vollständig konfigurierbaren Modulen verstanden, die mit möglichst wenigen Einschränkungen zusammengesteckt werden können. Dabei sind neuronale Schichten, Kostenfunktionen, Optimierer, Initialisierungsschemata, Aktivierungsfunktionen und Regularisierungsschemata eigenständige Module, welche konfigurierbar sind um neue Modelle erstellen zu können.
3. Einfache Erweiterbarkeit: Neue Module können einfach hinzugefügt werden. Vorhandene Module bieten zahlreiche Beispiele dieser Möglichkeit.
4. Arbeiten mit Python: Keine separaten Modellkonfigurationsdateien in einem deklarativen Format. Modelle werden in Python-Code beschrieben, welcher kompakt und einfacher zu debuggen ist und eine einfache Erweiterbarkeit ermöglicht.

3.4 Xilinx Vivado HLx Editions

Autor: Tim Wieborg

Die Xilinx Vivado Design Suite ist ein Softwaretool zur Synthese sowie Analyse von HDL-Code wie Verilog oder VHDL. Es löst die 15 Jahre alte ISE Design Suite ab. Es beinhaltet sowohl das Vivado SDK an sich, als auch eine zusätzliche IDE für das Entwickeln von C, C++ und SystemC-Code (Vivado HLS).

3.4.1 Installation

Die Installation des WebPacks stellt keine große Herausforderung dar. Eine Anleitung für die Installation unter Linux kann im Anhang unter 77 gefunden werden. Man sollte lediglich darauf achten, dass genug Speicherplatz vorhanden ist. Die gepackten Installationsdateien haben eine Größe von etwa 21,5 GB. Das anschließende Installationsverzeichnis hat etwa eine Größe von 25 GB.

3.4.2 PYNQ-Z1 spezifisches Projekt erstellen

Um ein PYNQ-Z1 spezifisches Projekt in der Vivado Suit zu erstellen, müssen zu aller erst die Board-Files des PYNQ-Z1 in das Board-Repository hinzugefügt werden. Eine genaue Beschreibung kann im Anhang auf Seite 76 gefunden werden.

Sind nun die Board-Files hinzugefügt, kann die Vivado IDE gestartet werden. Über „File->Project->New“ in der Toolbar, öffnet sich ein neuer Assistent zum Erstellen eines neuen Projektes.

Nach dem klick auf „Next“, legt man nun den Projekt Namen sowie den Standort des Projektverzeichnisses fest. Der Haken bei „Create project subdirectory“ sollte gesetzt sein. Weiter mit „Next“.

Nun wird der Projekt Typ festgelegt. Es handelt sich hierbei um ein „RTL Project“. Möchte man nicht direkt schon eine Datei hinzufügen, muss außerdem der Haken bei „Do not specify sources at this time“ gesetzt sein.

Als nächstes muss das PYNQ-Z1 Board ausgewählt werden. Dazu wechselt man auf den Reiter „Boards“, sucht nach PYNQ-Z1 und wählt diesen Eintrag aus. Es darf nur auf den Tabelleneintrag geklickt werden und nicht auf den Board-Namen, da sich sonst der Webbrowser öffnet und weitere Informationen über das Board liefert. Weiter mit „Next“ und „Finish“.

Die Standard Programmiersprache ist auf Verilog eingestellt. Möchte man die Ressourcen jedoch in VHDL schreiben, muss im „Project Manger->Settings“ unter „Project Settings->General“ die Box „Target language“ auf **VHDL** umgestellt werden.

Um den FPGA die Kommunikation mit dem Artix-7 zu ermöglichen und die Basis des PYNQs zu schaffen, muss nun ein IP-Core in das Design geladen werden. Dazu klickt man im „Flow Navigator“ unter IP Integrator auf „Create Block Design“. Nun kann der Name des Block Designs gewählt werden. Dies hat aber keine Auswirkungen auf die Funktionalität, weshalb auch der Standartname reicht. Mit einem klick auf OK wird das neue Design erstellt.

Über das + im Diagram-Bereich, wird nun der IP-Core „ZYNQ7 Processing System“ sowie „AXI Interconnect“ hinzugefügt und automatisch mit „Run Block Automation“ an die Schnittstellen verbunden. Für eine Standard Konfiguration sollte in dem neu erschienenen Fenster „Cross Trigger In / Out“ deaktiviert und „Apply Board Preset“ aktiviert sein. Anschließend sollten im Block Design drei IP-Cores zu sehen sein. Das „PYNQ7 Processing System“, „AXI Interconnect“ und der „Processor System Reset“. Dies ist der Grundaufbau, da SoC und FPGA über AXI (Advanced eXtensible Interface) kommunizieren.

Zu guter Letzt muss darauf geachtet werden, dass ein Block Design immer von einem HDL Wrapper ummantelt ist. Dafür wechselt man im Block Design von „Design“ auf „Sources“, macht ein Rechtsklick auf die .bd-Datei und wählt „Create HDL Wrapper...“.

3.5 PyCharm

Autor: Sebastian von Minden

PyCharm ist eine IDE für die Programmiersprache Python. Vertrieben wird PyCharm von JetBrains in der kostenlosen Community Version und in der gebührenpflichtigen Professional Version.

3.5.1 Installation

Autor: Sebastian von Minden

Grundlegend für die Installation von PyCharm ist ein Rechner mit 4GB RAM, wobei 8 GB vom Hersteller empfohlen werden. Des weiteren ist eine vorherige Installation von Python notwendig. Dabei sollte entweder Python2 in der 2.7.X Version oder Python3 in der Version 3.4.X oder höher verwendet werden. JetBrains empfiehlt für die Verwendung von PyCharm ein 64 Bit Linux-Betriebssystem mit einer KDE, Unity oder Gnome Desktop Umgebung.

Unter Ubuntu 18.04 lässt sich PyCharm einfach über das Ubuntu-Software Tool installieren. Über das Tool kann dann je nach bedarf die PyCharm CE (Community Edition) oder die PyCharm Pro Version Installiert werden.

Um die IDE auf anderen Linux Systemen zu installieren werden Sudo Rechte sowie die Installationsfiles von der JetBrains Seite [JET] benötigt. Wie auch unter Ubuntu lässt sich hier wieder aus den zwei Versionen Pro und Community Edition auswählen. Die heruntergeladenen Files sollten als pycharm-*.tar.gz (* entspricht der Edition und der Versionsnummer Beispiel: pycharm-professional-2019.2.5.tar.gz) im Download Verzeichnis zu finden sein.

Das Entpacken dieser Datei sollte nach Herstellerempfehlung im Ordner /opt geschehen. Dies kann jedoch auch angepasst werden, wenn gewünscht. Zum entpacken selbst muss dann in einer Kommandozeile an den Ort der PyCharm-*.tar.gz navigiert werden. Ist man im Ordner der Heruntergeladenen tar-Datei und möchte diese aus dem Download Ordner in den /opt/ Ordner verschieben geschieht dies über den Befehl:

sudo mv PyCharm-.tar.gz /opt/*

Ist alles verschoben, genügt ein einfaches aufrufen des tar-Befehls mit dem Zielordner in den die Daten entpackt werden sollen:

sudo tar xzf pycharm-.tar.gz -C /opt/*

Nachdem alles entpackt wurde, muss die *pycharm.sh* ausgeführt werden. Diese ist in dem Ordner */opt/pycharm-*/bin* zu finden. Zum Ausführen dieses Shell-Scripts muss nun noch *./pycharm.sh* eingegeben werden. Im Anschluss wird ein Wizard aufgerufen. Dieser Fragt zuerst ob bereits bestehende Einstellungen übernommen werden sollen. Wenn man ohne vorheriger Einstellungen weiter macht, wird man gefragt welcher Farbmodus für den Editor verwendet werden soll. Ist man mit dem auswählen des Farbschemas durch, befindet man sich in der Projekt Auswahl. In der Projektauswahl kann man nun wählen, ob man bei dem Anlegen eines neuen Projektes die VirtualEnv, eine Virtuelle Umgebung verwenden möchte oder ein Projekt auf dem herkömmlichen Weg angelegt werden soll. Die Virtuelle Umgebung für das Projekt lässt den Nutzer Bibliotheken und Pakete in unterschiedlichen Versionen zu verschiedenen Projekte installieren.

3.6 Versionsverwaltung

Autor: Colin von Huth

Um Quellcode und Dokumente auszutauschen und Versionen von Quellcode verwalten zu können, werden für das Projekt einige Repositories benötigt. Hierbei haben sich die Projektteilnehmer aufgrund vorhandener Erfahrungen für die Versionsverwaltung mit Git [GIT] entschieden.

3.6.1 GitLab

Da bereits zu Beginn des Projektes klar war, dass mehrere Repositories benötigt werden, wurde die Gruppe „hsb-projekt-ws1920“ auf GitLab [GLb] eingerichtet. Die Plattform GitLab ist einigen Projektteilnehmern bereits aus anderen Projekten bekannt. Aufgrund der guten Erfahrungen mit GitLab hat sich die Projektgruppe für den Einsatz dieser Plattform entschieden.

Gitlab ist im wesentlichen eine Webanwendung, die viele Funktionen für die Zusammenarbeit zwischen Entwicklern bereitstellt. Es können (Projekt-)Gruppen und darunter Projekte angelegt werden. Innerhalb von Projekten gibt es beispielsweise ein Repository für den Quellcode, ein Wiki für die Dokumentation, eine Aufgabenverwaltung inkl. Scrum-Board und Möglichkeiten für Continuous Integration (CI). Zudem können Gruppen- oder Projektweit Rollen verteilt werden, welche sich auf die Rechte von Benutzern auswirken.

Die Repositories und Inhalte dieser sind im Anhang D beschrieben. Die Inhalte der Repositories liegen ebenfalls in der beigelegten ZIP-Datei bei.

3.6.2 Integrationen

GitLab lässt sich an allerhand Tools anbinden. Da die Projektteilnehmer intern über Slack [SL] kommunizieren, bat es sich an, eine Verbindung zwischen diesen Tools herzustellen. Es ist möglich, per Slack über Aktivitäten in den Projekt-Repositories benachrichtigt zu werden. Wie dies konfiguriert wird, ist in der GitLab-Dokumentation [GLa] beschrieben.

Kapitel 4

Rechercheergebnisse zur werkzeugunterstützten Generierung von neuronalen Netzen in VHDL

Autor: Marvin Soldin

In diesem Kapitel wird auf die verschiedenen Möglichkeiten eingegangen, neuronale Netze mit möglichst wenig Aufwand auf FPGAs abzubilden. Diese Möglichkeiten werden evaluiert, um einen genauen Überblick zu schaffen.

4.1 BNN-PYNQ

Autor: Marvin Soldin

Dieses Framework, welches auf dem FINN-Framework aufbaut, enthält verschiedene Python Overlays für die Plattformen PYNQ-Z1, PYNQ-Z2 und Ultra96. Diese Overlays ermöglichen den Einsatz von Quantisierten neuronalen Netzen, siehe [Umu+17]. Dabei werden folgende Implementationen neuronaler Netze angeboten:

- 1 Bit Gewicht mit einer 1 Bit Aktivierungsfunktion für CNV und LFC Netze.
- 1 Bit Gewicht mit einer 2 Bit Aktivierungsfunktion für CNV und LFC Netze.
- 2 Bit Gewicht mit einer 2 Bit Aktivierungsfunktion für CNV.

Das LFC Netz besteht aus einem vollständig verbundenen, dreischichtigen, neuronalen Netzwerk, welches zum Klassifizieren des MNIST-Datensatzes [Coh+17] genutzt wird. In einer einzelnen Schicht sind 1024 Neuronen. Das Netzwerk akzeptiert 28x28 Pixel große Bilder im Binärformat und gibt als Rückgabewert einen 10-Bit großen One-Hot-Kodierten Vektor zurück. Dieser gibt die Richtige Klassifizierung der Eingabedaten, mit einer Genauigkeit von 98.4% an [Umu+17, S.7].

Das CNV Netz besteht aus einem Convolutional Neural Network, zu Deutsch etwa „faltendes neuronales Netzwerk“, welches von BinaryNet [Cou+16] und VGG-16 [SZ14] inspiriert ist. Das neuronale Netz wird zur Klassifizierung der beiden Datensätze CIFAR-10 [KH10] und SVHN [Net+11] verwendet. Das neuronale Netz akzeptiert 32x32 Pixel große Bilder mit 24 Bit pro Pixel und gibt einen 10-Element-Vektor mit 16-Bit-Werten als Ergebnis zurück [Umu+17, S.7].

4.1.1 FINN Framework

Das FINN Framework ist ein experimentelles Framework der Xilinx Research Labs zur Untersuchung der Inferenz neuronaler Netze auf FPGAs. Es zielt speziell auf quantisierte neuronale Netzwerke ab, wobei der Schwerpunkt auf der Erzeugung von Datenflussarchitekturen liegt, die für jedes Netzwerk individuell angepasst sind. Es handelt sich dabei nicht um einen generischen DNN-Beschleuniger wie der Xilinx Deep Neural Network (xDNN) Beschleuniger, sondern um ein Tool zur Erforschung des Entwurfsraums von DNN-Inferenzbeschleunigern auf FPGAs. [Umu+17]

Der xDNN Beschleuniger ist eine Möglichkeit, um neuronale Netze auf den Xilinx Alveo Data Center Beschleuniger Karten auszuführen. Diese Möglichkeit übertrifft momentan viele gängige CPU- und GPU-Plattformen. [XIL18b]

4.1.2 Nutzung des Frameworks

Die Nutzung dieser Lösung erweist sich als sehr schwierig. Zwar sind Beispiele enthalten, die wirklich leicht auszuführen sind, dennoch gibt es keine genaue Anleitung, wie man neue neuronale Netze konfiguriert und trainiert. Die Beispiele lassen sich nach Öffnen direkt ausführen und für die unterschiedlichen Anwendungen benutzen.

Folgende Beispiele sind enthalten:

- BNN-Beispiel für den Datensatz CIFAR-10.
- BNN-Beispiel für Stopp-Schilder Erkennung.
- BNN-Biespiel für den Datensatz SVHN.
- und viele Weitere ...

Diese Beispiele verdeutlichen sehr stark die Möglichkeiten der Hardware und der Quantisierung. Das Beispiel der Stoppschild-Erkennung gibt in der Entwicklungsumgebung die Geschwindigkeit der Verarbeitung an. So gibt beispielsweise die Software an, man hätte auf der Hardware eine Verarbeitungsgeschwindigkeit von 3038,67 Bilder pro Sekunde und auf Software lediglich eine Geschwindigkeit von 0,89 Bilder pro Sekunde.

4.1.3 Fazit

Aufgrund der fehlenden Dokumentation lässt sich dieses Framework ohne großen Zeitaufwand leider nicht dazu benutzen, um neue neuronale Netze für eine Hardware Umgebung zu erstellen.

Die Entwickler arbeiten aber momentan daran, eine neue Toolchain zu erstellen, um den Ablauf modularer zu gestalten. Leider fehlt auch hier bisher eine genaue Dokumentation, welcher aber laut Entwicklerseite noch folgen soll. [Umu+17]

Abbildung 4.1: Überarbeite Toolchain [Umu+17]

Aus diesem Grund haben wir uns gegen den Einsatz dieser Software entschieden, da wir nicht in der Lage sind neuronale Netze auf neue Problemstellungen anzuwenden. In Zukunft könnte dieses Framework aber noch einiges an Potenzial bieten, da es beeindruckende Ergebnisse mithilfe der Quantisierung erzielt, welche in nachfolgender Grafik vom Hersteller zusammengefasst worden sind.

Name	Dataset	Platform	Precision	Err. (%)	kFPS	P_{chip} (W)	P_{wall} (W)	kFPS/P_{chip}	kFPS/P_{wall}	GOPS
SFC-max	MNIST	ZC706	1	4.17	12,361	7.3	21.2	1693.29	583.07	8,265.45
LFC-max	MNIST	ZC706	1	1.60	1,561	8.8	22.6	177.39	69.07	9,085.67
MFC-max	MNIST	ZC706	1	2.31	6,238	11.3	28.5	552	218.8	11,612.86
CNV-max	CIFAR-10	ZC706	1	19.90	21.9	3.6	11.7	6.08	1.87	2,465.5
CNV-max	SVHN	ZC706	1	5.10	21.9	3.6	11.7	6.08	1.87	2,465.5
Alemdar et al. [1]	MNIST	Kintex-7 160T	2	2.24	255.10	0.32	-	806.45	-	~96.68
Alemdar et al. [1]	MNIST	Kintex-7 160T	2	1.71	255.10	1.84	-	138.50	-	~448.47
Alemdar et al. [1]	MNIST	Kintex-7 160T	2	1.67	255.10	2.76	-	92.59	-	~864.03
Park and Sung [20]	MNIST	ZC706	3	-	70	4.98	-	14.06	-	~210
TrueNorth [6]	CIFAR-10	TrueNorth	1	16.59	1.249	0.2044	-	6.11	-	-
TrueNorth [6]	SVHN	TrueNorth	1	3.34	2.526	0.2565	-	9.85	-	-
CaffePresso [9]	MNIST	Keystone-II	16	-	5	-	14	-	0.357	44.82
CaffePresso [9]	CIFAR-10	Keystone-II	16	-	10	-	14	-	0.714	146.14
CaffePresso [9]	MNIST	Parallella	32	-	0.64	-	5	-	0.129	5.78
CaffePresso [9]	CIFAR-10	Parallella	32	-	0.1	-	5	-	0.019	1.40
Ovtcharov et al. [19]	CIFAR-10	Stratix V D5	32	~11-26	2.32	-	25	-	0.093	-

Abbildung 4.2: Ergebnisse der Ausführung Quantisierter neuronaler Netze [Umu+17]

4.2 LeFlow

Autor: Tim Wieborg

LeFlow ist ein „Tool-flow“ um in Python geschriebenen Code für maschinelles lernen z.B. mithilfe des TensorFlow Frameworks in synthetisierbaren Code wie VHDL oder Verilog zu transformieren“[NSW18].

Wie in Abbildung 4.3 zu sehen ist, besteht der Workflow aus mehreren, verschiedenen Schritten. Die Toolchain beginnt mit ganz einfachem TensorFlow Code. Dieser wird mittels dem von Google speziell für TensorFlow entwickelten XLA Optimierer [TEN] in LLVM-IR (LLVM-compatible intermediate representation) übersetzt, jedoch nicht weiter optimiert. LLVM ist ein Compiler-Unterbau ähnlich der Java Virtual Machine. Durch die Zwischenschicht kann noch im Nachgang der Maschinen-Code für unterschiedliche Systeme kompiliert werden.

LeFlow optimiert nun den von XLA generierten Rechengraphen (computational graph) und transformiert ihn, sodass die generierte, Software ähnliche, Schnittstelle zu einer Hardware ähnlichen Schnittstelle wird. Anschließend wird LegUp verwendet, um aus dem LLVM-IR Rechengraphen Hardware-kompatiblen Verilog-Code zu generieren. Dieser kann dann mittels FPGA Compiler für die entsprechende Hardware generiert werden.

LegUp ist ein Compiler, der aus LLVM-Programmcode direkt synthetisierbaren Programmcode wie Verilog oder VHDL generieren kann. Für LeFlow wird von den Entwicklern die Version 4.0 empfohlen da sie für die Entwicklung von LeFlow verwendet wurde.

Anmerkung: Seit der Entwicklung von LeFlow wurde LegUp weiterentwickelt. Die aktuellste Version lautet 7.4 (Stand November 2019) und wird nicht mehr über die „University of Toronto“ vertrieben, sondern über eine eigene Webseite[LEGb].

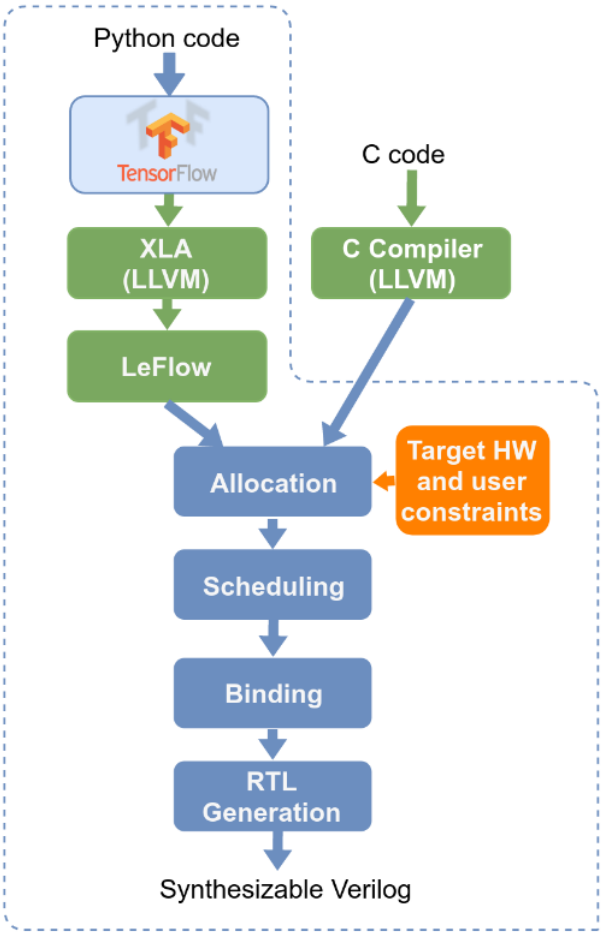

Abbildung 4.3: LeFlow's Tool-flow [NSW18]

Um LeFlow zu verstehen, wurde die LegUp VM heruntergeladen und die Schritt für Schritt Anleitung aus dem LeFlow Repository befolgt [LEF]. Anschließend wurde eines der mitgelieferten Beispiele von LeFlow kompiliert und das Resultat analysiert. Aufgefallen ist die Anzahl an neu generierten Dateien und Verzeichnissen. Durch die kaum vorhandene Dokumentation war es nicht möglich, die benötigten Dateien zu selektieren bzw. allgemein Verständnis aufzubauen.

Nach einem Blick in die LegUp Dokumentation entschieden wir uns schließlich, nach anderen Lösungsansätzen als LeFlow bzw. LegUp zu suchen. In der Dokumentation der genutzten Version von LegUp steht geschrieben, dass FPGAs der Firma Xilinx nur rudimentär unterstützt werden, was ein Ausschlusskriterium ist [LEGa]. Zwar ist LegUp in dem Tool-flow von LeFlow ein Austauschbares Bauteil, jedoch müssten wir für den Austausch zu viel Zeit investieren, um die Toolchain für unser Problem und unsere Projektumgebung anzupassen.

Selbst wenn letztendlich der Tool-flow funktioniert hätte, müssten sowohl die Datenströme noch in den FPGA geleitet werden als auch die Resultate zurückgeholt werden. Da dies direkt auf der Hardware geschieht, würde auch das wieder einen sehr großen Aufwand bedeuten, da das entsprechende Senden der Inputwerte und das Empfangen der Resultate in Verilog oder VHDL beschrieben werden muss.

4.3 MyHDL

Autor: Sebastian von Minden

MyHDL ist ein Open Source Paket für Python. Dieses Paket ermöglicht dem Benutzer, Hardware in Python zu programmieren und diesen Code dann anschließend in VHDL oder Verilog umzuwandeln. Weiter ist es möglich, Testbenches und Simulationen mit MyHDL zu erstellen und durchzuführen.

Die Sprache von MyHDL baut dabei auf die bestehende Python Syntax auf und erweitert diese um mehrere Module, um eine HDL konforme Form des Codes herzustellen.

Infolge der Tatsache, dass MyHDL einen komplexen Aufbau besitzt und die Umwandlung von Python-Code in VHDL einige Nacharbeit benötigt, ist die Zeitersparnis bei vielen Konvertierungen kaum bis sehr gering einzuschätzen. Daher ist der Weg, direkt in VHDL zu entwickeln, sinnvoller und benötigt keine weitere Einführung in spezielle Pakete für Python.

4.4 DPU (Deep Learning Processor Unit)

Autor: Ngwa Tingoh Kingsley

DPU ist ein Xilinx Hardware IP-Core, der konfiguriert werden kann. Er wurde speziell für Convolutional Neural Networks (CNN) entwickelt[DPU19, S.5 ff]. Die Einheit enthält ein Registerkonfigurationsmodul, ein Datenkontrollmodul und ein Convolution Computing Modul. Es gibt einen speziellen Befehlssatz für DPU, der es ermöglicht, effizient für viele CNNs zu arbeiten. DPU umfasst folgende bekannte CNNs: VGG, ResNet, GoogLeNet, YOLO, SSD, MobileNet, FPN, etc. Die DPU IP kann als Block in die programmierbare Logik (PL) der ausgewählten Zynq®-7000 SoC und Zynq UltraScaleTM+ MPSoC Geräte mit direkter Verbindung zum Verarbeitungssystem (PS) integriert werden.

Nach einer detaillierten Untersuchung der Funktionsweise der DPU konnte festgestellt werden, dass es derzeit keine offizielle Unterstützung für die PYNQ-Z1 Platine gibt. Hinzu kommt, dass die DPU von Xilinx über eigene Befehlssätze verfügt und diese auf das ausgewählte Board abgestimmt sein müssen. Wenn dies nicht der Fall ist, dann könnten bei einer Anwendung unvorhersehbare Probleme auftreten. Deswegen wurde an dieser Stelle DPU für das Projekt ausgeschlossen.

Zusammenfassend lässt sich dennoch konstatieren, dass es sich lohnt, diese Technologie im Auge zu behalten. Denn sie könnte den Prozess des Einsatzes von CNN schnell beschleunigen.

4.5 VGT (VHDL auto-generation tool for optimized hardware acceleration of convolutional neural networks on FPGA)

Autor: Ngwa Tingoh Kingsley

VGT ist ein Werkzeug zur VHDL-Generierung, das den Zeit- und Arbeitsaufwand bei der Implementierung von CNNs auf FPGAs reduzieren soll [VGT17, S.1 ff]. Das Werkzeug ermöglicht es den Benutzern, ein CNN-Modell über eine grafische Benutzeroberfläche einfach zu konfigurieren und einen hoch optimierten VHDL-Code dafür zu generieren. Der generierte Code soll eine modulare, hoch parallele, skalierbare, rekonfigurierbare Implementierung mitsamt vollständiger Pipelines des CNN-Zielmodells widerspiegeln.

Nach dem Herunterladen, der Installation und dem Befolgen der Anweisungen, zeigten sich vielversprechende Möglichkeiten, da das spezifische FPGA gewählt werden konnte, das genau in das PYNQ-Z1-Board integriert ist. Aufgrund der modularen Struktur konnten außerdem die Gewichte, die Bias, die Aktivierungsfunktionen und die Anzahl der Schichten gewählt werden, die im Wunsch-CNN verwendet werden sollten. Aber bei dem Versuch, den Code in Vivado auszuführen, traten einige Fehler auf, die ohne eine detaillierte Implementierung nicht behoben werden konnten. Wegen der Fehler beim Ausführen des VHDL Codes wurde der Autor kontaktiert, bisher jedoch ohne Erfolg. Deshalb kommt das Werkzeug für das Projekt nicht in Frage.

4.6 Zusammenfassung

Autor: Colin von Huth

Die Recherche zu Werkzeugen/Tools für die automatische Generierung von VHDL-Quellcode, welche neuronale Netze darstellen, lieferte einige Ergebnisse.

Das Framework BNN-PYNQ konnte aufgrund unzureichender Dokumentation nicht erfolgreich getestet werden. Aus zeitlichen Gründen wurde dieser Ansatz für das Projekt verworfen. Allerdings scheinen die Entwickler derzeit an einer Neuauflage der Toolchain zu arbeiten. Demnach könnte diese in Zukunft von Relevanz sein.

Bei LeFlow mangelte es ebenfalls an aussagekräftiger Dokumentation, weswegen kein zureichendes Verständnis für die Verwendung aufgebaut werden konnte. Zudem schien LeFlow keine Xilinx FPGAs zu unterstützen. Aus diesen Gründen wurde LeFlow nicht weiter betrachtet.

Mit MyHDL kann VHDL oder Verilog Quellcode erzeugt werden. Leider scheint der erzeugte Quellcode nicht einfach anwendbar zu sein. Der Quellcode müsste weitläufig nachbearbeitet werden, um zum Funktionieren gebracht werden zu können. Aus zeitlichen Gründen wurde dieser Ansatz verworfen.

Der IP-Core DPU von Xilinx schien auch ein vielversprechender Ansatz zu sein, jedoch unterstützt DPU nicht das PYNQ-Z1-Board. Aus diesem Grund wurde sich damit nicht weiter beschäftigt. Zukünftig könnte möglicherweise eine Unterstützung entwickelt werden, aber dies ist ungewiss.

Das Tool VGT zur Generierung von VHDL-Code erschien zunächst vielversprechend, da es den auf dem PYNQ-Z1 verbauten FPGA unterstützt. Leider ist der generierte Quellcode unbrauchbar, da bei der Synthetisierung allerhand Fehler ausgegeben wurden. Ohne Zeit in eine intensive Fehlersuche zu investieren, ist dieses Tool also unbrauchbar.

Zusammenfassend lässt sich sagen, dass die recherchierten Werkzeuge zwar interessante Ansätze bieten, aber für die Umsetzung dieses Projektes nicht ausreichend dokumentiert, ausgereift oder grundsätzlich nicht geeignet sind. Aus diesem Grund fiel die Entscheidung darauf, neuronale Netze selbst mit VHDL auf der Hardware zu implementieren. Diese Implementierung wird im Kapitel 6.4 auf Seite 42 beschrieben.

Kapitel 5

Rechercheergebnisse zur Abbildung von neuronalen Netzen auf FPGAs

Autor: Colin von Huth

In diesem Kapitel wird darauf eingegangen, wie neuronale Netze auf einem Xilinx FPGA abgebildet werden können. Insbesondere wird dabei das Board PYNQ-Z1 betrachtet.

5.1 Advanced Extensible Interface (AXI)

Autor: Tim Wieborg

Xilinx verwendet seit den Spartan-6 und Virtex-6 Geräten das AXI Protokoll für die Kommunikation zwischen IP-Cores. Auch auf dem von uns verwendeten FPGA wird das Protokoll zum Ansprechen von IP-Cores verwendet. Gerade für unser Projekt ist das AXI-Protokoll von besonderer Bedeutung, da wir mit dem PYNQ-Z1 den Datenstrom vom Processing System (PS) zur Programming Logic (PL) über AXI leiten müssen. Mittels sog. Overlays können wir in Python auf Adressen zugreifen, die wiederum dem AXI bekannt sind. Ein IP-Core kann die Werte in den Adressen dann über das AXI-Protokoll auslesen und anschließend, falls nötig, an eine weitere Adresse das Resultat hinschreiben, was wiederum von Python ausgelesen werden kann.

Um die verschiedenen AXI-Protokolle „Lite“ & „Stream“ zu verstehen, wurde jeweils ein kleines Beispiel für beide Protokolle implementiert. Bei dem AXI-Lite Beispiel (6.2) wird ein Addierer implementiert, bei dem AXI-Stream Beispiel (6.3) wird ein Datenstrom zum FPGA gesendet und anschließend wieder empfangen.

5.1.1 AXI4 und AXI4-Lite

Autor: Sebastian von Minden

AXI4 ist für Memory-Mapping I/O, also Speicher bezogene Adressierung mit hoher Übertragungsleistung vorgesehen. AXI4-Lite basiert auf dem selben Prinzip wie AXI4-Standard, jedoch mit geringerer Übertragungsrate, da AXI4-Lite kein Burst Transfer (schreiben von mehreren Daten während eines Aufrufs) beherrscht, sondern nur wortweise die Daten mit einer Wortlänge von 32 Bit beim Schreiben und Lesen. AXI4 kann mit Hilfe des Bursts bis zu 256 Bit beim Lesen und Schreiben übertragen.

Die Lite Variante sollte vorrangig zum Übertragen von Kontroll- und Status-Registern verwendet werden. Der Datenaustausch unter Lite beschränkt sich auf ein Datum per Transaktion.

AXI4 hingegen kann durch seine höhere Bandbreite auch für größere Datentransfers verwendet werden. Daten können simultan in beide Richtungen zwischen Master und Slave über diese Kanäle übertragen werden.

Das AXI4-Lite sowie das AXI4 Interface bestehen aus 5 Kanälen, dem Read-Adresskanal, Read-Datakanal, Write-Adresskanal, Write-Datakanal und dem Write-Responsekanal.

5.1.2 AXI4-Stream

Autor: Sebastian von Minden

AXI4-Stream verwendet für die Datenübertragung einen Kanal. Dieser Kanal ist wie der Write-Data Kanal von AXI4 aufgebaut, mit dem Unterschied, das über diesen konstant Daten mit Burst übertragen werden können. Eine Übertragung mit AXI-Stream erfolgt ebenfalls zwischen Master und Slave. Die Daten werden innerhalb eines Taktzyklus übertragen. Um diese Übertragung zu starten legt der Sender ein Validierungssignal auf den Bus, das TVALID ist das Signal, das den Empfänger fragt ob dieser empfangsbereit ist, dies beantwortet dieser dann mit dem TREADY. Ist der Slave empfangsbereit legt der Master die zu übertragenden Daten TDATA auf den Bus. Die Übertragung eines TDATA Pakets wird dann mit deinem TLAST abgeschlossen. Anschließend wird mit dem nächsten Taktzyklus ein weiteres Paket übertragen, dies geschieht solange bis der Master nichts mehr zu senden hat oder die Übertragung unterbrochen wurde.

5.2 Projekte auf dem PYNQ-Z1 ausführen

Autor: Tim Wieborg

Um ein Projekt auf dem PYNQ auszuführen, müssen das Hardware-Design in Form einer .tcl-Datei und die Bitstream-Datei des in Vivado erstellten Designs auf den PYNQ-Z1 kopiert werden. Wie man sich mit dem PYNQ-Z1 verbinden kann, ist bereits unter 3.1.2 beschrieben. Anschließend kann in der Jupyter-Oberfläche im Webbrowser dann ein neues Jupyter-Notebook erstellt werden, in der dann das Modul „Overlay“ importiert wird. Nun kann dann eine Overlay-Instanz mit dem Pfad zur Bitstream-Datei erstellt werden. Die Bitstream-Datei ist das Hardware-Design, welches nun auf den FPGA hochgeladen wird. Über das Overlay hat man Zugriff auf die IP-Cores bzw. auf die Register im Adressraum der IP-Cores.

5.3 Python Overlay

Autor: Tim Wieborg

Dieser Abschnitt ist eine Zusammenfassung und Interpretation der Dokumentation des PYNQs. Siehe dazu [PYNQb].
Overlays sind das Grundprinzip der PYNQ-Boards. Sie sind die Schnittstelle zwischen der PL und Python. Es besteht aus einer Bitstream-Datei, die bei dem synthetisieren des Codes generiert wird und einer .tcl-Datei, die aus dem Block-Design exportiert wird.

Der Grundgedanke bei Overlays ist die Nutzung von sog. IP-Cores, welche wahlweise in C, C++ oder SystemC geschrieben werden können. Diese Cores können dann in ein Block-Design integriert und mit dem Rest verbunden werden. In Python wird dann mit der Bitstream-Datei ein Overlay geladen, über das man dann auf die verschiedenen IP-Cores mittels AXI zugreifen und kommunizieren kann. Wichtig ist dabei die Benennung der Bitstream- und .tcl-Datei. Diese müssen den gleichen Namen haben, da beim Erstellen des Overlays nach einer gleichnamigen .tcl-Datei gesucht wird.

Ein Overlay kann in Python folgendermaßen geladen werden:

```
from pynq import Overlay
ol = Overlay("base.bit")
```

Anschließend kann mit dem Befehl *?ol* eine Übersicht des Overlays generiert werden. Es zeigt z.B. alle geladenen IP-Cores an. Wichtig dabei ist, dass nicht die IP-Cores direkt angesprochen werden können,

sondern lediglich gemeinsamer Speicher oder definierte GPIOs.

Wurde ein IP-Core im Overlay gefunden, kann dieser als direkte Variable des Overlays angesprochen werden. Enthält das oben genannte Overlay z.B. einen IP-Core Namens *meinIPCore* und wurde gemeinsamer Speicher für den Core reserviert, kann dieser Folgendermaßen angesprochen werden:

```
ip_core = ol.meinIPCore
ip_core.write(0x10, 64)
```

Ist die Variable *ip_core* angelegt, kann man auch hier wieder über *?ip_core* mehr erfahren. In diesem Beispiel wird nun an der Adresse *0x10* der Wert 64 geschrieben. Was der IP-Core nun mit diesem Wert macht, muss bei der Definition des Cores festgelegt worden sein.

5.4 Custom Driver

Autor: Tim Wieborg

Mithilfe von Custom Driver können komplexe Zugriffe auf IP-Cores in einfache Python Funktionen abstrahiert werden. Dieses Kapitel ist eine Zusammenfassung der PYNQ-Z1 Dokumentation unter [PYNQ17a].

Ein Beispiel:
Das Addieren von zwei Werten und auslesen der Summe beinhaltet das Schreiben zweier Werte in bestimmten Registern sowie das Auslesen der Summe aus einem anderen Register. Anstatt jedes mal wieder diese 3 Aufrufe zu tätigen und sich die fixen Adressen zu merken, kann dies in eine Funktion ausgelagert werden, die lediglich zwei Eingabewerte erhält und die Summe zurückgibt.

Dies kann nun in eine sog. Custom Driver-Klasse ausgelagert werden. Beim einlesen der Bitstream-Datei beinhalten die IP-Cores eine ID, anhand der nach einem *Driver* für den entsprechenden IP-Core gesucht wird. Wird kein spezieller Driver gefunden, wird *DefaultIP* geladen. Wurde jedoch ein Custom Driver gefunden, der die entsprechende ID des Cores beinhaltet, wird stattdessen dieser Driver geladen und dem Core zugewiesen.

Damit automatisch das richtige Overlay für die Schaltung gewählt wird, muss eine neue Klasse in Python implementiert werden, welche von *DefaultIP* erbt.

Die *__init__*-Funktion muss mindestens Folgendes beinhalten:

```
def __init__(self, description):
    super().__init__(description=description)
```

Damit der Driver bestimmten IP-Cores zugewiesen werden kann, muss die *bindto*-Variable genutzt werden. Sie beinhaltet eine Liste mit allen IDs der IP-Cores, denen dieser Driver zugewiesen werden soll.

Die ID eines IP-Cores kann in Vivado im Block Design bei ausgewähltem Core in den Properties der Eigenschaft *VLNV* entnommen werden. Die bindto-Variable kann folgendermaßen aussehen:

```
bindto = ['user.org:user:axi_adder:1.0']
```

Nun können außerdem jegliche Funktionen der Klasse hinzugefügt werden, die später mit dem IP-Core zutun haben. Für den Addierer kann eine Funktion folgendermaßen aussehen:

```
def add(self, a, b):
    self.write(0x0, a)
    self.write(0x4, b)
    return self.read(0x8)
```

Damit der Driver nun automatisch geladen wird und die Klasse über die *import*-Funktionalität gefunden wird, muss die Datei der Python Umgebung bekannt gemacht werden. Das einfachste ist es, die Datei im Verzeichnis */home/xilinx/pynq/* oder darunter abzulegen, da dieses Verzeichnis der Jupyter Umgebung als Library-Source bekannt ist.

Anschließend muss die Klasse in Python importiert und das Overlay geladen werden.

Mittels *?<overlay>.<ip_core>* kann nun eine Übersicht über den IP-Core generiert werden. Unter *Type* kann nun der geladene Driver gefunden werden. Ist dies nicht *DefaultIP*, wurde ein Custom Driver geladen.

5.5 Erstellen eines AXI IP-Cores in C

Autor: Tim Wieborg

Die Anleitung für ein „Custom Overlay“ für den PYNQ-Z1 beinhaltet in einer etwas älteren Version der Anleitung (v2.0) das Erstellen eines IP-Cores in C. Da zu Beginn des Projekts noch unklar war, wie Overlays generell funktionieren, war dies also die erste Anlaufstelle. Anhand der Anleitung wurde ein einfacher „adder“ IP-Core in C implementiert.

Da wir uns in diesem Projekt auf VHDL fokussiert haben, war die Anleitung in der PYNQ-Dokumentation lediglich eine gute Einführung, aber nicht relevant für das Projekt. Die genaue Anleitung kann im Anhang C gefunden werden.

5.6 Erstellen eines AXI IP-Cores in VHDL

Autor: Tim Wieborg

Damit ein IP-Core im Overlay erkannt wird und mit Python kommunizieren kann, muss der IP-Core mittels AXI angeschlossen werden. Bei der Erstellung eines solchen IP-Cores kann Vivado einem helfen, indem die AXI-Schnittstelle bereits vorgeneriert wird. Um einen neuen IP-Core zu erstellen und direkt im Projekt nutzen zu können, muss bereits ein Projekt in Vivado geöffnet sein. Eine Anleitung, wie man ein Vivado Projekt spezifisch für den PYNQ-Z1 erstellt, kann unter 3.4.2 gefunden werden.

Anschließend wird über „Tools->Create and Package New IP“ ein neuer Assistent aufgerufen. Nach einem Klick auf „Next“ muss nun „Create a new AXI4 peripheral“ ausgewählt werden. Auf der nächsten Seite können nun spezifische Details über den IP-Core spezifiziert werden. Anschließend können mehrere AXI Interface hinzugefügt werden. Zur Auswahl gibt es **Lite**, **Full** und **Stream**. Die genauen Unterschiede können im AXI Reference Guide [XIL17] gefunden werden.

Je nach benötigten Ressourcen kann nun die Datengröße und die Anzahl der Register angepasst werden. Über die Register können nachher Python und FPGA miteinander Kommunizieren und Daten austauschen.

Sind alle Einstellungen vorgenommen, muss auf der nächsten Seite unter „Next Steps:“ noch „Edit IP“ ausgewählt und auf „Finish“ geklickt werden. Der neue IP-Core inkl. AXI-Interface wird nun erstellt. Es öffnet sich eine neue Vivado Instanz, in der der IP-Core nun bearbeitet werden kann.

Zu Beginn besteht der IP-Core aus zwei VHDL-Dateien. Eine AXI-VHDL-Datei, die die Schnittstellen für das AXI-Modul definiert und die Top-Datei, die eine Instanz aus dem AXI-Interface erzeugt. Möchte man nun seine eigene Komponente in den IP-Core hinzufügen bzw. einen Prozess erstellen, der über Python angesprochen werden kann, muss in dem Untermodul gearbeitet werden, da dort die Register definiert sind. Möchte man die Register wo anders benutzen, müssen Sie entsprechend in der Schnittstelle als Input bzw. Output spezifiziert werden.

Grundsätzlich sollten die spezifische Änderungen immer am Ende der Dateien eingefügt werden. Dies wird auch durch die automatisch generierten Kommentare „Add user logic here“ und „User logic ends“ gekennzeichnet.

Sind alle Änderungen vorgenommen, muss der IP-Core neu paketiert werden. Dazu muss im „Project Manager->Edit Packaged IP“ alle **Packaging Steps** mit grünen Haken versehen werden. Unter File Groups reicht es z.B. aus, „Merge changes from FileGroups Wizard“ anzuklicken. Im letzten Step „Review und Package“ muss dann final auf Re-Package IP geklickt werden. Der IP-Core wird nun mit den Änderungen gepackt und zum eigentlichen Projekt hinzugefügt.

Ist man wieder zurück in dem eigentlichen Projekt, kann der neu generierte IP-Core nun im Block Design über + hinzugefügt werden. Da der IP-Core mit den Standardschnittstellen des AXI-Interfaces generiert wurde, kann über „Run Block Automation“ der IP-Core automatisch mit dem **AXI Interconnect** Modul verbunden werden.

Um das Projekt nun aktiv nutzen zu können, muss der Bitstream generiert und das Block Design exportiert werden. Der Bitstream wird mittels „PROGRAM AND DEBUG -> Generate Bitstream“ generiert. Bei größeren Projekten kann das Generieren sehr lange dauern. Hat man einen Leistungsstärkeren Computer mit mehreren Rechenkernen zu Verfügung, kann bei dem Fenster „Launch Runs“ die „Number of Jobs“ erhöht werden. Die fertige Bitstream-Datei befindet sich anschließend im Projektordner unter <projektname>.runs. Es ist die .bit-Datei.

Das Block Design kann ganz einfach über „File->Export->Export Block Design“ exportiert werden. Dabei ist es wichtig, dass man sich im Block Design befindet, da die Option **Export Block Design** sonst nicht verfügbar ist. Das Zielverzeichnis des exportierten Block Designs kann selbst gewählt werden.

Sowohl die Bitstream-Datei als auch die generierte .tcl-Datei des Block Designs werden anschließend für das Overlay benötigt. Mehr dazu unter 5.3.

5.7 Auswahl einer Aktivierungsfunktion

Autor: Ngwa Tingoh Kingsley

Aktivierungsfunktionen sind immer noch ein relevantes Forschungsthema und es ist wahrscheinlich, dass sich in Zukunft neue Perspektiven bezüglich passender Aktivierungsfunktionen für eine bestimmte Anwendung ergeben werden [Cro+19, S.490, ff].

Ursprünglich war die Aktivierungsfunktion Sigmoid eine sehr beliebte Wahl, da sie mit einem einfach zu berechnenden Gradienten für die Rückpropagation arbeitet. Diese ist für kleine NN geeignet, jedoch waren tiefe NN schwer zu trainieren, da sehr kleine oder sehr große Werte die Neuronenausgabe sättigen und den Lernprozess stagnieren würden. Die Aktivierungsfunktion Tanh hat ähnliche Eigenschaften wie Sigmoid, kann aber sowohl negative als auch positive Werte ausgeben, was den einzelnen Neuronen mehr Spielraum für die Arbeit bietet.

Die derzeit dominierende Wahl für die meisten Anwendungen, insbesondere aber für die Computer-Vision, ist die Aktivierungsfunktion ReLU. Die Implementierung von ReLU ist sehr einfach, mit nur einer max()-Funktion, die eine Eingabe übergibt, wenn sie über Null liegt und eine Null ausgibt, wenn sie negativ ist. ReLU hat sich erstmals als Teil der AlexNet-Implementierung für ImageNet bei Bilddaten bewährt und zeigte eine Steigerung der Konvergenzgeschwindigkeit beim Training um den Faktor 6. Seitdem ist es zur beliebtesten Wahl der Aktivierungsfunktion geworden.

Obwohl die Sigmoid- und Tanh-Funktionen in der Computer-Vision weniger häufig verwendet werden, sind sie dennoch in sehr erfolgreichen tiefen NN-Architekturen zu finden, die für Sprachverarbeitung (NLP) Aufgaben eingesetzt werden. Zum Beispiel verwenden LSTM-Netze (Long Short Term Memory) in ihrem Kern Sigmoid- und Tanh-Funktionen.

Kapitel 6

Umsetzung

Autor: Colin von Huth

In diesem Kapitel werden verschiedene Projekte vorgestellt, welche im währen der gesamten Projektlaufzeit umgesetzt wurden. Dazu zählen Beispiele für neuronale Netze mit Tensorflow und Keras, sowie AXI Lite und AXI Stream Projekte in Vivado. Außerdem wird ein komplettes, ausführbares, neuronales Netz entwickelt, welches das Verhalten von einem XOR hat. Abschließend wird die Umsetzung der Aktivierungsfunktion Sigmoid in VHDL beschrieben. Nach diesem Vorgehen können weitere, komplexe Aktivierungsfunktionen in Hardware implementiert werden. Die Aktivierungsfunktion wurde nicht im XOR-Projekt verwendet, da das Problem mit der weniger komplexen Aktivierungsfunktion ReLU gelöst werden konnte.

6.1 Beispiel-Projekte in Tensorflow und Keras

Autor: Colin von Huth, Marvin Soldin

In diesem Kapitel geht es um verschiedene neuronale Netze, die beispielhaft in Tensorflow und Keras umgesetzt wurden.

Zur Verwendung der Beispiele kann mit der Entwicklungsumgebung PyCharm gearbeitet werden (siehe 3.5 auf Seite 19). Grundsätzlich ist der normale Interpreter von Python in der Version 3.7 dafür ausreichend. Die folgenden Pakete müssen zusätzlich installiert sein, da sie für die Ausführung der Beispiele notwendig sind.

scikit-learn:	0.21.3
numpy:	1.17.2
matplotlib:	3.1.1
keras:	2.3.1
tensorflow:	2.0.0

Der Quellcode befindet sich, wie in Kapitel D beschrieben, im entsprechenden Unterverzeichnis in der beigelegten ZIP-Datei.

6.1.1 Verdopplung

Autor: Colin von Huth

Dieses Beispiel trainiert ein neuronales Netz mit Ein- und Ausgabewerten. Jeder Ausgabewert entspricht der Verdopplung des jeweiligen Eingabewertes. Im Anschluss an das Training kann eine Vorhersage für einen Eingabewert erfolgen. Diese Vorhersage wird ausgegeben.

Der vollständige Quellcode ist in der Datei *double.py*, wie unter D.4 auf Seite 84 beschrieben, zu finden.

6.1.2 Umrechnung von Längeneinheiten

Autor: Marvin Soldin

Dieses Beispiel verdeutlicht die Funktionsweise eines einzelnen Neurons, geht also noch nicht auf die Verbindung mehrere Neuronen ein. Im Genaueren behandelt es die Umrechnung von Kilometern in Meilen.

$$km = mi * 1,609 \tag{6.1}$$

```
# Training-Daten
training_data = [
    [10],
    [15],
    [60]
]

# Ergebnis-Daten
target_data = [
    6.2,
    9.3,
    37.3
]

# Lineare Regression ohne Bias, der Bias wird durch fit_intercept=True angegeben.
model = LinearRegression(fit_intercept=False)
```

Der vollständige Quellcode ist in der Datei *kilometers_miles.py*, wie unter D.4 auf Seite 84 beschrieben, zu finden.

6.1.3 Umrechnung von Temperatureinheiten

Autor: Marvin Soldin

Dieses Beispiel geht ebenfalls auf die Funktionsweise eines einzelnen Neurons ein. Im Genaueren behandelt es die Umrechnung von °C in °F. Das Besondere ist, dass die Funktion immer einen Schwellwert hat, sodass das Verständnis über Neuronen mit Schwellwert verdeutlicht wird.

$$^{\circ}F = ^{\circ} C * 1.8000 + 32.00 \tag{6.2}$$

```
# Trainings-Daten
training_data = [
    [-10],
    [0],
    [20]
]

# Ergebnis-Daten
target_data = [
    14,
    32,
    68
]

# Lineare Regression, der Bias wird durch fit_intercept=True angegeben.
model = LinearRegression(fit_intercept=True)

# Modell trainieren
model.fit(training_data, target_data)
```

Der vollständige Quellcode ist in der Datei *temperature.py*, wie unter D.4 auf Seite 84 beschrieben, zu finden.

6.1.4 XOR

Autor: Colin von Huth

Dieses Beispiel stellt ein neuronales Netz dar, welches die Funktion von einem XOR aus der Digitaltechnik abbildet.

Hier wurden erstmalig einzelne Schichten (layer) mit einigen Parametern definiert. Es wurde explizit für jede Schicht eine Aktivierungsfunktion angegeben. Außerdem wurden Bereiche für die Startwerte und die Gewichte definiert. Eine ausführliche Beschreibung ist im Kapitel 6.4.2 auf Seite 43 zu finden.

Beim Training werden als Eingabe alle möglichen Eingabekombinationen und als Ausgabe die jeweilige Ausgabe von einem XOR verwendet:

IN1	IN2	OUT
0	0	0
0	1	1
1	0	1
1	1	0

Der vollständige Quellcode ist in der Datei *xor.py*, wie unter D.4 auf Seite 84 beschrieben, zu finden.

6.1.5 Bestimmung von Kleidungsstücken

Autor: Marvin Soldin

Dieses Beispiel verdeutlicht das erste größere neuronale Netz, welches zur Klassifikation von Kleidung genutzt werden kann. Es analysiert dabei Bilder in der Größe von 28x28 Pixeln und er erzeugt als Ausgabe einen 10-Bit großen One-Hot-Kodierten Vektor zurück. Dabei gibt der Index an um welches Element es sich handelt.

- T-Shirt
- Trouser
- Pullover
- Dress
- Coat
- Sandal
- Shirt
- Sneaker
- Bag
- Ankle Boot

Der vollständige Quellcode ist in der Datei *fashion_mnist.py*, wie unter D.4 auf Seite 84 beschrieben, zu finden.

6.2 Ein AXI Lite Projekt

Autor: Tim Wieborg

Für dieses Beispiel wurde ein „Addierer“ implementiert. Die Idee ist, dass mittels der AXI-Lite Register zwei Werte vom Prozessor an den FPGA übertragen werden. Dieser addiert die zwei Werte und schreibt die Summe in ein drittes Register, welches dann vom Prozessor wieder ausgelesen und über Python ausgegeben wird.

Die Projektressourcen können im Anhang D.2 auf Seite 82 gefunden werden.

Dazu wurde ein neues Projekt in Vivado angelegt und das PYNQ-Z1 Board aus den Boards ausgewählt. Die Implementierungssprache ist VHDL. Anschließend wurde, wie unter 5.6 beschrieben ein neuer AXI-Lite IP-Core mit 4 Registern und 32 Bit Länge pro Register erstellt.

Anschließend wurde in der Unterdatei (Standardmäßig endet die Datei mit S00_AXI) am Ende der Datei folgender Prozess hinzugefügt:

```
process (S_AXI_ACLK)
begin
  if (rising_edge(S_AXI_ACLK)) then
    result_out <= std_logic_vector(signed(slv_reg0) + signed(slv_reg1));
  end if;
end process;
```

Dies führt dazu, dass die Werte aus Register 0 (*slv_reg0*) und Register 1 (*slv_reg1*) addiert und die Summe in das Signal *result_out* geschrieben wird. Das Signal *result_out* wird in Zeile 120 definiert. Anschließend muss *result_out* noch auf Register 3 (*slv_reg2*) gemapped werden. Dies passiert in Zeile 361:

```
120  signal result_out : std_logic_vector(
  C_S_AXI_DATA_WIDTH-1 downto 0);

361  reg_data_out <= result_out;
```

Der IP-Core wurde anschließend „Re-Packaged“ und das Projekt geschlossen. Nun wurde in dem Hauptprojekt ein neues Block Design erstellt und folgendermaßen aufgebaut:

Aus dem Block Design wurde ein neuer HDL Wrapper erstellt, sodass anschließend die Bitstream-Datei generiert und das Block Design exportiert werden konnte. Die benötigten Dateien wurden auf das PYNQ-Z1 Board kopiert, sodass Sie in einem neuen Jupyter Notebook genutzt werden können. Für dieses Beispiel wurde außerdem ein „Custom Driver“ mit der enthaltenen Funktion „add“ in Python geschrieben. Dieser bezweckt, dass nur die Funktion aufgerufen werden muss, um den gesamten Prozess zur Übertragung der Werte und das Auslesen des Resultats zu abstrahieren. Die Klasse sieht folgendermaßen aus:

```
from pynq import DefaultIP

class AddDriver(DefaultIP):
  def __init__(self, description):
    super().__init__(description=description)

 # VLNV-Adresse zu finden in Vivado Block Design View unter
 # den Properties des IP-Cores
  bindto = ['user.org:user:axi_adder:1.0']

  def add(self, a, b):
   self.write(0x0, a)
   self.write(0x4, b)
   return self.read(0x8)
```

Das Jupyter Notebook sieht wie folgt aus (Vorausgesetzt, .bit, .tcl und die Driver-Klasse liegen unter */home/xilinx/pynq/overlays/axi_lite_adder* auf dem PYNQ):

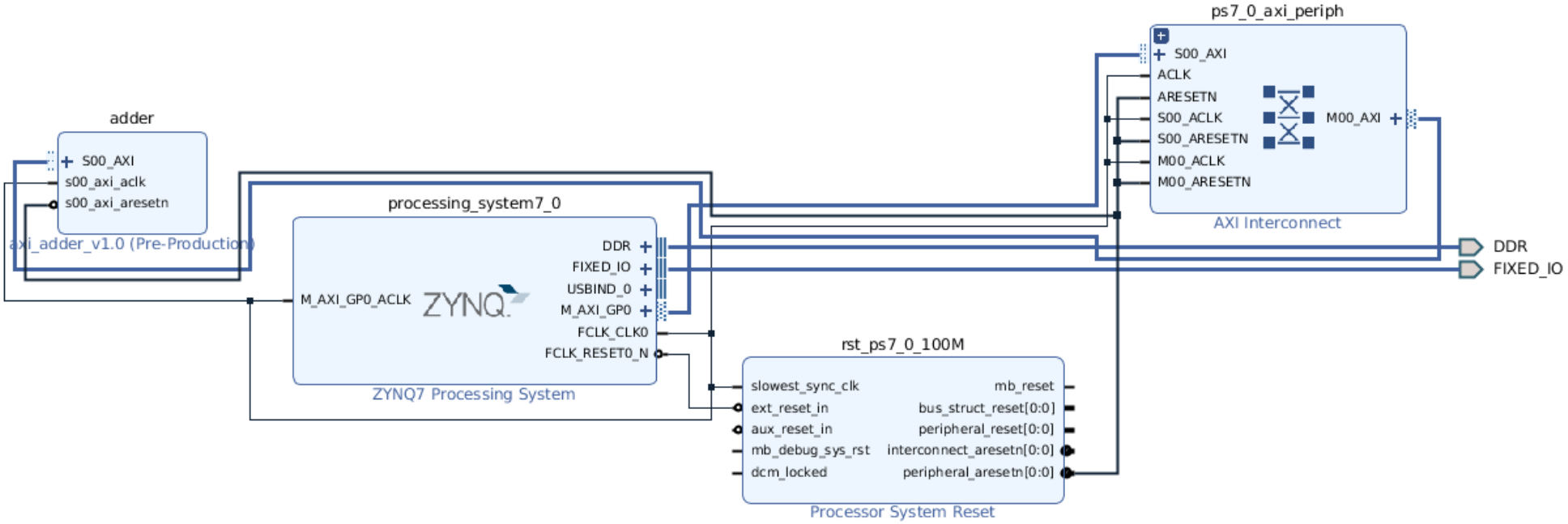

Abbildung 6.1: Block Design des Projekts „AXI-Lite Adder“

```
from pynq import Overlay

# Selbst geschriebener 'driver' fuer den adder.
# Ueber die methode add koennen nun 2 Werte summiert werden
from pynq.overlays.axi_lite_adder import add_driver
overlay=Overlay('/home/xilinx/pynq/overlays/axi_lite_adder/axi_lite_adder.bit')

# Greift auf den IP-Core 'adder' zu.
# Durch die VLNV Nummer wurde beim import der custom Driver 'Adder' geladen.
ip_adder=overlay.adder

ip_adder.add(7,8)
```

Über die add() Funktion können nun alle 2^{32} Bit Integer-Werte addiert werden. Lediglich das Addieren zweier Werte, die ein negatives Resultat ergeben, führt zu falschen Ergebnissen. Dies liegt vermutlich an der Interpretation von Integerwerten in Python.

6.3 Ein AXI Stream Projekt

Autor: Tim Wieborg

Um zu verstehen, wie der AXI Stream funktioniert, wurde ein eine einfache First-In First-Out(FIFO) Schaltung implementiert. Mittels dieser Schaltung kann ein Datenstream an die PL übertragen und empfangen werden. Dazu wird auf PS-Seite in Python ein Numpy Array mit den entsprechenden Werten angelegt. Dieses wird über einen Direct-Memory-Access (DMA) IP-Core in den RAM geschrieben. Anschließend können die Werte über den DMA wieder in gleicher Reihenfolge ausgelesen und zurückgegeben werden.

Die Projektressourcen können im Anhang D.1 auf Seite 82 gefunden werden.

Für die Implementierung wurde ein neues Projekt in Vivado angelegt und das PYNQ-Z1 Board ausgewählt. Anschließend wird ein neues Block Design generiert und das Zynq7 Processing System als Core hinzugefügt. Nun sollte die „Block Automation“ einmal ausgeführt werden, damit das System an den RAM und die I/O angeschlossen wird. Mit einem Doppelklick auf den Core öffnet sich ein neues Fenster, in dem einige Variablen des Systems angepasst werden können. Unter dem Reiter „PS-PL Configuration“ muss nun sichergestellt werden, dass „AXI Non Secure Enablement->GP Master AXI Interface->M AXI GP0 interface“ aktiviert ist. Zusätzlich muss noch „HP Slave AXI Interface->S AXI HP0 interface“ aktiviert werden. Die *DATA WIDTH* sollte auf 64 Bit gestellt sein. Über „OK“ wird das Fenster geschlossen und die Einstellungen übernommen.

Nun wird ein „AXI SmartConnect“ zum Block Design hinzugefügt. Der *M00_AXI* des SmartConnects Port wird nun mit *S_AXI_HP0* des Processing Systems verbunden. Nun wird ein „AXI Direct Memory Access“ IP-Core hinzugefügt und mit einem Doppelklick konfiguriert.

Wichtig: Nun müssen die Haken aus „Enable Scatter Gather Engine“ und „Enable Control / Status Stream“ entfernt werden. Die Overlay Architektur unterstützt keine Scatter Gather Engine (aktueller Stand). Bleibt diese aktiviert, ist die Schaltung außer Funktion.

Zurück im Block Design, werden nun vom DMA die Anschlüsse *M_AXI_SG* und *M_AXI_MM2S* an die Slave-Verbindungen (*S00* & *S01*) des vorherig hinzugefügten SmartConnect angeschlossen.

Nun wird ein „AXI4-Stream Data FIFO“ hinzugefügt und der *M_AXIS_MM2S* des DMA mit *S_AXIS* des FIFOs verbunden. Zusätzlich muss noch *M_AXIS* des FIFOs mit *S_AXIS_S2MM* des DMAs verbunden werden.

Zu guter letzt muss ein weiterer „AXI SmartConnect“ mit einem Master- und einem Slave-Interface hinzugefügt werden. Der *M00_AXI* des SmartConnects wird mit *S_AXI_LITE* des DMAs und der *S00_AXI* mit dem *M_AXI_GP0* des Processing Systems verbunden. Nun kann die *Connection Automation* erneut ausgeführt werden, sodass alle benötigten Verbindungen wie Takt und Reset angeschlossen werden.

Die fertige Schaltung sollte nun wie in Abbildung 6.2 aussehen.

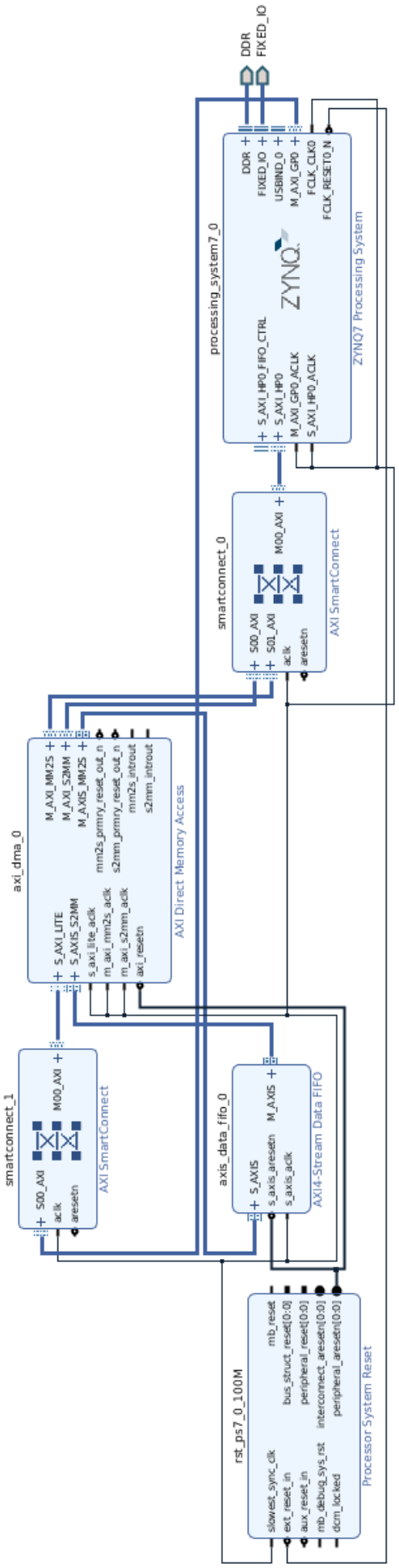

Abbildung 6.2: Block Design des Projekts „AXI-Stream FIFO“

Da nun mit dem RAM-Modul gearbeitet wird, müssen für die IP-Cores Adressen reserviert werden. Dazu geht man vom Block Design aus in den „Address Editor“ und klickt auf *Auto Assign Address*. Sowohl das *Processing System* als auch der *DMA* sollten nun einen Adressbereich zugewiesen bekommen haben.

Anschließend wird aus dem Block Design ein HDL Wrapper erstellt und die Bitstream-Datei generiert. Zusätzlich muss noch das Block Design exportiert und die Dateien auf den PYNQ kopiert werden.

Das Jupyter Notebook sieht wie folgt aus (Vorausgesetztm .bit- und .tcl-Datei liegen unter */home/xilinx/pynq/overlays/axi_stream_fifo/* auf dem PYNQ):

```
import numpy as np
from pynq import Overlay
from pynq import Xlnk

overlay=Overlay('/home/xilinx/pynq/overlays/axi_stream_fifo/axi_stream_fifo.bit')

dma = overlay.axi_dma_0

data_size = 10

xlnk = Xlnk()
input_buffer = xlnk.cma_array(shape=(data_size,), dtype=np.float32)
output_buffer = xlnk.cma_array(shape=(data_size,), dtype=np.float32)

for i in range(data_size):
input_buffer[i] = i

# Send Data
dma.sendchannel.transfer(input_buffer)
dma.sendchannel.wait()

# Receive Data
dma.recvchannel.transfer(output_buffer)
dma.recvchannel.wait()
```

Zu aller erst werden die Bibliotheken NumPy, Overlay und Xlnk importiert und das Overlay aus der Bitstream-Datei geladen. Anschließend wird für den einfacheren Zugriff der DMA in der Variable *dma* gespeichert. Mittels Xlnk können Arrays erstellt werden, die direkt im RAM Speicheradressen reservieren. Über Xlnk werden *input_buffer* und *output_buffer* Arrays erstellt. Mithilfe einer for-Schleife wird das input-Array mit Werten gefüllt. Mithilfe der sendchannel.transfer()-Funktion kann nun ein Stream an das Processing System gesendet werden. Über recvchannel.transfer() kann ein Stream abgefragt werden. Die wait()-Funktion wartet auf das abschließen des entprechenden Transfers.

6.4 XOR als neuronales Netz

Autor: Marvin Soldin, Colin von Huth

Zu Beginn wurde ein einfaches Beispiel zur Implementation gewählt. Als Beispiel wird die das Verhalten einer XOR-Funktion in Form eines neuronalen Netzes implementiert. Die XOR-Funktion wurde als Problem gewählt, da es in der Literatur [OR06, S.48 ff.] ein sehr populäres Beispiel ist. Es ist eines der ersten Probleme, welches sich nicht durch eine einfache Lineare Regression lösen lässt.

Dieses Kapitel behandelt die Umsetzung des neuronalen Netzes. Außerdem wird dargestellt, welche Komponenten wie genutzt werden und wie die Kommunikation der Komponenten untereinander funktioniert.

Generell sieht das System wie in Abbildung 6.3 aus. Ein Python-Programm kann ein sogenanntes Overlay importieren. Über Funktionen in dem Overlay kann dann abstrahiert mit der Hardware kommuniziert werden. Diese Kommunikation findet über eine spezielle Schnittstelle (AXI) statt. Die Kommunikationsendpunkte sind sogenannte IP-Cores. Diese können in VHDL beschrieben und auf dem FPGA synthetisiert werden.

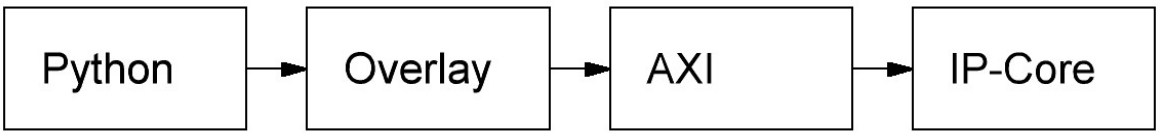

Abbildung 6.3: Überblick über die Implementierung

6.4.1 Quantisierung der Werte

Autor: Marvin Soldin

Da neuronale Netze üblicherweise mit Fließkomma-Operationen rechnen entstand die Überlegung die Darstellung der Zahlen zu verändern, damit eine kostengünstige Hardware-Architektur entsteht.

Aus diesem Grund haben wir uns dazu entschieden die Arithmetik auf Integer-Operationen zu begrenzen, also eine Quantisierung vorzunehmen. Dabei haben wir die entstandenen Gewichte mit dem Faktor 1000 multipliziert und die restlichen Kommastellen abgeschnitten.

Voraussetzung dafür war jedoch das alle Gewichte in dem Bereich von -1 bis 1 liegen, sodass folgende Konvertierung entstanden ist:

Software	Hardware
-1	-1000
-0.975	-975
-0.5	-500
...	...
0.5	500
0.975	975
1	1000

6.4.2 Training des neuronalen Netzes

Autor: Marvin Soldin

Um ein neuronales Netz überhaupt ausführen zu können, müssen die Gewichte der Verbindungen bekannt sein. Da es in diesem Projekt nur darum geht die Netze auf der Hardware auszuführen, können die Netze auch in Software modelliert und trainiert werden, siehe 6.1.4 auf Seite 36.

Dafür wurde die frei verfügbare Software Tensorflow in Kombination mit Keras genutzt.

Folgende Bedingungen mussten beim Trainieren beachtet werden:

- Möglichst einfache Aktivierungsfunktion (Exponentialfunktionen sind schwer in Hardware zu modellieren und müssen angenähert werden).
- Alle Gewichte müssen in dem Bereich von -1 bis 1 liegen, siehe 6.4.1 auf der vorherigen Seite
- Möglichst wenig Neuronen, damit Hardware gespart werden kann.

Als Aktivierungsfunktion wurde lediglich ReLU, siehe Gleichung 6 auf Seite 9, verwendet. Die Bereichsgrenzen konnten über sogenannte „Constraints“ abgesichert werden. Um den Bereich der Gewichte auf -1 bis 1 zu begrenzen, wurde die „MinMaxNorm“ verwendet. Diese gilt für den positiven und negativen Bereich, daher war es nicht möglich die Werte auf 0 bis 1 zu beschränken, sodass ein Datentyp mit negativen Bereich benutzt werden musste.

Die einzelnen Parameter wurden durch empirisches Austesten in Kombination mit Empfehlungen von Keras herausgefunden. Es handelt sich hierbei also nicht um nachgewiesene Parameter, die für dieses Beispiel am besten geeignet sind.

```
import numpy as np
from keras.models import Sequential
from keras.layers.core import Dense
from keras.constraints import MinMaxNorm
from keras.initializers import RandomUniform

tranining_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], "float32")
target_data = np.array([[0], [1], [1], [0]], "float32")

constraint = MinMaxNorm(min_value=0.0, max_value=1.0, rate=1.0, axis=0)
initializer = RandomUniform(minval=-0.05, maxval=0.05, seed=None)

model = Sequential()
model.add(Dense(2, input_dim=2, activation="relu", kernel_constraint=constraint,
    kernel_initializer=initializer, use_bias=False))
model.add(Dense(1, activation="relu", kernel_constraint=constraint,
    kernel_initializer=initializer, use_bias=False))

model.compile(loss='mean_squared_error', # Hat sich bei dem Testen als gut
    herausgestellt
              optimizer='sgd', # Stochastic Gradient Descent Methode
              metrics=['accuracy']) # Welche Metriken sollen bei dem Training
                  wiedergegeben werden

# Training des neuronalen Netzes ueber 1000 Epochen mit einer Eingabevariation von 4
    Kombinationen
model.fit(tranining_data, target_data, epochs=10000, batch_size=4, verbose=2)


for layer in model.layers:
    print(layer.get_weights()) # Ausgabe der Gewichte nur ueber print bisher
        moeglich.
```

Aus dem Training entstand nun das folgende Netz mit folgenden Gewichten:

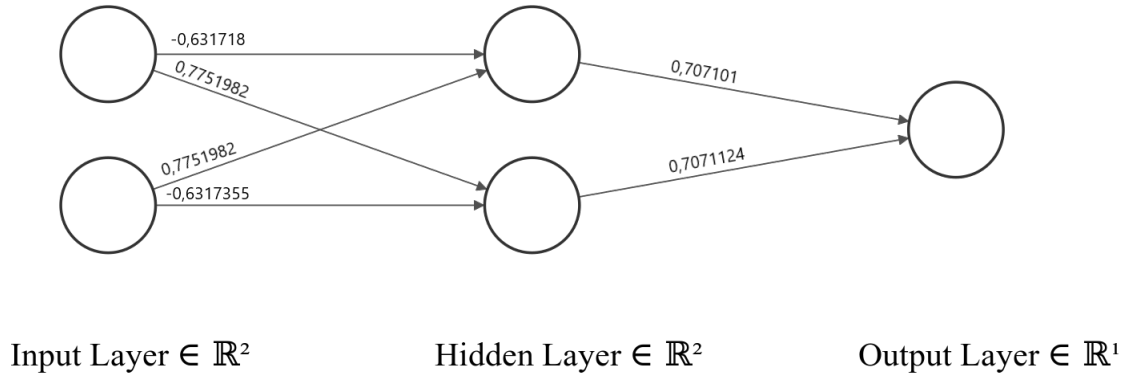

Abbildung 6.4: Topologie des neuronalen Netzes

6.4.3 Hardware-Architektur

Autor: Marvin Soldin

Zur einfachen Darstellung werden die einzelnen Module genauer erklärt, um danach den gesamten Datenfluss darzustellen. Wichtig ist, das bei der Entwicklung darauf geachtet wurde eine Wiederverwendbarkeit selbst für einfache Operationen wie der Multiplikation herzustellen.

Die Definition der Datenbereiche wird in der Datei layer_pkg.vhd angegeben, welche beliebig verändert werden kann. Dafür sind zwei Konstanten vorhanden:

1. bitwidth: Gibt an wie groß der Bitvektor des Datentypen ist.
2. normalization_factor: Gibt den Normalisierungsfaktor an, das heißt um wie viele Werte der Kommawert verschoben wurde.

Hierbei ist aber zu beachten, dass keine Fehlerüberprüfung bei falschen Werten stattfindet. In der Architektur wird ausschließlich mit diesem Wert gerechnet, das heißt, der Datentyp sollte niemals überlaufen. Dieses Problem haben wir mit der Darstellung von -1 bis 1 umgangen, da diese Werte multipliziert immer -1 bis 1 ergeben. Dennoch werden unsere Werte auf den Bereich -1000 bis 1000 abgebildet, sodass es kurzfristig zu höheren Werten kommen kann. Im schlimmsten Fall entsteht bei unserem Netz mit 2 Neuronen in Schicht eins, ein Maximalwert von 2.000.000 oder -2.000.000. Das heißt, wir müssen zum Berechnen mindestens 22 Bit verwenden, haben uns dennoch erst mal auf 32 Bit geeinigt, um die Zahlen ohne weite Konvertierung aus Python nutzen zu können.

Die 22 Bit kommen zustande, da die Normierung erst nach der Summierung einer Schicht erfolgt. Ein besserer Ansatz, wäre es direkt nach der Multiplikation zu normieren, sodass maximal ein Wert von 2000 oder -2000 entstehen kann. Dies hätte zur Folge das man nur noch 12 Bit zur Darstellung benötigt.

Das neuronale Netz besteht in der Hardware aus einem feed_forward_layer.vhd und einem output_layer.vhd. Die Auftrennung war nötig da bei der Ausgabe keine Matrizenmultiplikation mehr stattfindet, sondern nur noch die Berechnung eines Skalarproduktes.

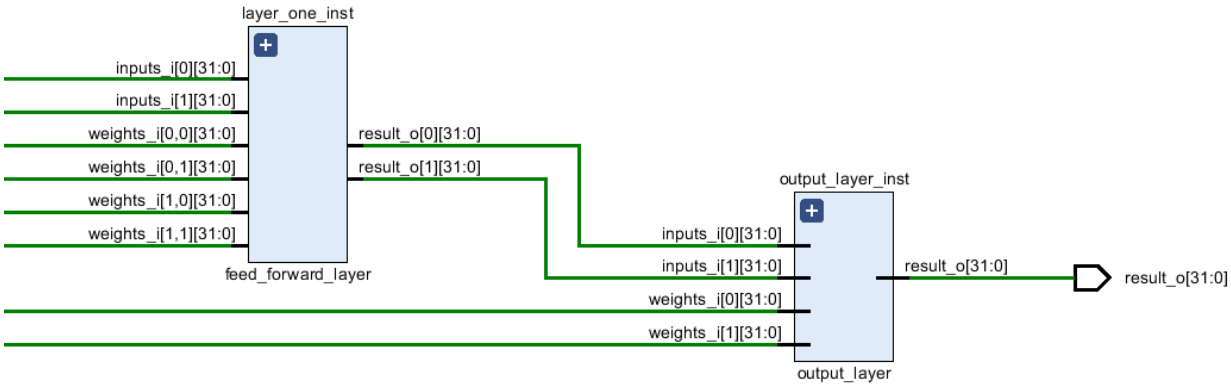

Abbildung 6.5: Architektur des neuronalen Netzes

Aufbau eines Feed Forward Layers

1. Source File: feed_forward_layer.vhd
2. Generische Attribute:
   (a) ROWS : Gibt an, wie viele Reihen die Gewichtsmatrix hat.
   (b) COLUMNS : Gibt an, wie viele Spalten die Gewichtsmatrix und der Eingabevektor haben.
3. Schnittstellen:
   (a) weights_i : Gewichtsmatrix mit der Größe ROWSxCOLUMNS
   (b) inputs_i : Eingabevektor mit der Größe COLUMNS
   (c) result_o : Ausgabevektor mit der Größe ROWS
4. Genutzte Module:
   (a) mat_mult.vhd
   (b) normalization_layer.vhd
   (c) relu_layer.vhd

Der Ablauf kann in drei Schritte unterteilt werden:

1. Die Gewichtsmatrix wird mit dem Eingabevektor multipliziert und auf den internen Vektor mult_result abgebildet.
2. Der interne Vektor mult_result wird normalisiert und das Ergebnis auf den Vektor normalization_result abgebildet.
3. Der interne Vektor normalization_result wird in die Aktivierungsfunktion weitergeleitet und das Ergebnis wird an den Ausgabevektor result_o weitergeleitet.

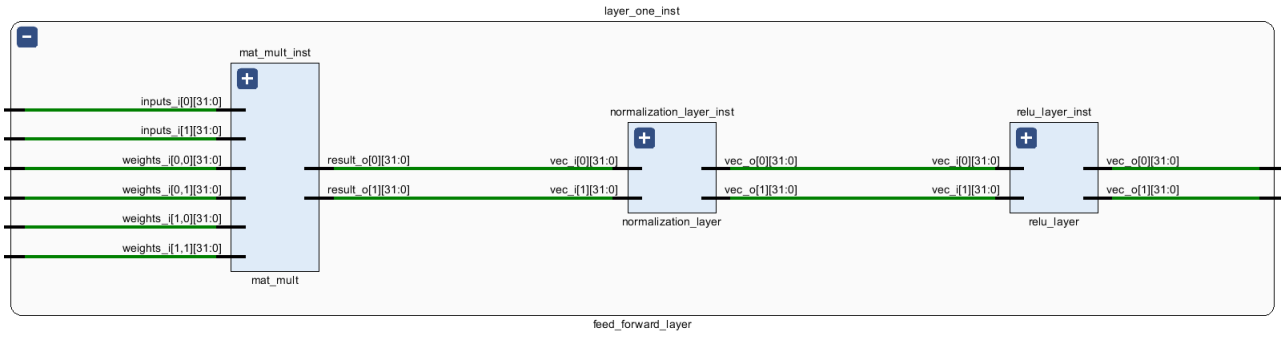

Abbildung 6.6: Aufbau eines Feed-Forward-Layer

Aufbau eines Output-Layers

1. Source File: output_layer.vhd
2. Generische Attribute:
   (a) VECTOR_SIZE : Gibt an, wie viele Reihen die Eingabevektoren haben.
3. Schnittstellen:
   (a) weights_i : Gewichte Eingabevektor mit der Größe VECTOR_SIZE
   (b) inputs_i : Eingabevektor mit der Größe VECTOR_SIZE
   (c) result_o : Ausgabewert des definierten Datentyps t_number
4. Genutzte Module:
   (a) dot_product_.vhd
   (b) normalization_unit.vhd
   (c) relu_unit.vhd
5. Ablauf:
   (a) Das Skalarprodukt der Eingabevektoren wird berechnet und das Ergebnis auf die interne Variable dot_product_result abgebildet.
   (b) Der interne Wert dot_product_result wird normalisiert und das Ergebnis auf den internen Wert normalization_result abgebildet.
   (c) Der interne Wert normalization_result wird in die Aktivierungsfunktion weitergeleitet und das Ergebnis an den Ausgabewert result_o weitergeleitet.

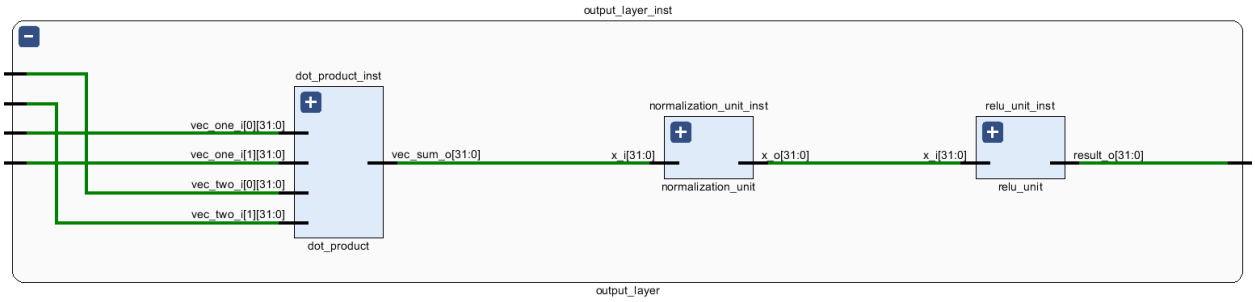

Abbildung 6.7: Aufbau eines Output-Layer

1

[1]Da bei einer einfachen Entscheidung zwischen 0 und 1 nur ein Skalarprodukt nötig ist benötigt diese Schicht keine Matrizenmultiplikation. Dies wäre bei einem Klassifikator mit mehreren Ausgaben nicht der Fall, sodass dort ein weiteres Feedforward Layer genutzt werden könnte.

Aufbau der Matrizenmultiplikation

1. Source File: mat_mult.vhd
2. Generische Attribute:
   (a) ROWS : Gibt an, wie viele Reihen die Matrix hat.
   (b) COLUMNS : Gibt an, wie viele Spalten die Matrix und der Eingabevektor haben.
3. Schnittstellen:
   (a) weights_i : Eingabematrix mit der Größe ROWSxCOLUMNS
   (b) inputs_i : Eingabevektor mit der Größe COLUMNS
   (c) result_o : Ausgabevektor mit der Größe ROWS
4. Genutzte Module:
   (a) dot_product.vhd
5. Ablauf:
   (a) Für jede Reihe * Spalte Multiplikation wird ein Skalarprodukt berechnet und das Ergebnis in dem korrespondierenden Feld vom Ausgabevektor abgebildet.

   Aufgrund der Berechnung von unterschiedlichen Skalarprodukten kann die Matrizenmultiplikation parallelisiert werden anstatt sie sequenziell auszuführen.

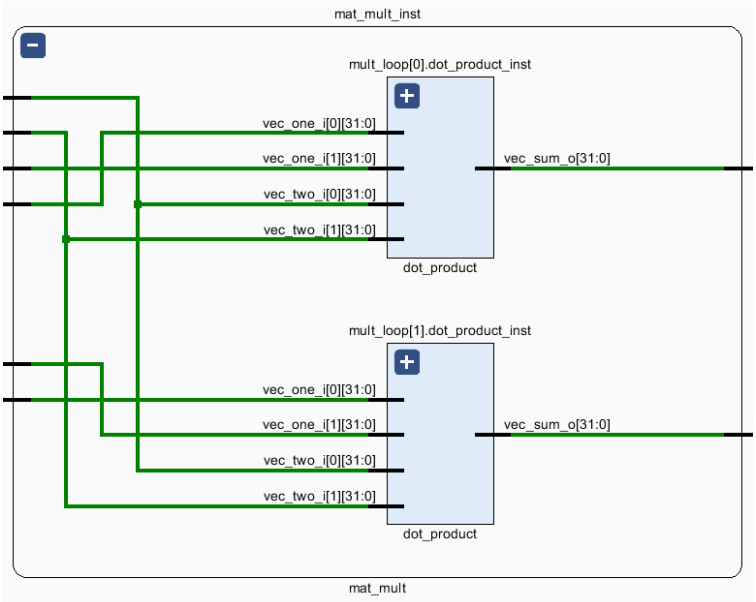

Abbildung 6.8: Aufbau der Matrizenmultiplikation

2

[2]Es handelt sich hierbei um keine „echte“ Matrizenmultiplikation, sondern lediglich um eine Multiplikation von einer Matrix mit einem Vektor.

Aufbau des Skalarproduktes

1. Source File: dot_product.vhd
2. Generische Attribute:
   (a) VECTOR_SIZE : Gibt an, wie groß die Eingabevektoren sind.
3. Schnittstellen:
   (a) vec_one_i : Eingabevektor mit der Größe VECTOR_SIZE
   (b) vec_two_i : Eingabevektor mit der Größe VECTOR_SIZE
   (c) vec_sum_o : Ausgabewert vom Datentypen t_number
4. Genutzte Module:
   (a) signed_mult.vhd
   (b) signed_vector_sum.vhd
5. Ablauf:
   (a) Die einzelnen Werte der Vektoren werden miteinander multipliziert und im internen Vektor multiplication_result abgebildet.

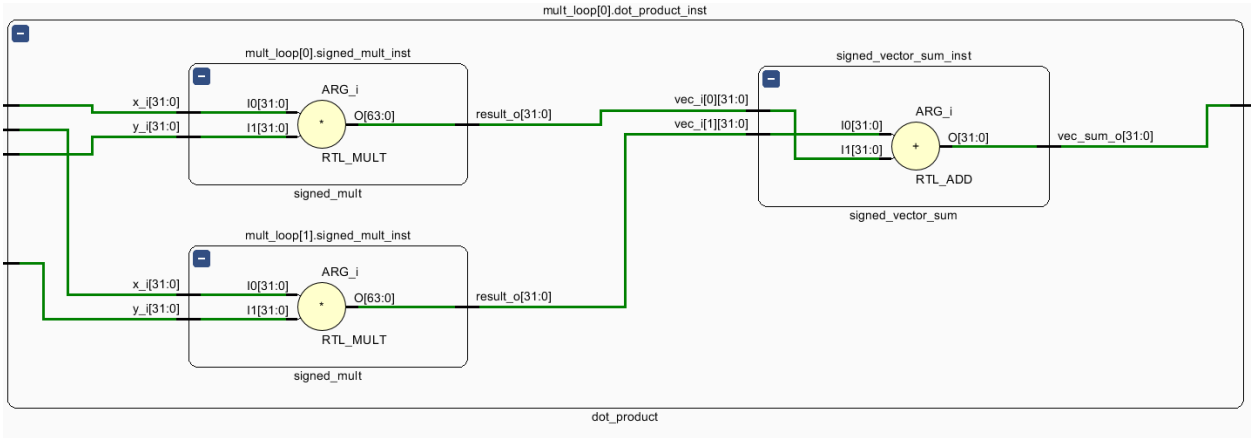

Abbildung 6.9: Aufbau des Skalarproduktes

Aufbau der Normalisierungsschicht

1. Source File: normalization_layer.vhd
2. Generische Attribute:
   (a) VECTOR_SIZE : Gibt an, wie groß der Eingabevektor ist.
3. Schnittstellen:
   (a) vec_i : Eingabevektor mit der Größe VECTOR_SIZE
   (b) vec_o : Ausgabevektor mit der Größe VECTOR_SIZE
4. Genutzte Module:
   (a) normalization_unit.vhd
5. Ablauf:
   (a) Die einzelnen Werte des Eingabevektoren werden normalisiert und auf den Ausgabevektor vec_o ausgegeben.

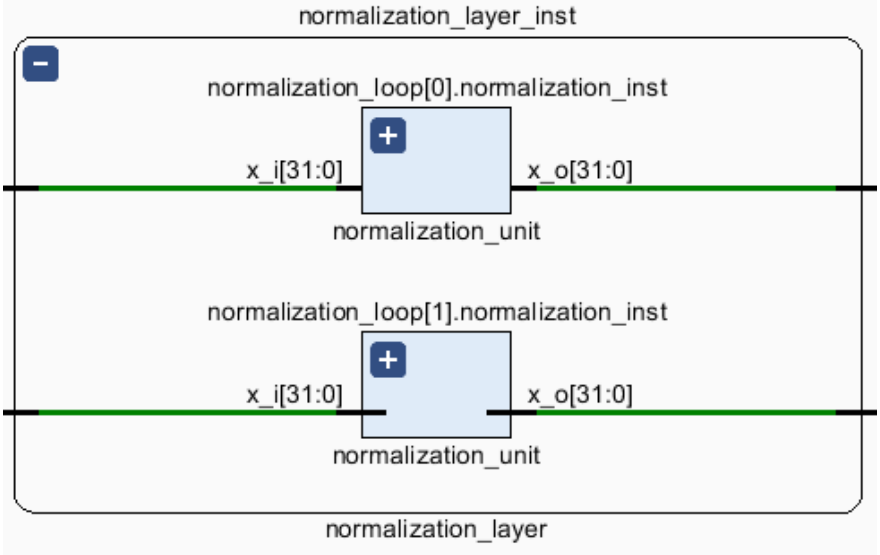

Abbildung 6.10: Aufbau der Normalisierungsschicht

Aufbau der Normalisierungseinheit

1. Source File: normalization_unit.vhd
2. Schnittstellen:
   (a) x_i Eingabewert vom Datentyp t_number
   (b) x_o Ausgabewert vom Datentyp t_number
3. Ablauf:
   (a) Der Eingabewert x_i wird durch den Normalisierungsfaktor dividiert.

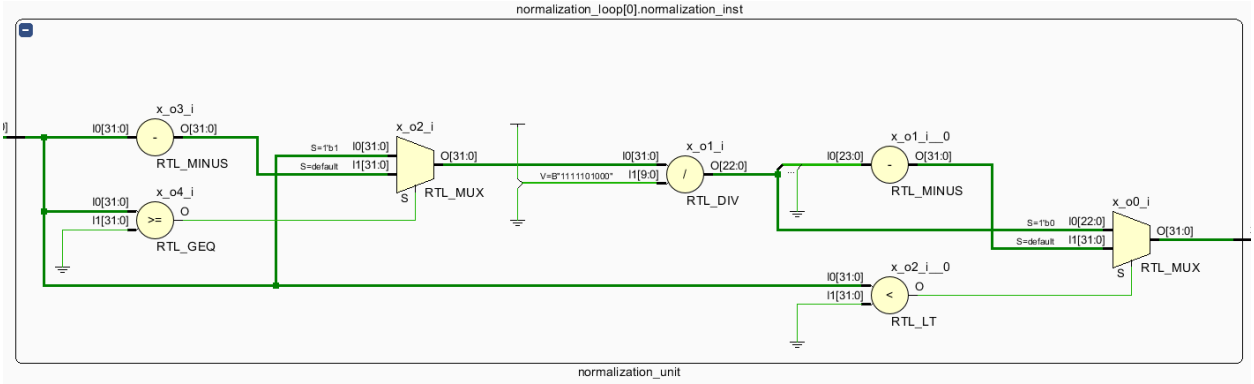

Abbildung 6.11: Aufbau der Normalisierungseinheit

3

[3]Im Falle des XOR-Beispiels wird der Eingabewert durch den Faktor 1000 geteilt, um wieder einen Wert zu erhalten der in dem Bereich von -1000 bis 1000 liegt.

Aufbau der Multiplikation-Berechnungseinheit

1. Source File: signed_mult.vhd
2. Schnittstellen:
   (a) x_i: Eingabewert vom Datentyp t_number
   (b) y_i: Eingabewert vom Datentyp t_number
   (c) result_o: Ausgabewert vom Datentyp t_number
3. Ablauf:
   (a) Die Eingabewerte x_i und y_i werden vorzeichenrichtig multipliziert.

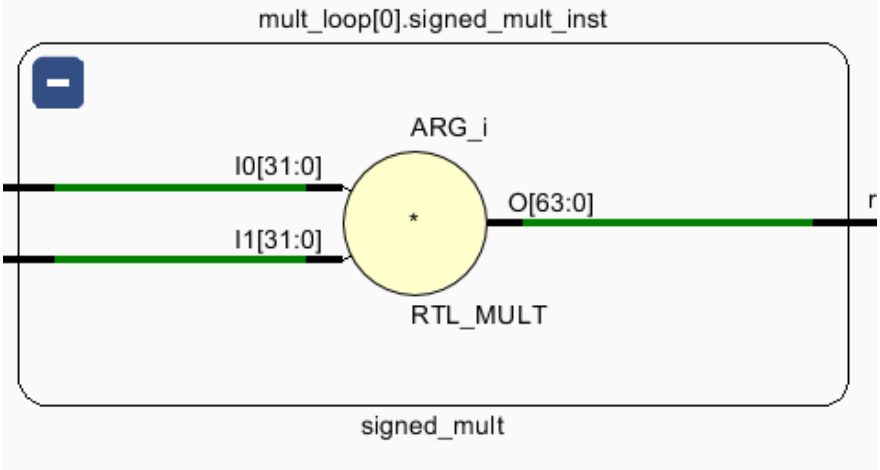

Abbildung 6.12: Aufbau der Multiplikation-Berechnungseinheit

Aufbau der ReLU-Berechnungsschicht

1. Source File: relu_layer.vhd
2. Schnittstellen:
   (a) vec_i
   (b) vec_o
3. Ablauf:
   (a) Die Eingabewerte vom Vektor vec_i werden in separate ReLu Funktionen weitergeleitet und die Ausgabewerte werden in den Vektor vec_o geschrieben.

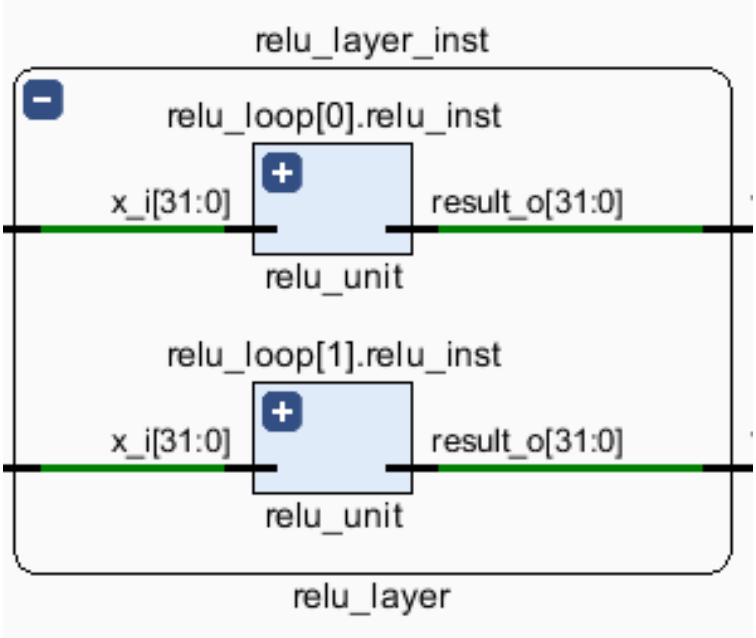

Abbildung 6.13: Aufbau der ReLU-Berechnungsschicht

Aufbau der ReLU-Berechnungseinheit

1. Source File: relu_unit.vhd
2. Schnittstellen:
   (a) x_i
   (b) result_o
3. Ablauf:
   (a) Es wird die Aktivierungsfunktion auf den Eingabewert x_i und das Ergebnis auf den Ausgabewert result_o geschrieben.

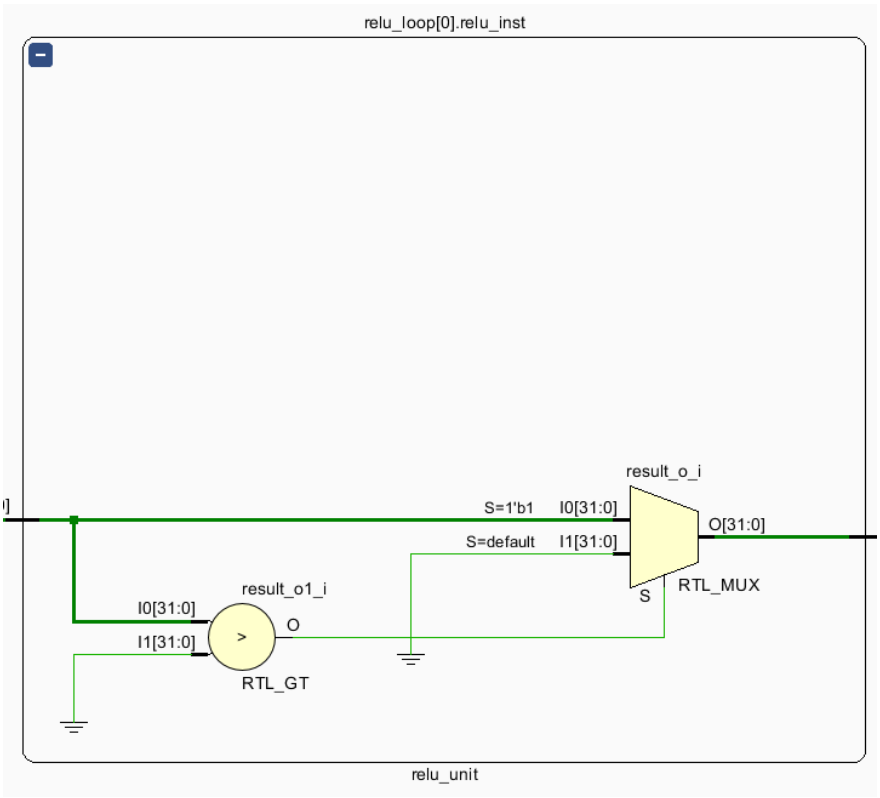

Abbildung 6.14: Aufbau der ReLU-Berechnungseinheit

Aufbau der Vektorsumme

1. Source File: signed_vector_sum.vhd
2. Generische Attribute:
   (a) VECTOR_SIZE : Gibt an, wie groß der Eingabevektor ist.
3. Schnittstellen:
   (a) vec_i : Eingabevektor mit der Größe VECTOR_SIZE
   (b) vec_sum_o : Aufsummierter Vektor als Datentyp t_number
4. Ablauf:
   (a) Die einzelnen Werte des Eingabevektoren werden sequentiell zusammengerechnet und das Ergebnis auf den Ausgabewert vec_sum_o geschrieben.

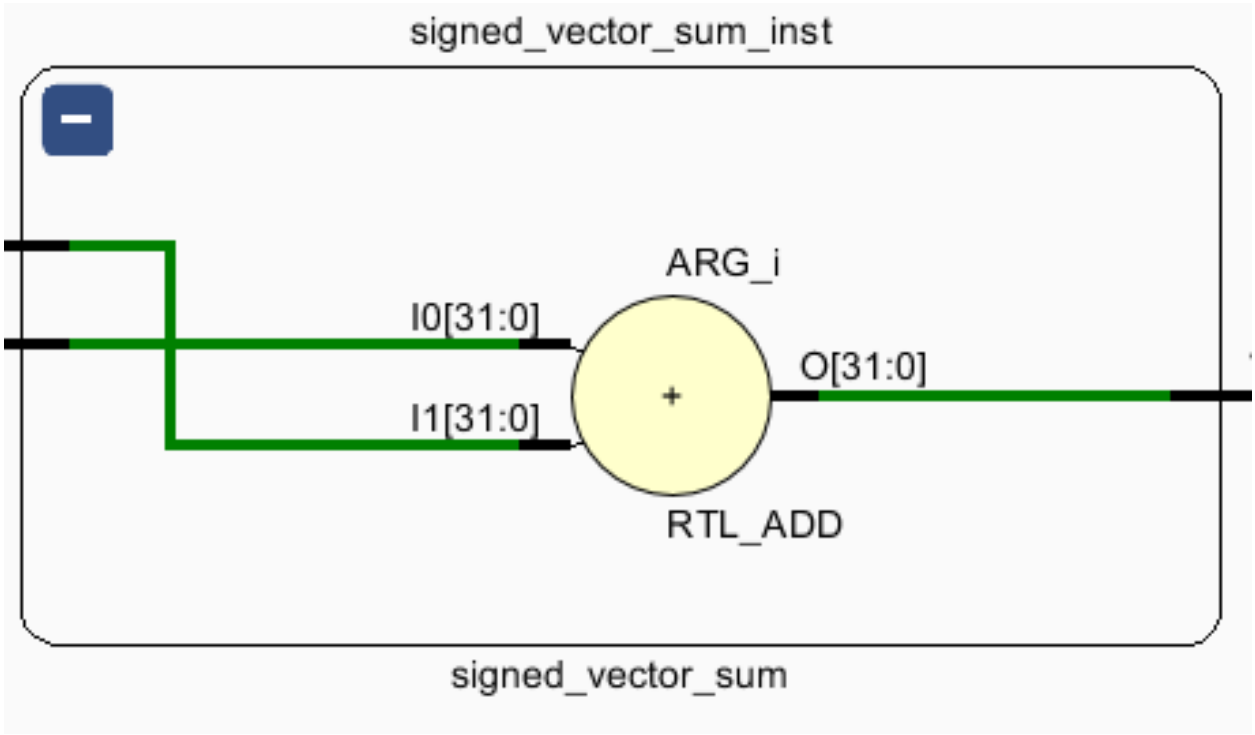

Abbildung 6.15: Aufbau der Vektorsumme

6.4.4 Testen der Hardware-Architektur

Autor: Colin von Huth

Um die Hardware-Architektur testen zu können, wurden entsprechende Testbenches für jede Komponente und für das Gesamtsystem geschrieben. Diese Testbenches testen die grundlegende Funktionalität und stellen eine Möglichkeit zur Simulation einzelner Komponenten oder des Gesamtsystems dar.

Das Testziel für die Testbenches besteht darin, stichprobenartig die Funktionalität nachzuweisen. Ein Austesten, also die Behandlung aller möglichen Eingaben, ist in diesem Fall nicht notwendig, da lediglich die grundlegende Funktionalität getestet werden soll.

Der Quellcode befindet sich, wie in Kapitel D.3 auf Seite 83 beschrieben, im entsprechenden Unterverzeichnis in der beigelegten ZIP-Datei.

Normalisierungseinheit

Um die Normalisierungseinheit zu testen, wird mit einer Schleife der Zahlenbereich eines 16-Bit-Integers (-32768 bis 32767) an den Eingang angelegt. Es wird davon ausgegangen, dass die Normalisierungseinheit sich für größere Zahlenbereiche analog verhält. In der Implementierung des XOR (Kapitel 6.4 auf Seite 42) wird ein 32-Bit-Integer verwendet. Dieser größere Zahlenbereich wird nicht getestet, da die Anzahl der Schleifendurchläufe und somit die erforderliche Zeit für die Durchführung des Tests exponentiell steigen würde.

Für jeden Eingabewert wird geprüft, ob das Ergebnis am Ausgang durch den Normalisierungsfaktor geteilt wurde. Falls dies nicht der Fall ist, wird ein Fehler ausgegeben.

Ausschnitt aus dem Quellcode:

```
test_loop: for i in -32768 to 32767 loop
    x_i <= to_signed(i, t_number'length);
    wait for clk_period;
    assert(x_o = x_i / normalization_factor)
        report "x_o is not right! x_i = " & integer'image(i) &
        "; x_o = " & integer'image(to_integer(x_o)) severity failure;
end loop test_loop;
```

Der vollständige Quellcode ist in der Datei *normalisation_unit_tb.vhd*, wie unter D.3 auf Seite 83 beschrieben, zu finden.

Skalarprodukt-Einheit

Die Skalarprodukt-Einheit wird Stichprobenartig mit zwei unterschiedlichen Multiplikationen von Vektoren mit jeweils zwei Komponenten getestet. Die Testfälle für die Skalarprodukte sind:

$$\begin{pmatrix}1\\2\end{pmatrix} * \begin{pmatrix}3\\4\end{pmatrix} = 11 \tag{6.3}$$

$$\begin{pmatrix}2\\2\end{pmatrix} * \begin{pmatrix}2\\2\end{pmatrix} = 8 \tag{6.4}$$

Ausschnitt aus dem Quellcode für den ersten Testfall:

```
-- Test: (1 2) * (3 4) = 11
vec_one_i <= (to_signed(1, t_number'length), to_signed(2, t_number'length));
vec_two_i <= (to_signed(3, t_number'length), to_signed(4, t_number'length));
wait for clk_period;
assert(11 = to_integer(vec_sum_o))
    report "vec_sum_o is not 11! vec_sum_o = " &
    integer'image(to_integer(vec_sum_o)) severity note;
```

Der vollständige Quellcode ist in der Datei *dot_product_tb.vhd*, wie unter D.3 auf Seite 83 beschrieben, zu finden.

Matrizenmultiplikationseinheit

Die Matrizenmultiplikationseinheit wurde mit vier unterschiedlichen Testfällen getestet. Der erste Testfall soll eine grundlegende Funktionalität testet. Hierfür wurde die folgende, einfache Matrizenmultiplikation durchgeführt:

$$\begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix} * \begin{pmatrix} 2 \\ 2 \end{pmatrix} = \begin{pmatrix} 2*2+2*2 \\ 2*2+2*2 \end{pmatrix} = \begin{pmatrix} 8 \\ 8 \end{pmatrix} \tag{6.5}$$

Der zweite Testfall stellt ebenfalls eine einfache Funktionsprüfung dar. Die Multiplikation einer Matrix mit einem Nullvektor ergibt immer einen Nullvektor. Die folgende Multiplikation wird getestet:

$$\begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix} * \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 2*0+2*0 \\ 2*0+2*0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{6.6}$$

Der dritte Testfall testet mit unterschiedlichen Elementen in der Matrix, ob die Multiplikation korrekt ausgeführt wird. Hiermit könnten mögliche Vertauschungen im Algorithmus ausgeschlossen werden. Die folgende Multiplikation wird getestet:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} * \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 1*1+2*2 \\ 3*1+4*2 \end{pmatrix} = \begin{pmatrix} 5 \\ 11 \end{pmatrix} \tag{6.7}$$

Der vierte Testfall soll testen, ob Vorzeichen korrekt behandelt werden. Dafür wurde die Matrix und der Vektor so gewählt, dass die folgenden Fälle getestet werden:

- Positive Zahl multipliziert mit einer positiven Zahl
- Positive Zahl multipliziert mit einer negativen Zahl
- Negative Zahl multipliziert mit einer negativen Zahl
- Negative Zahl multipliziert mit einer positiven Zahl

Die folgende Multiplikation wurde getestet:

$$\begin{pmatrix} -5 & 16 \\ 100 & -42 \end{pmatrix} * \begin{pmatrix} 15 \\ -6 \end{pmatrix} = \begin{pmatrix} (-5*15)+(16*-6) \\ (100*15)+(-42*-6) \end{pmatrix} = \begin{pmatrix} (-75)+(-96) \\ 1500+252 \end{pmatrix} = \begin{pmatrix} -171 \\ 1752 \end{pmatrix} \tag{6.8}$$

Ausschnitt aus dem Quellcode für den vierten Testfall:

```
weights_i <= (
    (to_signed(-5, t_number'length), to_signed(16, t_number'length)),
    (to_signed(100, t_number'length), to_signed(-42, t_number'length))
);
vec_i <= (to_signed(15, t_number'length), to_signed(-6, t_number'length));
wait for clk_period;
assert((-171 = to_integer(vec_o(0))) and (1752 = to_integer(vec_o(1))))
    report "vec_o is not (-171 1752)" & "vec_o = ("
    & integer'image(to_integer(vec_o(0))) & " "
    & integer'image(to_integer(vec_o(1))) & ")" severity failure;
```

Der vollständige Quellcode ist in der Datei *mat_mult_tb.vhd*, wie unter D.3 auf Seite 83 beschrieben, zu finden.

ReLU-Einheit

Für den Test der ReLU-Funktion werden mit zwei Schleifen jeweils der negative und der positive Zahlenbereich eines 16-Bit-Integers an den Eingang angelegt. Es wurde hierfür ebenfalls nicht der Zahlenbereich eines 32-Bit-Integers getestet, da, wie schon bei der Testbeschreibung der Normalisierungseinheit erwähnt wurde, die Anzahl der Schleifendurchläufe und somit die Dauer des Tests exponentiell steigen würde.

Mit der ersten Schleife wird demnach der Zahlenbereich von -32768 bis 0 an den Eingang gelegt. Hierbei soll das Ergebnis immer gleich 0 sein. Falls dies nicht der Fall ist, wird ein Fehler ausgegeben. Mit der zweiten Schleife wird der Zahlenbereich von 1 bis 32767 an den Eingang gelegt. Das Ergebnis sollte jeweils der Eingabe entsprechen. Falls dies nicht zutrifft, wird ebenfalls ein Fehler ausgegeben.

Der vollständige Quellcode ist in der Datei *relu_unit_tb.vhd*, wie unter D.3 auf Seite 83 beschrieben, zu finden.

Vektorsumme

Um die Vektorsumme zu testen, wurde stichprobenartig die Aufsummierung eines Vektors getestet:

$$VECSUM(\begin{pmatrix} 7 \\ 23 \end{pmatrix}) = 7 + 23 = 30 \tag{6.9}$$

Es wird der Vektor als Eingang angelegt und das Ergebnis überprüft. Falls die Ausgabe nicht dem erwarteten Ergebnis entspricht, wird ein Fehler ausgegeben.

Der vollständige Quellcode ist in der Datei *signed_vector_sum_tb.vhd*, wie unter D.3 auf Seite 83 beschrieben, zu finden.

Vorzeichenbehaftete Multiplikation

Um die vorzeichenbehaftete Multiplikation zu testen, wurde stichprobenartig eine Multiplikation getestet:

$$-2 * 8 = -16 \tag{6.10}$$

Es werden die Faktoren der Multiplikation an den Eingang angelegt und anschließend das Ergebnis überprüft. Falls die Ausgabe nicht dem erwarteten Ergebnis entspricht, wird ein Fehler ausgegeben.

Der vollständige Quellcode ist in der Datei *signed_mult_tb.vhd*, wie unter D.3 auf Seite 83 beschrieben, zu finden.

Gesamtsystem: Neuronales Netz

Um das gesamte neuronale Netz testen zu können, wurden die Gewichte des trainierten Netzes exportiert. Diese Gewichte werden als Initialisierung zunächst an die jeweilige Schicht übergeben. Anschließend werden die einzelnen Testfälle durchgegangen. Falls die Ausgabe bei einem Testfall nicht dem erwarteten Ergebnis entspricht, wird ein Fehler ausgegeben.

Die Testfolge des Gesamtsystems entspricht der Ausgabe von einem XOR:

IN1	IN2	OUT
0	0	0
0	1	1
1	0	1
1	1	0

Ausschnitt aus dem Quellcode, welcher die Initialisierung der Gewichte un den ersten Testfall zeigt:

```
-- Init weights
weights_layer_one_i <= (
    (to_signed(-631, t_number'length),
        to_signed(775, t_number'length)),
    (to_signed(775, t_number'length),
        to_signed(-631, t_number'length))
);
weights_layer_two_i <= (others => to_signed(707, t_number'length));

-- Test: 0 xor 0 = 0
inputs_i <= (to_signed(0, t_number'length),
    to_signed(0, t_number'length));
wait for clk_period;
assert(to_integer(result_o) < 500)
    report "0 xor 0 != 0!" severity failure;
```

Der vollständige Quellcode ist in der Datei *xor_net_tb.vhd*, wie unter D.3 auf Seite 83 beschrieben, zu finden.

6.4.5 Neuronales Netz als IP-Core

Autor: Tim Wieborg

Um das neuronale Netz nun AXI-Kompatibel zu machen, haben wir uns für eine Implementierung des XOR-Netzes als IP-Core entschieden. Der Vorteil hierbei ist, dass mittels eines Assistenten ein AXI-Kompatibles IP-Core-Template erstellt werden kann. In dieses Template wird dann der zusätzliche Code hinzugefügt und die Verbindung zu AXI geschlossen. Anschließend wird der IP-Core generiert und kann im Block Design hinzugefügt werden. Wie ein sog. AXI IP-Core erstellt wird, kann in Kapitel 5.6 auf Seite 32 gefunden werden.

Es wurde ein neues Vivado Projekt **XOR Net** angelegt und ein neuer AXI IP-Core namens **XOR Net IP** in der Version 2.0 erstellt.

Um die Gewichte der Neuronen variabel einstellen zu können, sowie die Ein- und Ausgabe über Python regeln zu können, wurde das AXI-Lite Interface mit 16 Registern über den Assistenten erstellt. Zu diesem Zeitpunkt werden zwar nur 11 Register benötigt, es schadet aber nie, für die Zukunft noch einige Register zur Verfügung zu haben.

Die Register sind belegt wie in Tabelle 6.4.5.

Register	Adresse	Nutzung
slv_reg0	0x00	Gewicht Layer 1(0, 0)
slv_reg1	0x04	Gewicht Layer 1(0, 1)
slv_reg2	0x08	Gewicht Layer 1(1, 0)
slv_reg3	0x0C	Gewicht Layer 1(1, 1)
slv_reg4	0x10	Gewicht Layer 2 (0)
slv_reg5	0x14	Gewicht Layer 2 (1)
slv_reg6	0x18	Eingang 1
slv_reg7	0x1C	Eingang 2
slv_reg13	0x34	Start Flag
slv_reg14	0x38	Fertig Flag
slv_reg15	0x3C	Ergebnis

Dabei ist zu betonen, dass die Register 14 und 15 nicht direkt genutzt werden. Stattdessen wird, wenn über das AXI Protokoll eines der genannten Register angefragt wird, einfach ein anderes Signal ausgegeben. Wird nach dem *slv_reg14* gefragt, wird z.B. stattdessen das „Fertig Flag“ zurückgegeben.

Anschließend wurden alle für das XOR-Network benötigten Ressource-Dateien in das IP-Core Projekt hinzugefügt. Es handelt sich um alle .VHD-Dateien aus dem XOR-Kapitel.

In der Datei *xor_net_ip_c2_0_S00_AXI_inst* wird nun in Zeile 6 *use work.layer_pkg.all;* hinzugefügt. Anschließend wird in der Architektur ab Zeile 91 die Komponentenbeschreibung des xor_net hinzugefügt. Diese sieht wie folgt aus:

```
component xor_net
port(
  weights_layer_one_i: in t_matrix(1 DOWNTO 0, 1 DOWNTO 0);
  weights_layer_two_i: in t_vector(1 DOWNTO 0);
  inputs_i: in t_vector(1 DOWNTO 0);
  result_o: out t_number
);
end component;
```

Zusätzlich werden folgende interne Signale ab Zeile 112 deklariert:

```
signal weights_layer_one:t_matrix(1 DOWNTO 0, 1 DOWNTO 0):= (
  others=>(others=>(others =>'0'))
);
signal weights_layer_two:t_vector(1 DOWNTO 0):=(others=>(others=>'0'));
signal inputs:t_vector(1 DOWNTO 0) := (others => (others => '0'));
signal result:t_number := (others => '0');
signal counter:integer := 0;
signal start_flag:std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
signal finish_flag:std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
```

Wie bereits beschrieben, sollen das *finish_flag* und *result* statt den Registern *slv_reg14* und *slv_reg15* ausgegeben werden. Dazu werden die Zeilen 539 und 541 folgendermaßen abgeändert:

```
539     reg_data_out <= std_logic_vector(finish_flag);
541     reg_data_out <= std_logic_vector(result);
```

Ab Zeile 568 wird nun eine Instanz der xor_net Komponente erzeugt:

```
xor_net_inst: xor_net
port map(
  weights_layer_one_i => weights_layer_one,
  weights_layer_two_i => weights_layer_two,
  inputs_i => inputs,
  result_o => result
);
```

Durch bereits vorher erfolgte Testversuche ist aufgefallen, dass das neuronale Netz über 70 nS benötigt, um das korrekte Ergebnis zu berechnen. Aus diesem Grund wird ein Zähler eingebaut, der das Ergebnis erst beim achten Takt nach beginn der Berechnung in das *result* Signal schreibt. Das ganze Verhalten des Rechennetzes kann wie folgt beschrieben werden:

Hinweis: Der IP-Core ist komplett taktflankengesteuert. In folgender Beschreibung wird immer von einer steigenden Taktflanke ausgegangen, wenn etwas gesetzt oder abgerufen wird.

Sobald an *slv_reg13* eine '1' liegt, wird eine '1' auf das *start_flag* geschoben. Der Prozess dazu sieht wie folgt aus:

```
process(S_AXI_ACLK, slv_reg13)
begin
  if (rising_edge(S_AXI_ACLK)) then
    if (slv_reg13 = std_logic_vector(to_signed(1,C_S_AXI_DATA_WIDTH))) then
      start_flag <= slv_reg13;
    else
      start_flag <= (others => '0');
    end if;
  end if;
end process;
```

Ein weiterer Prozess ist für den Zähler zuständig. Wenn das *start_flag* auf '1' ist, wird der Zähler um 1 erhöht, bis dieser den Wert 8 erreicht hat. Dann wird der Zähler zurückgesetzt und das *finish_flag* wird auf '1' gesetzt. Sollte das *start_flag* vorzeitig wieder auf '0' (oder jeden anderen Wert außer 1) gesetzt werden, wird der Zähler sowie das *finish_flag* zurückgesetzt.

Der Prozess dazu sieht wie folgt aus:

```
process(S_AXI_ACLK, start_flag)
begin
  if (rising_edge(S_AXI_ACLK)) then
    if (start_flag = std_logic_vector(to_signed(1,C_S_AXI_DATA_WIDTH))) then
      if (counter >= 8) then
        counter <= 0;
        finish_flag <= (0 => '1', others => '0');
      else
        counter <= counter + 1;
      end if;
    else
      finish_flag <= (others => '0');
      counter <= 0;
    end if;
  end if;
end process;
```

Der letzte Prozess ist nun zum Übergeben der Gewichte sowie der Eingangswerte zum Neuronalen Netz zuständig. Nur wenn der Zähler auf 0, sowie das *start_flag* auf '1' ist, werden die Werte gesetzt. Das Ergebnis der Berechnung wird direkt aus der xor_net-Komponente an das *output*-Signal gekoppelt. Auf Softwareebene wird später das Ergebnis ausgelesen, wenn as *finish_flag* auf '1' ist.

Der Prozess sieht folgendermaßen aus:

```
process(S_AXI_ACLK, start_flag, counter)
begin
  if (rising_edge(S_AXI_ACLK)) then
    if (start_flag = std_logic_vector(to_signed(1,C_S_AXI_DATA_WIDTH))) then
      if (counter = 0) then
        weights_layer_one(0,0) <= signed(slv_reg0); -- 0x00
        weights_layer_one(0,1) <= signed(slv_reg1); -- 0x04
        weights_layer_one(1,0) <= signed(slv_reg2); -- 0x08
        weights_layer_one(1,1) <= signed(slv_reg3); -- 0x0C

        weights_layer_two(0) <= signed(slv_reg4); -- 0x10
        weights_layer_two(1) <= signed(slv_reg5); -- 0x14

        inputs(0) <= signed(slv_reg6); -- 0x18
        inputs(1) <= signed(slv_reg7); -- 0x1C
      end if;
    end if;
  end if;
end process;
```

Der IP-Core ist damit fertig. Im „Project Manager“ muss nun über „Edit Packaged IP“ dafür gesorgt, werden, dass alle „Packaging Steps“ mit einem Haken versehen sind. Anschließend kann das Projekt geschlossen werden und zum Hauptprojekt zurückgekehrt werden.

Im Hauptprojekt wird nun ein neues Block Design angelegt und der Standardaufbau aus den letzten Projekten erzeugt. Dann wird der neue IP-Core „xor_net_ip“ hinzugefügt und über das automatische Tool verbunden. Der Aufbau sollte nun wie in Abbildung 6.16 aussehen.

6.4.6 Python-Integration

Autor: Tim Wieborg

Damit das Neuronale Netz einfach über Python angesprochen werden kann, wird ein Custom Driver „XorDriver“ als Klasse implementiert. Mit diesem Driver kann dann über die Funktion *xor(a, b)* Mit den Eingabeparametern *a* und *b* das Neuronale Netz angesprochen werden.

Der Driver sorgt dann dafür, dass die Gewichte sowie die Eingabewerte in die Register geschrieben werden und das *Start Flag* gesetzt wird. Dann wird so lange gewartet, bis das *Finish Flag* gesetzt ist, sodass das Ergebnis ausgelesen werden kann. Anschließend wird *Start Flag* wieder zurückgesetzt und das Ergebnis aus der Funktion zurückgegeben.

Ein kleiner Ausschnitt aus der Driver Klasse (*xor_net_driver.py*):

Hinweis: Die Werte der Gewichte wurden vorher mit Tensorflow auf einem anderen Gerät berechnet und exportiert. Eine Übersicht kann unter 6.4.5 auf Seite 60 gefunden werden.

```
bindto = ['user.org:user:xor_net_ip:2.0']

def sendWeights(self):
  self.write(0x00, -631)
  self.write(0x04, 775)
  self.write(0x08, 775)
  self.write(0x0C, -631)
  self.write(0x10, 707)
  self.write(0x14, 707)

def xor(self, a, b):
  value = self.xorRaw(a, b)
  if value >= 500:
    return 1
  else:
    return 0

def xorRaw(self, a, b):
  self.isAccepted(a)
  self.isAccepted(b)

  self.sendWeights()
  self.write(0x18, int(a*1000))
  self.write(0x1C, int(b*1000))
  self.write(0x34, 1)

  while(self.read(0x38) == 0):
    print("Calculating")

  self.write(0x34, 0)
  return self.read(0x3c)
```

Diese Klasse inklusive der generierten Bitstream-Datei aus Vivado sowie das exportierte Block Design werden nun auf den PYNQ-Z1 unter */home/xilinx/pynq/overlays/xor_net/* abgelegt. Über den Webbrowser wird nun auf die Jupyter App des PYNQs zugegriffen und ein neues Notebook erstellt. Der Inhalt sieht wie folgt aus:

```
from pynq import Overlay

from pynq.overlays.xor import xor_net_driver
overlay = Overlay('/home/xilinx/pynq/overlays/xor/xor_net.bit')

xor_net = overlay.xor_net

?xor_net

xor_net.xor(1, 1) \\ Ausgabe: 0
xor_net.xor(1, 0) \\ Ausgabe: 1
```

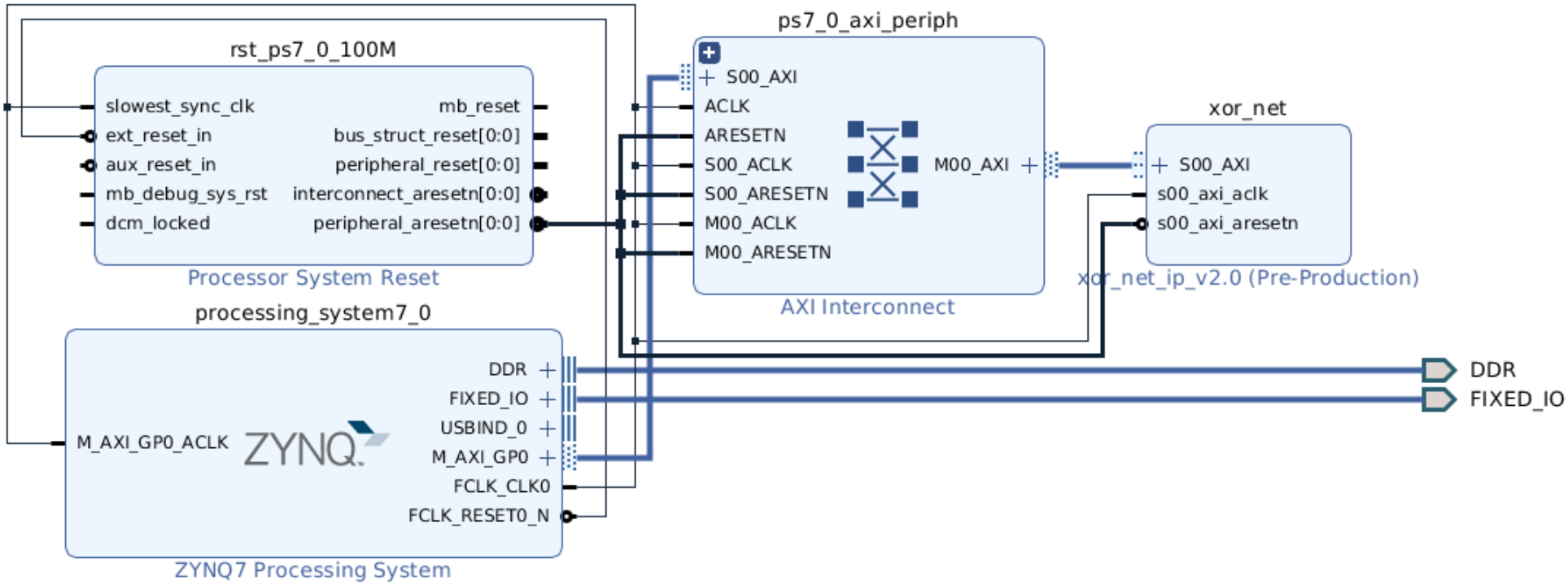

Abbildung 6.16: Block Design des Projekts „XOR-Network“

```
xor_net.xor(0, 1) \\ Ausgabe: 1
xor_net.xor(0, 0) \\ Ausgabe: 0
```

Ist die Python Abfrage nun schneller als die Berechnung auf dem FPGA, wird durch die *while*-Schleife in der *xorRaw()*-Funktion des Drivers „Calculating“ ausgegeben. Anschließend wird das Ergebnis angezeigt.

6.5 Umsetzung der Sigmoidfunktion

Autor: Ngwa Tingoh Kingsley

Einer der wichtigsten Teile eines Neurons ist seine Aktivierungsfunktion. Eine nichtlineare Aktivierungsfunktion macht es möglich, dass das neuronale Netz beliebige nichtlineare Funktionen lernt. Im folgenden Abschnitt wird ein effizienter Ansatz zur Approximation einer Sigmoid Aktivierungsfunktion vorgestellt. Es ist auch wichtig zu beachten, dass nach diesem Ansatz jede nichtlineare Aktivierungsfunktion approximiert werden kann. Um die Sigmoidfunktion in VHDL zu realisieren, wurde in Matlab ein Look-Up-Table(LUT) einer Sigmoidfunktion generiert.

Dazu wurde der folgende Matlab-Code verwendet.

```
x = -5:1:5;
y= 1./(1+exp(-x));
figure;
plot(x,y); title ('sigmoid')
h = findobj(gca, 'Type,'line')
x= get(h,'Xdata')
y= get(h, 'Ydata')
```

Der resultierende Plot ist auf Abbildung 6.17 zu sehen.

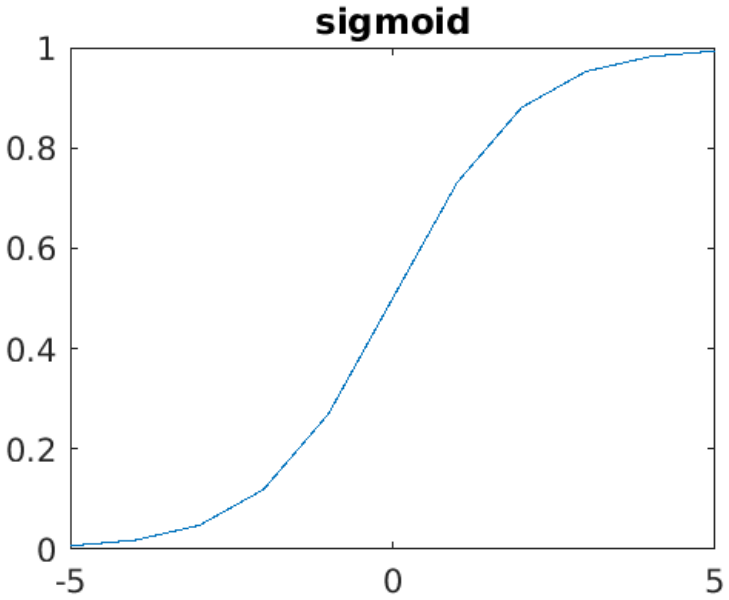

Abbildung 6.17: Matlab Sigmoid Plot

Der obige Matlab-Code zeigt, dass die X Werte im Bereich von -5 bis 5 mit einem Schritt von 1 liegen. Dieser Schritt wurde mit dem Ziel gewählt, die VHDL-Realisierung durch Reduzierung des Volumens der LUT zu vereinfachen. Die Y Werte liegen im Bereich von 0 bis 1, so dass der gesamte Plot die Aktivierungsfunktion Nummer 3, Abschnitt 2.2.2 auf Seite 9 vorgestellten Soft step entspricht. Es ist wichtig zu erwähnen, dass das Sigmoid auch als Soft step bezeichnet wird, wie auf 2.2.2 auf Seite 9 eingeführt wurde.

Um den resultierenden Plot besser erkennen zu können, wurde der obige Matlab-Code leicht angepasst:

```
x = -5:1:5;
y= 1./(1+exp(-x));
h = stairs(x,y);
figure;
plot(x,y); title ('sigmoid')
h = findobj(gca, 'Type,'line')
x= get(h,'Xdata')
y= get(h, 'Ydata')
```

Aus dem angepassten Code ergibt sich die folgende Darstellung 6.18.

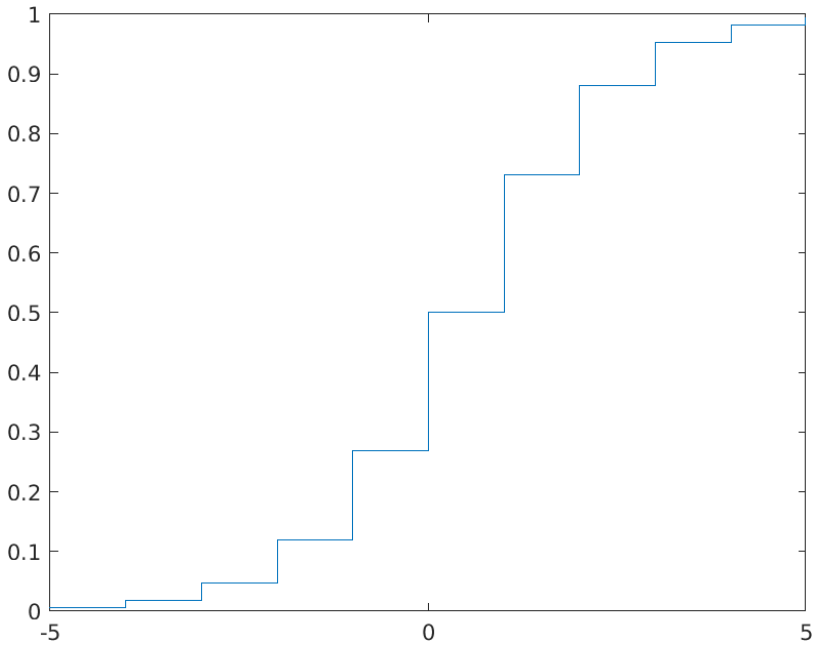

Abbildung 6.18: Matlab Sigmoid "Stairs" Plot.

Aus diesem Plot wurden die X- und Y-Werte extrahiert. Um die Arithmetik auf Integer-Operationen zu begrenzen, wurde eine Quantisierung vorgenommen. In dem Zusammenhang, wurden die Y und X werte mit dem Faktor 1000 multipliziert und die verbleibenden Kommastellen abgeschnitten. Die folgende Tabelle zeigt die Werte. Damit wird eine LUT einer Sigmoidfunktion in VHDL implementiert.

X	Y
-5000	6
-4000	18
-3000	47
...	...
3000	952
4000	982
5000	993

Der vollständige Quellcode ist, wie unter D.3 auf Seite 83 beschrieben, zu finden.

1. Verzeichnis: Additional_ activation_functions
2. Source File: sigmoid_unit.vhd.
3. Schnittstellen:
   (a) x_i
   (b) result_o
4. Ablauf:
   (a) Es wird die Aktivierungsfunktion auf den Eingabewert x_i und das Ergebnis auf den Ausgabewert result_o geschrieben

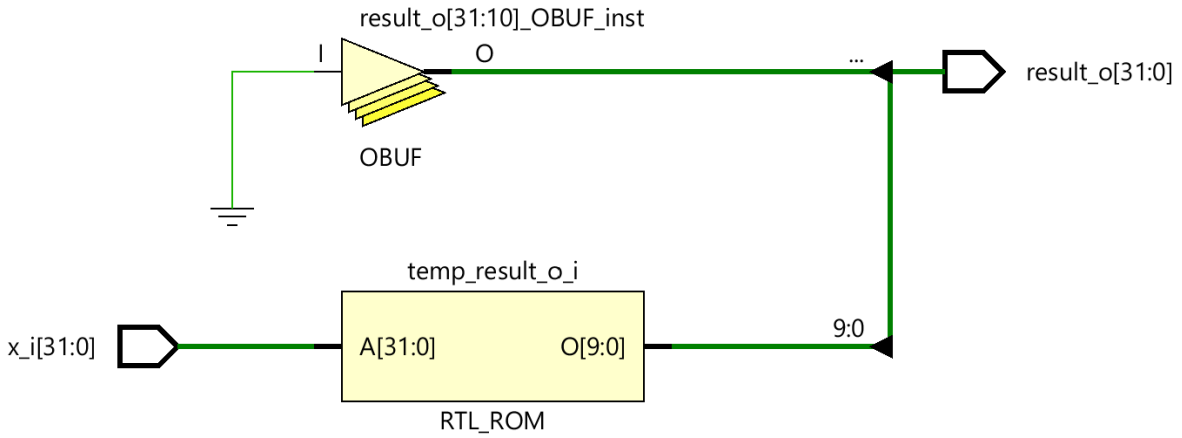

Abbildung 6.19: Aufbau der Sigmoid Funktion

Testen der Sigmoid-Einheit
Die Testbench wurde so geschrieben dass, jede gegebene Eingangskonstante mindestens einmal überprüft wurde, um sicherzustellen, dass der richtige konstante Ausgangswert zugewiesen wurde. Falls dies nicht der Fall ist, wird ein Fehler ausgegeben. Außerdem wurde stichprobenartig geprüft, dass jeder Wert außerhalb des Bereiches -5 bis 5 einen Default-Wert erhält.

Der vollständige Quellcode ist in der Datei *sigmoid_unit_tb.vhd*, wie unter D.3 auf Seite 83 beschrieben, zu finden.

Kapitel 7

Evaluierung

Autor: Colin von Huth

Um zu evaluieren, ob der Einsatz von FPGAs zur Ausführung von neuronalen Netzen sinnvoll ist, werden einige Kriterien analysiert:

- Ausführungsgeschwindigkeit
- Ressourcenverbrauch auf der Hardware
- Aufwand der Implementierung

7.1 Ausführungsgeschwindigkeit

Autor: Tim Wieborg

Um einen aussagekräftigen Vergleich der Ausführungsgeschwindigkeit zu erhalten, wurden folgende Tests durchgeführt:

- Vivado Timing Analyse des implementierten Schaltnetzes
- Geschwindigkeit der Gesamtlösung (Python + Schaltnetz)
- Geschwindigkeit einer Python Implementierung auf dem PYNQ-Z1
- Geschwindigkeit einer Python Implementierung auf einem durchschnittlichen Laptop

Abgesehen von der Vivado Timing Analyse wurde jeder Test 10000x ausgeführt und der Mittelwert aus allen Testdurchgängen gebildet, sodass externe Einflüsse so gut wie möglich herausgefiltert werden konnten.

Hinweis: Die Dateien können im Anhang unter D.3 auf Seite 83 gefunden werden.

7.1.1 Vivado Timing Analyse des implementierten Schaltnetzes

In Vivado wird bei jedem Synthetisierungsvorgang automatisch auch eine Timing-Analyse gemacht. Der Vorteil hierbei ist, dass eine Timing Analyse nicht jedes mal von Hand gestartet werden muss. Man erhält somit nach jedem Run direkt eine Übersicht, ob das Timing passt. Allerdings dauert ein Run dadurch auch etwas länger.

Erste Timing Analysen zeigten, dass erst nach über 60 ns das Schaltnetz komplett durchlaufen ist. Dies liegt an der Größe des Schaltnetzes. Somit wurde, wie unter 6.4.5 auf Seite 60 beschrieben, ein Zähler

mit eingebaut. Dieser Zähler ermöglicht es, die Ausgabe des Ergebnisses etwas zu Verzögern, sodass es erst mit dem 7. Takt nach beginn der Berechnung abgerufen werden kann.

Da das Schaltnetz nun mehr Zeit hat, um das Ergebnis zu berechnen, muss dies auch in der Timing Analyse beachtet werden. Diese sog. **Constraints** können über den „Flow Navigator“ unter „Implementation->Edit Timing Constraints“ gesetzt werden. Innerhalb des „Timing Constraints“-Tabs geht man auf „Set Multicycle Path“. Mit einem Rechtsklick auf „Create Constraint“ öffnet sich ein neues Fenster. Folgende Werte müssen nun gesetzt werde:

```
Path multiplier: 7
From: [get_pins -hierarchical -regexp -filter { NAME =~  ".*/C(LK)?$.*" &&
    PARENT_CELL =~  ".*xor_net_ip_v2.*" }]
Transition: rise/fall

To: [get_pins -hierarchical *rdata_reg*/D]
Transition: rise/fall
```

Zusätzlich muss in dem Options-Tab „Setup/Hold“ aktiviert und auf **setup** gesetzt werden. Mit „OK“ das Fenster schließen.

Anschließend muss ein weiteres Contraint mit folgenden Werten erstellt werden:

```
Path multiplier: 6
From: [get_pins -hierarchical -regexp -filter { NAME =~  ".*/C(LK)?$.*" &&
    PARENT_CELL =~  ".*xor_net_ip_v2.*" }]
Transition: rise/fall

To: [get_pins -hierarchical *rdata_reg*/D]
Transition: rise/fall
```

Auch dort unter „Options“ wieder „Setup/Hold“ aktivieren. Diesmal wird der Wert aber auf **hold** gesetzt. Mit „OK“ das Fenster schließen.

Wichtig: Erst durch das Speichern werden die Werte wirklich gesetzt. Es muss nun also via *Strg + S* oder durch klicken auf den „Speichern-Button“ das Projekt gespeichert werden. Nun sind die Multicycle Paths gesetzt, sodass ein erneuter Run gestartet werden kann. Nun wird bei der Timing Analyse beachtet, dass das Schaltnetz 7 Takte Zeit hat, um das Ergebnis auszugeben.

Das PL nutzt einen Takt von 100 Mhz, also dauert ein Takt 10 ns. Wie bereits erwähnt, kann das Ergebnis nach 7 Taktperioden, also nach 70 ns, über den AXI-Bus abgerufen werden. Man kann also sagen, dass das System etwa 70 ns zum Berechnen des Resultats benötigt.

7.1.2 Geschwindigkeit der Gesamtlösung

Um die durchschnittliche Ausführungsgeschwindigkeit der Gesamtlösung zu ermitteln, wurde der unter 6.4.6 auf Seite 63 genannte Driver aus der Datei *xor_net_driver.py* genutzt. Dieser wird auf dem Board in das Verzeichnis */home/xilinx/pynq/overlays/xor/* abgelegt, sodass es in anderen Python Skripten importiert werden kann. Zusätzlich werden noch die Bitstream, sowie die .tcl-Datei in das selbe Verzeichnis kopiert.

Das Skript für die Timing Analyse (*timing_xor_vhdl.py*) sieht wie folgt aus:

```
from pynq import Overlay
from pynq.overlays.xor import xor_net_driver
import numpy as np
import time
import random

overlay = Overlay('/home/xilinx/pynq/overlays/xor/xor_net.bit')
xor_net_vhdl = overlay.xor_net
runs = 10000
times = []
```

```
for i in range(runs):
  rand1 = random.randint(0,1)
  rand2 = random.randint(0,1)

  start = time.time()
  res = xor_net_vhdl.xor(rand1, rand2)
  end = time.time()

times.append(end-start)

print(f"Average time of {runs} runs: {np.average(times) * 1000 * 1000} us")
```

In der Schleife werden zuerst die beiden Eingabewerte zufällig für das XOR generiert. Anschließend wird die aktuelle Systemzeit gespeichert und die *xor()*-Funktion aus dem Custom Driver aufgerufen. Sobald die Funktion das Ergebnis zurückgegeben hat, wird erneut die Systemzeit gespeichert. Die Differenz aus der Endzeit und der Startzeit ist nun die benötigte Zeit der Gesamtlösung, um das Ergebnis zu berechnen. Diese Zeit wird in ein Array geschrieben, aus dem nachher der Mittelwert berechnet und ausgegeben wird.

Als Ergebnis wurde ein Mittelwert von etwa $370\mu s$ für die Gesamtlösung berechnet.

7.1.3 Geschwindigkeit einer Python Implementierung

Um einen Vergleich zu erhalten, wie schnell die Hardware-beschleunigte Version ist, wurde außerdem eine rein Software-implementierte XOR-Version in Python programmiert.

Benötigt wird lediglich das Python-Modul *NumPy*. Es werden die gleichen Gewichte und der gleiche Aufbau des neuronalen Netzes gewählt, damit ein Vergleich möglich ist. Das Python-Skript(*xor_net_custom.py*) sieht wie folgt aus:

```
import numpy as np

def xor(val1, val2):
  input_values = [val1 , val2]

  weights_layer_one = [[-0.631, 0.775], [0.775, -0.631]]
  weights_layer_two = [0.707, 0.707]

  res1 = np.dot(weights_layer_one, input_values)

  for val in range(len(res1)):
    if res1[val] <= 0:
      res1[val] = 0


  res2 = np.dot(weights_layer_two, res1)

  if res2 <= 0:
    res2 = 0

  if res2 > 0.5:
    return 1
  else:
    return 0
```

Auffällig ist direkt, wie einfach das neuronale Netz im Vergleich zur aufwändigen Hardware Implementation ist. Während für die Gesamtlösung mehrere Wochen zur Implementierung benötigt wurden, hat diese Implementation nur wenige Minuten gebraucht. Dennoch erfüllt sie den gleichen Zweck.

Da dieses Skript nicht an irgendeine Hardware gebunden ist, kann es auf sämtlichen Geräten ausgeführt werden, für die es Python3 und das NumPy-Paket gibt. Aus diesem Grund wurde der Test sowohl auf dem PYNQ-Z1 als auch auf einem durchschnittlichen Laptop ausgeführt.

Auch bei dieser Implementierung wurde genau wie bei 7.1.2 auf Seite 69 der Mittelwert aus 10000 Versuchen gebildet. Das Skript für die Testdurchläufe sieht ähnlich dem Skript unter 7.1.2 auf Seite 69 aus und kann unter dem Namen *timing_xor_custom.py* gefunden werden. Damit es ausgeführt werden kann, muss auf dem PYNQ-Z1 die oben genannte *xor_net_custom.py*-Datei in dem Verzeichnis */home/xilinx/pynq/overlays/xor/* liegen.

Durch das Ausführen des Skripts erhält man einen Mittelwert von $162.5\mu s$.

7.1.4 Vergleich der Resultate

Insgesamt betrachtet sind tatsächlich sehr unterschiedliche Ergebnisse zustande gekommen. Das komplett in Python implementierte neuronale Netz ist mehr als doppelt so schnell ($162.5\mu s$), als die Gesamtlösung bestehend aus einem Softwareanteil geschrieben in Python und dem Hardwareanteil geschrieben in VHDL ($370\mu s$). Durch eine Timing Analyse in Vivado konnte jedoch festgestellt werden, dass der Hardwareanteil mit etwa $70ns$ quasi gar nicht ins Gewicht schlägt.

In der *xor()*-Funktion des Custom Drivers wird mittels einer Schleife so lange gewartet, bis das Ergebnis des PL vorliegt. Während das Ergebnis noch nicht vorliegt, wird stattdessen „Calculating“ in der Konsole ausgegeben. Das Ergebnis liegt allerdings immer so schnell vor, dass niemals etwas in der Konsole ausgegeben wird. Dies bedeutet, dass die Hardware schneller das Ergebnis berechnet, als die Schleife ihre Bedingung evaluiert. Python ist also der Flaschenhals.

Könnte der Softwareanteil in Zukunft weiter optimiert oder sogar wegoptimiert werden, könnte die Gesamtlösung über 2300x schneller als die reine Softwarelösung und sogar über 5000x schneller als die derzeitige Gesamtlösung sein.

Selbst auf einem durchschnittlichen Laptop mit einem Intel Core i5 mit 2 GHz Taktung benötigt das in Python implementierte neuronale Netz noch gemittelte $7\mu s$. Dies ist eine deutliche Geschwindigkeitssteigerung gegenüber dem PYNQ-Z1 und seiner begrenzten Hardware, jedoch ist die Berechnung auch auf diesem Laptop noch etwa 100x langsamer als die des reinen Schaltnetzes.

Zusammenfassend kann man sagen, das nach dem derzeitigen Stand die hier entstandene Gesamtlösung nur etwa halb schnell ist, wie ein rein in Python entwickeltes neuronales Netz. Dies liegt aber ganz allein an der Softwareseitigen Implementierung der Gesamtlösung. Der Hardwareseitige Anteil ist so schnell, dass er nicht nennenswert mit ins Gewicht fällt. Da Hardware- und Softwareteil parallel zueinander laufen können, kann theoretisch sogar ein deutlich komplexeres neuronales Netz in der Hardware implementiert werden, ohne eine Steigerung der Gesamtlaufzeit zu erwarten. Denn mit einer Laufzeit von $70ns$ gibt das PL viel schneller das Ergebnis zurück, als Python es überhaupt abfragen kann.

Sowohl für die Tests als auch die Gesamtlösung wurde bewusst auf Tensorflow & Keras verzichtet, da es zum jetzigen Stand keine kompatible Version für die Architektur des auf dem PYNQ-Z1 verbauten CPUs gibt. Bei der in Python implementierten Version des neuronalen XOR-Netzes wurde jedoch darauf geachtet, es mithilfe des NumPy-Moduls so schnell wie möglich zu machen. Es wird vermutet, dass durch den Wegfall des Overheads von Tensorflow, die jetzige Implementation sogar noch schneller und optimierter ist.

7.2 Ressourcenverbrauch auf der Hardware

Autor: Ngwa Tingoh Kingsley

Um den Ressourcenverbrauch zu beschreiben, kann auf unterschiedliche Aspekte zurückgegriffen werden. Die wichtigsten Determinanten für die Bewertung des gesamten Ressourcenbedarfs sind Lookup-Tabellen, LUTRAMs, Flipflops, Digital Signal processing (DSPs) oder Globale Buffer(BUFGs). Um dabei zu aussagekräftigen Zahlen zu kommen, ist es wichtig, den jeweiligen Verbrauch nach der Synthese zu betrachten. Denn das Routing legt auch Pfade durch LUTs, sodass zusätzliche Ressourcen genutzt werden, die mit dem eigentlichen Design nichts zu tun haben. Die Tabelle 7.1 gibt die einzelnen Verbräuche sowie die noch vorhandenen Ressourcen an.

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	2692	53200	5.06
LUTRAM	60	17400	0.34
FF	1020	106400	0.96
DSP	24	220	10.91
BUFG	1	32	3.13

Abbildung 7.1: Ressourcenverbrauch auf der Hardware

Beim Vergleich der im Projekt verwendeten LUTs, LUTRAM, Flip-Flops, DSPs und BUFGs mit den im FPGA verfügbaren Designs, kann der Auslastungsfaktor des FPGA ermittelt werden.

Um relevantere Statistiken über den Ressourcenbedarf zu erhalten, wäre es notwendig, mit bestimmten Netzwerkkomponenten zu experimentieren. Eine Möglichkeit wäre die Skalierung oder Verdoppelung der Anzahl an Schichten des Xor Designs, um den Unterschied zum bisherigen Design sichtbar zu machen. Allerdings kann dies an dieser Stelle als zukünftige Arbeit betrachtet werden.

7.3 Aufwand der Implementierung

Autor: Marvin Soldin

Dieses Kapitel soll einen genaueren Überblick darüber geben wie Aufwändig eine Implementierung auf der Hardware ist. Im allgemeinen kann man sagen, dass die Umsetzung sehr stark vom momentanen Kenntnisstand in den Unterschiedlichen Themengebieten abhängt. Dabei können wir nur einen Überblick über unsere Implementierung geben, da wir bspw. keine anderen Entwicklungsmöglichkeiten in Vergleich gezogen haben.

Der größte Aufwand ist durch folgende Themen entstanden:

- Recherche zu vorhandenen Implementierungsmöglichkeiten
- Recherche zu neuronalen Netzen
- Einarbeitung in AXI, VHDL und der Entwicklungsumgebung Vivado

Die Entwicklungsarbeit an dem neuronalen Netz gestaltet sich als simplere Aufgabe, denn ein neuronales Netz kann als eine einfache verkettete Matrizenmultiplikation gesehen werden. Daraus folgt das eine einfache Lösung schon nach schneller Zeit erreicht werden kann. Wir haben eine ungefähre Implementationszeit von circa 2-3 Wochen benötigt um das neuronale Netz in VHDL umzusetzen, dabei ist zu beachten das wir lediglich 10-14 Stunden in der Woche gearbeitet haben. Außerdem muss man im Auge behalten, dass die Lösung noch sehr simpel gestaltet ist und deswegen sehr viel Platz auf der Hardware benötigt.

Unsere Lösung kann aber nun für jedes Problem angewendet werden, welches folgende Kriterien erfüllt:

- Die Topologie des neuronalen Netzes ist ein Feed-Forward-Netz
- Das Netz verbraucht nicht zu viel Platz auf dem FPGA
- Es wird die Aktivierungsfunktion Sigmoid oder ReLU verwendet

Um das Netz für andere Probleme zu verwenden, muss man lediglich die Größe der unterschiedlichen Schichten anpassen und beliebig viele Schichten miteinander verbinden. Daher kann man neue Ergebnisse schon nach kurzer Zeit erreichen, indem man folgende Schritte ausführt:

1. Das neuronale Netz in TensorFlow bzw. Keras erstellen und trainieren
2. Die Topologie auf dem FPGA nachbilden, indem man unsere Dateien verwendet
3. Eine AXI-Schnittstelle erstellen und mit dem neuronalen Netz verbinden
4. Ein Overlay erstellen und die Gewichte einprogrammieren

Der Aufwand sollte dabei sehr gering sein und nach einigen wenigen Stunden Ergebnisse liefern.

Kapitel 8

Zusammenfassung

Autor: Colin von Huth

Ein Ergebnis des Projektes ist eine entstandene Toolchain für die Hard- und Softwareentwicklung mit dem Ziel, neuronale Netze auf einem FPGA auszuführen. Hierfür wurden vorhandene, mögliche Lösungen recherchiert und evaluiert.

Die recherchierten Werkzeuge bieten interessante Ansätze, sind aber für die Umsetzung dieses Projektes nicht ausreichend dokumentiert, ausgereift oder grundsätzlich nicht geeignet. Weitere Informationen sind im Kapitel 4.6 auf Seite 28 zu finden.

Anschließend wurde recherchiert, wie neuronale Netze selbst auf FPGAs implementiert werden können. Als Ergebnis dieser Recherche ist eine geeignete Toolchain entstanden. Mit der anschließenden Umsetzung von kleineren Teilprojekten und einem Gesamtprojekt, wurde die Funktionalität dieser Toolchain erfolgreich nachgewiesen.

Ein weiteres Ergebnis des Projektes ist das erste implementierte Gesamtprojekt. Es wurde ein neuronales Netz entwickelt, welches sich wie ein XOR aus der Digitaltechnik verhält. Es wurde sich aus zwei wesentlichen Gründen für die Umsetzung dieses Projektes entschieden. Zum einen ist ein XOR wenig komplex und zum Anderen besitzt es ein nicht-lineares Verhalten. Für die hardwareseitige Implementierung wurden Tests implementiert und durchgeführt, um die grundlegende Funktionalität des Netzes nachzuweisen.

Timing Analysen haben gezeigt, dass das Ausführen neuronaler Netze auf einem FPGA definitiv eine Geschwindigkeitssteigerung um den Faktor 1000 bringen kann. Um dies aber voll ausnutzen zu können, muss die Softwareseitige Schnittstelle weiter optimiert sein, damit Ein- und Ausgaben schnell zum FPGA gesendet und vom FPGA empfangen werden können. Optimal wäre es, die Software komplett wegzuoptimieren. Erst dann kann ein neuronales Netz implementiert in einer HDL sein volles Potential ausschöpfen.

Kapitel 9

Ausblick

Autor: Marvin Soldin

Da dieses Projekt sozusagen ein Piloten-Projekt war, wurden bisher nur grundlegende Fragestellungen beantwortet. Es gibt in diesem Bereich noch eine Menge Dinge, welche Optimiert oder Ausprobiert werden können. Dieses Kapitel beschäftigt sich damit, auf weitere Fragestellungen hinzuweisen.

Der erste Punkt beschäftigt sich mit der Komplexität des Netzes, denn bisher wurde ein einziges Beispiel für die Funktion XOR entwickelt. Interessanter ist es natürlich komplexere Beispiele zu betrachten, welche dem Anwender auch wirklich einen Mehrwert bieten, denn ein XOR muss in der Regel nicht von einem neuronalen Netz berechnet werden, sondern kann auch einfach digital abgebildet werden. Interessantere Beispiele sind daher eher Dinge wie zum Beispiel die Bilderkennung oder Signalerkennung. Gerade ersteres könnte auch im Weltraum zur Auswertung von Satellitenbildern verwendet werden.

Ein weiterer Punkt wäre die Überlegung komplexere Netze abzubilden. Denn um die Daten für komplexere Netze zu transferieren muss die momentan benutzte Schnittstelle optimiert werden. Bei der XOR-Funktion mussten wir lediglich 11 Register verwenden um alle Werte zum FPGA zu transferieren, dabei wurde jedem Wert ein Register zugeteilt. Bei Bildern im Bereich von 28x28 Pixeln, wären es schon allein 784 Eingabewerte und bei Bildern im Format von 1920x1080 Pixeln schon 2.073.600 Eingabewerte. Um diese Datenmenge zu transferieren reicht die vorhandene Schnittstelle daher nicht mehr aus. Des weiteren ist zu überlegen, ob die Python Schnittstelle als weitere Optimierung ebenfalls weggelassen werden kann, da Python momentan das Nadelöhr der vorhandenen Lösung darstellt.

Durch die Überlegung komplexere Netze abzubilden, stellt sich ebenfalls die Frage ob und wie diese Datenmengen verarbeiten werden können. Die momentane Lösung optimiert lediglich auf Geschwindigkeit, indem alle möglichen Rechenschritte parallelisiert werden. Es wird aber nicht beachtet, dass sich die Matrizenmultiplikation oder -addition im grundlegenden Aufbau nicht verändert. Daher wird für jede einzelne Operation neue Hardware benötigt, sodass ein enormer Platzverbrauch entsteht, siehe 7.2 auf Seite 71. Daher wäre eine Ausnutzung der Einmaligkeit der Operationen, durch Pipelining eine mögliche Optimierungsstrategie.

Ebenfalls wurden Punkte wie beispielsweise die Optimierung der Trainingsalgorithmen zur Reduzierung der Anzahl von Neuronen noch nicht in Betracht gezogen. Auch die genutzten Datentypen können möglicherweise noch optimiert werden. Ein Beispiel dafür wäre das Framework BNN-PYNQ [Umu+17].

Dieses Projekt hat im Allgemeinen auch nur die Verwendung der Lösung im Weltraum in Betracht gezogen. Aber auch Geräte auf der Erde, wie beispielsweise die Geräte im „Internet of Things“ müssen mit geringer Rechenkapazität arbeiten, sodass auch dort eine Optimierung durch Hardware ein möglicher Lösungsansatz wäre.

Anhang A

PYNQ-Z1 Boardfiles in Vivado integrieren

Autor: Sebastian von Minden

Detaillierte Anleitung unter:

`https://pynq.readthedocs.io/en/v2.5/overlay_design_methodology/board_settings.html`

Die Board-Files vom PYNQ-Z1 zur Implementierung in Vivado lassen sich von:

`https://github.com/cathalmccabe/PYNQ-Z1_board_files/raw/master/PYNQ-Z1.zip`

herunterladen. Nach dem Download und entpacken muss der Ordner PYNQ-Z1 in den Ordner „boards“ gelegt werden. Der Ordner „boards“ ist zu finden unter:

<Xilinx installation directory>/Vivado/<Version>/data/boards

Anschließend wird noch die XDC constraints file *PYNQ-Z1C.xdc* benötigt:

`https://reference.digilentinc.com/_media/reference/programmable-logic/PYNQ-Z1/PYNQ-Z1_c.zip`

Diese muss ebenfalls entpackt und anschließend dem Projekt in Vivado hinzugefügt werden. Zu beachten ist, dass die Datei auch in den Projektordner kopiert wird.

Anhang B

Installationsanleitung für das Xilinx Vivado WebPack 2019.1

Autor: Jan Bredereke

Hier wird erklärt, wie man das Xilinx Vivado WebPack auf seinem Rechner installieren kann. Das Xilinx Vivado WebPack ist eine leistungsfähige integrierte Entwicklungsumgebung für digitale Schaltungen. Sie ist vom Hersteller in der Version „WebPack“ kostenlos erhältlich. Diese Version reicht für alle Zwecke des Studiums voll aus. Lediglich das Zeichnen von Schaltplänen ist damit nicht möglich. Hier kann man bei Bedarf auf die Vorgängersoftware Xilinx ISE WebPack zurückgreifen. Beide WebPacks sind für Linux und für MS-Windows erhältlich.

B.1 Beschaffen einer kostenlosen Lizenz

Um das Programm installieren zu können, benötigen Sie eine Lizenz. Sie sollten sich die Lizenz *vor* der Installation besorgen, um einige Komplikationen zu vermeiden (z.B. einen Browser-Aufruf als Benutzer root). Die Lizenz kostet kein Geld, aber Sie müssen Ihre persönlichen Daten angeben. Weiterhin ist bei kostenlosen Lizenzen „WebTalk“ zwangsweise aktiviert, das bei jeder Programmbenutzung bestimmte statistische Daten an den Hersteller überträgt. Wenn keine Internetverbindung besteht (auch aufgrund einer Firewall...), dann werden keine Daten übertragen, das Programm funktioniert aber trotzdem.

- Gehen Sie auf die Webseite `https://www.xilinx.com`.
- Schalten Sie ggf. JavaScript und Cookies ein, auch für die Domains `secure.xilinx.com`, `xilinx.entitlenow.com` und `coveo.com`.
- Falls statt der FontAwesome-Icons am oberen Rand nur Rechtecke mit z.B. „F007“ angezeigt werden, dann schalten Sie ggf. Javascript auch für die Domain `bootstrapcdn.com` ein.
- Klicken Sie oben auf das Person-Icon, dann auf „Sign in“, und melden Sie sich an. Falls Sie noch keinen Account bei Xilinx haben, klicken Sie auf „Create Account“.
- Falls Sie sich jetzt zum ersten Mal anmelden, erscheint erst die Seite „Create Account and Password“ und später eine Seite „My Profile“ mit den persönlichen Daten. Füllen Sie die entsprechenden Felder aus und klicken Sie auf „Save Profile“. Es erscheint eine Bestätigungsseite. Klicken Sie links oben auf das Xilinx-Logo, um auf die Startseite zurückzukehren.
- Klicken Sie oben auf „Support“, dann in „Services“ auf „Downloads & Licensing“.
- Die Downloads-Seite erscheint.
- Klicken Sie oben auf den Button „Licensing Help“.
- Klicken Sie auf einen der vielen Links zu „Xilinx Product Licensing Site“.
- Es erscheint eine Seite „Product Licensing“. Bestätigen Sie die persönlichen Daten und klicken Sie „Next“.
- Es erscheint eine weitere Seite „Product Licensing“ (auf der Domain `xilinx.entitlenow.com`), der Reiter „Create New Licences“ ist angewählt.

- Setzen Sie unten ein Häkchen bei „Vivado Design Suite: HL WebPACK 2015 and earlier License“. Die Einschränkung „2015 and earlier“ stört nicht, da sie anscheinend nicht mehr gültig ist. Die resultierende Lizenz schaltet in der Praxis auch das aktuelle Vivado WebPACK frei.
  Falls Sie nur eine 30-Tage-Evaluations-Version sehen, dann liegt das evtl. daran, dass Sie bereits innerhalb des letzten Jahres eine WebPack-Lizenz erzeugt haben. Diese frühere Lizenz gilt auch für alle neuen WebPacks, die innerhalb eines Jahres herauskommen. Sie können die frühere Lizenz also ggf. einfach weiterverwenden.
- Klicken Sie unten den Button „Generate Node-Locked License“.
- Es erscheint ein Fenster „Generate Node Licence“, dort ist u.a. bei „Node“ „any“ eingetragen. Klicken Sie „Next“.
- Es erscheint eine Seite mit „REVIEW LICENSE REQUEST“. Klicken Sie „Next“.
- Es wird ein Schlüssel berechnet, dann erscheint eine Seite: „Congratulations – Your new license file has been successfully generated and e-mailed to xxxx.xxxx@stud.hs-bremen.de. [...] Please add this sender (xilinx.notification@entitlenow.com) to your address book.“
- Schließen Sie das Fenster mit „Congratulations“.
- Klicken Sie oben am Rand das Person-Icon und dann „Sign Out“.
- Nach kurzer Zeit kommt die versprochene Email von Xilinx.

B.2 Beschaffen der Software

Offiziell unterstützt werden folgende Betriebssysteme:

- Windows 7.1: 64-bit
- Windows 10 Professional versions 1809 and 1903: 64-bit
- Red Hat Enterprise Linux 7.4-7.6: 64-bit
- CentOS Linux 7.4-7.6: 64-bit
- SUSE Enterprise Linux 12.4: 64-bit
- Ubuntu Linux 16.04.5 and 18.04.1 LTS: 64-bit – Additional library installation required

[...] Um die Installationsdatei `Xilinx_Vivado_SDK_2019.1_0524_1430.tar.gz` (21 Gigabyte) von Xilinx herunterzuladen, gehen Sie wie folgt vor:

- Gehen Sie auf die Webseite `https://www.xilinx.com`.
- Schalten Sie ggf. JavaScript und Cookies ein, auch für die Domains `secure.xilinx.com`, `xilinx.entitlenow.com` und `coveo.com`.
- Falls statt der FontAwesome-Icons am oberen Rand nur Rechtecke mit z.B. „F007“ angezeigt werden, dann schalten Sie ggf. Javascript auch für die Domain `bootstrapcdn.com` ein.
- Klicken Sie oben auf das Person-Icon, dann auf „Sign in“, und melden Sie sich an.
- Klicken Sie oben auf „Support“, dann in „Services“ auf „Downloads & Licensing“.
- Die Downloads-Seite erscheint.
- Der Reite „Vivado“ sollte bereits ausgewählt sein.
- Links werden die verfügbaren Versionen angezeigt. Die Version „2019.1“ sollte bereits ausgewählt sein.
- Zuoberst werden vorhandene Updates aufgelistet. „Update 2“ und „Update 1“ sind für uns aber **nicht richtig**, da sie lediglich Unterstützung für zusätzliche FPGAs bringen. Die Updates sind zwar große Dateien, aber keine vollständige Software.
- Nach etwas Herunterscrollen kommt man zum Abschnitt „Vivado Design Suite – HLx Editions – 2019.1 **Full Product Installation**“, zuletzt aktualisiert am 29.5.2019. Hier gibt es zur Auswahl:
  - „Vivado HLx 2019.1: WebPACK and Editions - Windows Self Extracting Web Installer (EXE – 64.62 MB) “
  - „Vivado HLx 2019.1: WebPACK and Editions - Linux Self Extracting Web Installer (BIN – 115.05 MB)“
  - „ Vivado HLx 2019.1: All OS installer Single-File Download Vivado HLx 2019.1: All OS installer Single-File Download (TAR/GZIP – 21.39 GB)“

Klicken Sie auf die von Ihnen gewünschte Variante. – Wir haben die dritte Variante genommen, um die Komplexität durch den Web-Installer zu vermeiden, und um die große Datenmenge nicht vielfach herunterladen zu müssen.
- Führen Sie ggf. das „Sign in“ noch einmal durch.
- Bestätigen Sie Ihre Adressdaten und klicken Sie auf „Download“. Wählen Sie einen Speicherort.
- Sie erhalten die Datei `Xilinx_Vivado_SDK_2019.1_0524_1430.tar.gz`.
- Prüfen Sie ggf. die heruntergeladene Datei gegen die auf der Webseite angegebene MD5-Prüfsumme oder GnuPG-Signatur.

B.3 Installation unter Linux

- Vorab müssen die Linux-Pakete `gcc` (für den Simulator von Vivado) sowie `libgl1`, `libasound2` und `libegl1` (für DocNav) installiert sein.
- Packen Sie die heruntergeladene Datei aus:
  `tar xfz Xilinx_Vivado_SDK_2019.1_0524_1430.tar.gz`
- Die folgende Installation muss nicht (und sollte nicht) als Benutzer root erfolgen. Es muss lediglich das Verzeichnis `/opt/Xilinx/` für den installierenden Benutzer schreibbar sein.
- Führen Sie den Installer aus:
  `cd Xilinx_Vivado_SDK_2019.1_0524_1430 ; ./xsetup`
- Sie müssen einige Lizenzen abnicken.
- Als zu installierende Edition müssen Sie wählen: „Vivado HL WebPack“
- Die vorgeschlagene Auswahl von Software ist bereits richtig. Nur WebTalk für den SDK sollten Sie ausschalten, um ggf. weniger Informationen über sich preiszugeben.
  Nicht nötig für uns und auch nicht vorausgewählt sind „Systemgenerator for DSP“ und „Model Composer“.
- Sie können bei der Auswahl der Software aussuchen, ob Sie Einträge für Ihr Startmenü generiert bekommen möchten. Diese landen ggf. als `*.desktop`-Dateien in Ihrem Verzeichnis `~/.local/share/applications/`. In das Verzeichnis `~/.local/share/desktop-directories/` werden entsprechend ggf. einige `*.directory`-Dateien geschrieben.
- Sofern Sie das „Xilinx Information Center“ bei der Softwareauswahl nicht abwählen, wird bei der Installation eine Autostart-Datei in Ihren Ordner `~/.config/autostart/` geschrieben werden. In der Folge wird das Programm bei jedem Anmelden am Desktop übers Internet beim Hersteller abfragen, ob Informationen über neuere Softwareversionen als die derzeit installierten vorliegen und diese Informationen ggf. anzeigen. Durch Löschen der Autostart-Datei können Sie dies evtl. störende Verhalten ggf. leicht loswerden.
- Als Installationsziel wird `/tools/Xilinx` vorgeschlagen. Sinnvoller kann z.B. `/opt/Xilinx` sein. Das gewählte Verzeichnis muss bereits existieren und schreibbar sein. Bei der Installation werden dort später drei Unterverzeichnisse angelegt, für die drei installierten Software-Komponenten.
- Der Installer legt außerdem das Unterverzeichnis `/opt/Xilinx/.xinstall/` an und kopiert einige mehrere GByte große Teile seiner selbst dort hinein. Auch Installationsprotokolle landen dort. Nach Fertigstellung aller Installationsarbeiten können Sie es löschen.
- Sie sollten Dateieigentümer und Dateirechte so setzen, dass das Verzeichnis `/opt/Xilinx/` sowie die dort installierten Dateien nur für den Benutzer root schreibbar sind. Per Default sind zumindest zwei Konfigurationsdateien für WebTalk welt-schreibbar.
- Sofern Sie ein FPGA-Board an Ihren Rechner anschließen möchten, müssen Sie, nachdem Vivado installiert ist, in Ihrem Linux-System mit root-Rechten drei Dateien installieren. Im User-Guide UG973 für Vivado 2019.1 ist dies im Abschnitt „Installing Cable Drivers“ beschrieben. Es gibt ein Installationsskript im Vivado-Baum, dass das tut. Aber eigentlich kopiert es nur drei Dateien nach `/etc/udev/rules.d/`. Diese stehen in
  `/opt/Xilinx/Vivado/2019.1/`
  `data/xicom/cable_drivers/lin64/install_script/install_drivers/`

und heißen `52-xilinx-*.rules` .

- Nach Abschluss der Installation starten Sie Vivado an und wählen unter „Help“ / „Manage License...“ den Lizenzmanager. Dort klicken Sie links „Load License“. Wählen Sie die Ihnen zugesandte Lizenzdatei im Datei-Browser aus und klicken Sie „Copy License...“. Dadurch wird die Lizenz in Vivado importiert (und unter dem Verzeichnis `~/.Xilinx/` abgelegt).
- Vivado legt für einen Benutzer das Konfigurationsverzeichnis `~/.Xilinx/` an. Per Default ist es evtl. welt-schreibbar. Das sollten Sie ggf. korrigieren.
- Sie haben nun mehrere neue Icons auf dem Desktop, sofern Sie oben ausgewählt haben, dass Sie Einträge für Ihr Startmenü generiert bekommen möchten. Wenn das verwirrend oder störend ist, entfernen Sie alle bis auf „Vivado 2019.1“.
- Um zu testen, ob Ihre Installation grundsätzlich funktioniert, können Sie das Mini-Projekt in der Datei `xilinx-vivado-installcheck.zip` verwenden, die Sie in Aulis finden. Es sollte sowohl möglich sein, das Projekt im Simulator auszuführen, als auch das Projekt zu übersetzen und einen Bitstream für ein FPGA-Board zu erzeugen. Sofern Sie die drei Kabeltreiber-Dateien (s.o.) installiert haben, sollte es auch möglich sein, ein Basys 3-FPGA-Board per USB an Ihren Rechner anzuschließen und den Bitstream auf das Board zu laden. Wie dies geht, ist in den beiden zugehörigen Abschnitten des Dokuments „Projekte in Xilinx Vivado“ zu lesen, das in Aulis steht. In den VHDL-Quelldateien steht, was das Board nach dem Laden tun sollte.

B.4 Installation unter MS-Windows

Da wir das Xilinx ISE WebPack nicht selbst unter MS-Windows installiert haben, können wir keine detaillierte Anleitung geben. Grundsätzlich muss aber einfach nur der Installer ausgeführt werden.

Das Installationsprogramm fragt unter MS-Windows evtl. zusätzlich, ob die Option WinPCap mitinstalliert werden soll. (Sie wird für die Ethernet-Hardware-Kosimulation benötigt.) Hier sollten Sie Nein wählen, um nicht irgendwelche zusätzlichen Treiber in Ihr System zu bekommen, die Sie nicht benötigen werden.

Anhang C

Anleitung zum Erstellen eines AXI IP-Cores in C

Autor: Tim Wieborg

Um einen IP-Core zu implementieren, kann das von Vivado mitgelieferte „HLS IDE“ verwendet werden. Es handelt sich hierbei um eine IDE zum entwickeln von IP-Cores in C, C++ sowie SystemC. Zuerst muss ein neues Projekt mit einer Top Funktion angelegt werden. In dem Beispiel der PYNQ-Dokumentation wurde ein Addierer implementiert. Die Top-Funktion wird also „add“ gennant. Außerdem wird eine Datei „adder.cpp“ erstellt, in der die Top-Funktion implementiert wird. Da der IP-Core speziell für den PYNQ-Z1 deacht ist, wurde bei der Initialisierung des Projekts direkt das PYNQ-Z1 Board in der „Part-Selection“ ausgewählt. In der adder.cpp wird nun die Funktion „add“ implementiert. Der Code stammt aus der Overlays Anleitung 0[PYNQ17b] und sieht die folgt aus:

```
void add(int a, int b, int& c) {
        #pragma HLS INTERFACE ap_ctrl_none port=return
        #pragma HLS INTERFACE s_axilite port=a
        #pragma HLS INTERFACE s_axilite port=b
        #pragma HLS INTERFACE s_axilite port=c

        c = a + b;
}
```

Anschließend wird das Projekt synthetisiert und als IP-Core exportiert. Um den IP-Core in einem Vivado HLx-Projekt nutzen zu können, muss das Repository, in dem der Core abgelegt wurde, noch in das Projekt eingebunden werden. Der IP-Core kann anschließend im Block Design eingefügt werden. Durch die #pgragma-Einträge wird bei dem kompilieren bereits das AXI-Protokoll mit eingebunden.

Beim kompilieren werden außerdem Adressen im RAM festgelegt, in denen die Eingabe und Ausgabewerte des IP-Cores abgelegt werden. Es handelt sich hierbei um „lokale Adressen“ innerhalb des für den IP-Core reservierten Adressbereichs. Um später auf die eigentliche Adresse zuzugreifen, muss ein entsprechender Offset hinzuaddiert werden. Dies wird durch die entsprechende write() und read() Funktion in Python allerdings implizit gemacht.

Anhang D

Repositories

Hier können alle, für das Projekt erstellte, Repositories gefunden werden.

Hinweis: Die Repositories sind auf privat gestellt, sodass nur Mitglieder darauf zugreifen können. Die relevanten Inhalte der Repositories werden entsprechend in einzelnen Unterverzeichnissen der beigelegten ZIP-Datei abgelegt.

D.1 AXI-Stream FIFO Repository

Autor: Tim Wieborg

In diesem Repository befindet sich ein Beispiel eines PYNQ Projekts, mit dem man in Python einen Datenstrom zu dem FPGA sendet und den gleichen Datenstrom wieder zurück erhält. Dies funktioniert mittels des AXI-Stream Protokolls und des Direct-Memory-Access IP-Cores von Xilinx.

- **project_src**
  In diesem Verzeichnis befindet sich das vollständige Vivado-Projekt inklusive IP-Core und Board-Files. Sollten beim Öffnen des Projekts in Vivado die Board-Files nicht gefunden werden, müssen sie manuell wieder in Vivado hinzugefügt werden. Dazu muss einfach nur das „board_files“-Verzeichnis angegeben werden.

- **pynq_src**
  Dieses Verzeichnis enthält alle Dateien, die auf den PYNQ kopiert werden müssen, wenn man das Projekt ausführen möchte. Das „axi_stream_fifo“-Verzeichnis muss auf dem PYNQ unter */home/xilinx/pynq/overlays/* kopiert werden. Das „AXI Stream FIFO“-Notebook muss nach */home/xilinx/jupyter_notebooks* kopiert werden.

D.2 AXI-Lite Adder Repository

Autor: Tim Wieborg

In diesem Repository befindet sich ein Beispiel eines PYNQ Projekts, mit dem man über Python zwei Werte an den FPGA senden kann, der der diese dann addiert und die Summe zurückgibt. Dies funktioniert mittels der Kommunikationsschnittstelle AXI-Lite.

- **project_src**
  In diesem Verzeichnis befindet sich das vollständige Vivado-Projekt inklusive IP-Core und Board-Files. Sollten beim Öffnen des Projekts in Vivado die Board-Files nicht gefunden werden, müssen sie manuell wieder in Vivado hinzugefügt werden. Dazu muss einfach nur das „board_files“-Verzeichnis angegeben werden.

- **pynq_src**
  Dieses Verzeichnis enthält alle Dateien, die auf den PYNQ kopiert werden müssen, wenn man das Projekt ausführen möchte. Das „axi_lite_adder“-Verzeichnis muss auf dem PYNQ unter */home/xilinx/pynq/overlays/* kopiert werden. Das „AXI Lite Adder“-Notebook muss nach */home/xilinx/jupyter_notebooks* kopiert werden.

D.3 VHDL Neural Network XOR

Autor: Colin von Huth

Dieses Repository beinhaltet alles, was für die Umsetzung des XOR benötigt wird. In diesem Repository befinden sich Verzeichnisse mit folgenden Inhalten:

- **vhdl_src**
  In diesem Verzeichnis befindet sich das vollständige Vivado-Projekt, mit welchem ein neuronale Netz in VHDL implementiert wurde.
  - **src**
    In diesem Unterverzeichnis befinden sich alle Quellcode- und Testbench-Dateien, die Teil des XOR-Projektes sind.
  - **additional_activation_functions**
    Dieses Unterverzeichnis enthält eine Implementierung der Sigmoid-Aktivierungsfunktion. Sie wurde im XOR-Projekt nicht verwendet.
- **pynq_integration**
  Dieses Verzeichnis enthält das vollständige Vivado-Projekt, inklusive der Implementierung des XOR-Networks als IP-Core. Die benötigten Dateien dabei sind jedoch verknüpft zum Projekt im Verzeichnis „vhdl_src“. Falls somit Änderungen an den originalen Dateien gemacht werden, muss lediglich der IP-Core neu gebaut, nicht aber die Dateien ausgetauscht werden.
- **pynq_src**
  Es enthält alle Dateien, die auf den PYNQ kopiert werden müssen, wenn man das Projekt ausführen möchte. Das „xor“-Verzeichnis muss auf dem PYNQ unter */home/xilinx/pynq/overlays/* kopiert werden. Das „XOR Network“-Notebook muss nach */home/xilinx/jupyter_notebooks* kopiert werden.
  - **xor/timing_xor_custom.p**y
    Ist das Skript für die Timing Analyse des in Python implementierten neuronalen Netzwerks XOR.
  - **xor/timing_xor_vhdl.py**
    Ist das Skript für die Timing Analyse der Gesamtlösung.
  - **xor/xor_not_custom.py**
    Ist das neuronale Netzwerk implementiert in Python.
  - **xor/xor_net_driver.py**
    Ist der Custom Driver der die *xor()*-Funktion beinhaltet, um mit der Hardware zu kommunizieren.

D.4 Tensorflow Neural Network Examples

Autor: Marvin Soldin, Colin von Huth

Dieses Repository beinhaltet in Python implementierte Beispiele für neuronale Netze.

- **double.py**
  Beispiel für die Verdopplung
- **kilometers_miles**
  Beispiel zur Umrechnung von Kilometer in Meilen
- **temperature**
  Beispiel zur Umrechnung von °C in °F
- **xor.py**
  Neuronales Netz zur Berechnung der XOR-Funktion
- **fashion_mnist**
  - **fashion_mnist.py**
    Neuronales Netz zur Identifikation von Kleidungsstücken
  - **t10k-images-idx3-ubyte.gz**
    Test Datensatz an Bildern, beziehungsweise die Eingaben für das neuronale Netz
  - **t10k-labels-idx1-ubyte.gz**
    Test Datensatz an Ausgaben für die korrespondierenden Eingaben
  - **train-images-idx3-ubyte.gz**
    Training Datensatz an Bildern, beziehungsweise die Eingaben für das neuronale Netz
  - **train-labels-idx1-ubyte.gz**
    Training Datensatz an Ausgaben für die korrespondierenden Eingaben

Abbildungsverzeichnis

Literatur

[BAL] *balenaEtcher - Home*. URL: https://www.balena.io/etcher/ (besucht am 31. 10. 2019).

[Boh] Uta Bohnebeck. „Big Data and Machine Learning“. Script KSS - Master Degree Program.

[Bre19] Jan Bredereke. „Neuronale Netze auf strahlungstoleranten FPGAs für die Raumfahrt“. Studentischen Projekt Wintersemester 2019/20. 19. Sep. 2019.

[Cho+15] Francois Chollet u. a. *Keras*. 2015. URL: https://keras.io (besucht am 16. 12. 2019).

[Coh+17] Gregory Cohen u. a. „EMNIST: Extending MNIST to handwritten letters“. In: *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2017, S. 2921–2926.

[Cou+16] Matthieu Courbariaux u. a. *Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1*. 2016. arXiv: 1602.02830 [cs.LG].

[Cro+19] Louise H. Crockett u. a. *Exploring Zynq® MPSoC*. Apr. 2019. URL: https://www.zynq-mpsoc-book.com/wp-content/uploads/2019/04/ToC_Zynq_MPSoC_ebook_web.pdf.

[DIG17] *PYNQ-Z1 Board Reference Manual*. 13. Apr. 2017. URL: https://reference.digilentinc.com/_media/reference/programmable-logic/pynq-z1/pynq-rm.pdf (besucht am 04. 12. 2019).

[DPU19] *DPU for Convolutional Neural Network v1.2*. 26. März 2019. URL: https://www.xilinx.com/support/documentation/ip_documentation/dpu/v1_2/pg338-dpu.pdf (besucht am 22. 01. 2020).

[FF] URL: https://www.mozilla.org/de/firefox/ (besucht am 15. 01. 2020).

[GHa] *GitHub*. URL: https://github.com/Xilinx/PYNQ/issues/851 (besucht am 15. 01. 2020).

[GHb] *Github*. URL: https://github.com (besucht am 15. 01. 2020).

[GIT] *Git*. URL: https://git-scm.com/ (besucht am 17. 12. 2019).

[GLa] *Slack Notifications Service*. URL: https://docs.gitlab.com/ee/user/project/integrations/slack.html (besucht am 11. 02. 2020).

[GLb] *The first single application for the entire DevOps lifecycle - GitLab | GitLab*. URL: https://gitlab.com/ (besucht am 17. 12. 2019).

[GMG16] Philipp Gysel, Mohammad Motamedi und Soheil Ghiasi. „Hardware-oriented aproximation of convulutional neural networks“. In: (2016). URL: https://arxiv.org/pdf/1604.03168.pdf (besucht am 22. 11. 2019).

[JET] *PyCharm Download*. URL: https://www.jetbrains.com/pycharm/ (besucht am 24. 11. 2019).

[KH10] Alex Krizhevsky und Geoff Hinton. „Convolutional deep belief networks on cifar-10“. In: *Unpublished manuscript* 40.7 (2010), S. 1–9.

[LEF] *LeFlow Github Repository*. URL: https://github.com/danielholanda/LeFlow (besucht am 26. 10. 2019).

[LEGa] *LegUp v4.0 Dokumentation*. URL: http://legup.eecg.utoronto.ca/docs/4.0/xilinx.html (besucht am 26. 10. 2019).

[LEGb] *LegUp: Software IDE for Programming FPGAs*. URL: https://www.legupcomputing.com (besucht am 17. 11. 2019).

[Mar+15] Martin Abadi u. a. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/ (besucht am 16. 12. 2019).

[Net+11] Yuval Netzer u. a. „Reading digits in natural images with unsupervised feature learning“. In: (2011).

[Nie15] Micheal Nielsen. *Neural Networks and Deep Learning*. 2015. URL: http://neuralnetworksanddeeplearning.com/index.html (besucht am 31. 10. 2019).

[NSW18] D. H. Noronha, B. Salehpour und S. J. E. Wilton. „LeFlow: Enabling Flexible FPGA High-Level Synthesis of Tensorflow Deep Neural Networks“. In: *ArXiv e-prints* (Juli 2018). arXiv: 1807.05317.

[OR06] Amos R. Omondi und Jagath C. Rajapakse. *FPGA Implementation of Neural Networks*. 2006.

[Per10] Perhelion. *Modell eines künstlichen Neurons*. 2010. URL: https://de.wikipedia.org/wiki/K%C3%BCnstliches_neuronales_Netz#/media/Datei:NeuronModel_deutsch.svg (besucht am 31. 10. 2019).

[Pri13] Axel Prinz. *Bildverstehen*. 2013.

[PYNQa] *Development Boards*. URL: http://pynq.io/board.html (besucht am 15. 01. 2020).

[PYNQb] *Overlay Design Methodology*. URL: https://pynq.readthedocs.io/en/v2.5/overlay_design_methodology.html (besucht am 17. 11. 2019).

[PYNQc] *PYNQ: PYTHON PRODUCTIVITY FOR ZYNQ*. URL: http://pynq.io (besucht am 15. 01. 2020).

[PYNQ17a] *Custom Driver erstellen (v2.5)*. 2017. URL: https://pynq.readthedocs.io/en/latest/overlay_design_methodology/overlay_tutorial.html#Creating-a-Driver (besucht am 05. 01. 2020).

[PYNQ17b] *IP-Core in C erstellen (v2.0)*. 2017. URL: https://pynq.readthedocs.io/en/v2.0/overlay_design_methodology/overlay_tutorial.html (besucht am 28. 11. 2019).

[PYNQ18a] *Getting Started*. 2018. URL: https://pynq.readthedocs.io/en/latest/getting_started.html (besucht am 11. 02. 2020).

[PYNQ18b] *Opening a USB Serial Terminal*. 2018. URL: https://pynq.readthedocs.io/en/latest/getting_started/terminal.html (besucht am 11. 02. 2020).

[Ras17] Taric Rashid. *Neuronale Netze selbst programmieren Ein verständlicher Einstieg mit Python*. 2017.

[SL] *Einfach zusammenarbeiten | Slack*. URL: https://slack.com/intl/de-de/ (besucht am 17. 12. 2019).

[SMH11] Ilya Sutskever, James Martens und Geoffrey E Hinton. „Generating text with recurrent neural networks“. In: *Proceedings of the 28th international conference on machine learning (ICML-11)*. 2011, S. 1017–1024.

[SZ14] Karen Simonyan und Andrew Zisserman. „Very Deep Convolutional Networks for Large-Scale Image Recognition“. In: *arXiv e-prints*, arXiv:1409.1556 (Sep. 2014), arXiv:1409.1556. arXiv: 1409.1556 [cs.CV].

[TEN] *Accelerated Linear Algebra*. URL: https://www.tensorflow.org/xla (besucht am 17. 11. 2019).

[Umu+17] Umuroglu u. a. „FINN: A Framework for Fast, Scalable Binarized Neural Network Inference“. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. ACM, 2017, S. 65–74.

[VGT17] Muhammad K Hamdan und Diane T Rover. *Vhdl generator for a high performance convolutional neural network fpga-based accelerator*. IEEE. 31. Dez. 2017. URL: https://github.com/MHAMDAN91/CNN_VHDL_GENERATOR (besucht am 22. 01. 2020).

[XIL16] *7 Series FPGAs Configurable Logic Block*. 27. Sep. 2016. URL: https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf (besucht am 20. 11. 2019).

[XIL17] *AXI Reference Guide*. 15. Juli 2017. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf (besucht am 19. 11. 2019).

[XIL18a] *7 Series FPGAs Data Sheet: Overview*. 27. Feb. 2018. URL: https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf. (besucht am 11. 11. 2019).

[XIL18b] *Accelerating DNNs with Xilinx Alveo Accelerator Cards*. 2018. URL: https://www.xilinx.com/support/documentation/white_papers/wp504-accel-dnns.pdf (besucht am 24. 11. 2019).

[XIL18c] *Zynq-7000 SoC. Technical Reference Manual*. 1. Juli 2018. URL: http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf (besucht am 31. 10. 2019).

[XIL19a] *Creating a Boot Image*. 2019. URL: https://www.xilinx.com/html_docs/xilinx2019_1/SDK_Doc/SDK_tasks/task_creatingabootimage.html (besucht am 20. 11. 2019).

[XIL19b] *Using Xilinx SDK*. 2019. URL: https://www.xilinx.com/html_docs/xilinx2019_1/SDK_Doc/index.html (besucht am 20. 11. 2019).