Sprachkommandoerkennung auf einem FPGA

Projektbericht zur Lehrveranstaltung Embedded Systems

Autoren

Mohammad Alibrahim
Mattes Bielefeld
Jan Bredereke
Serkay-Günay Celik
Moses Dimmel
Jesse Gollub
Marvin Hoyer
Thomas Klein
Friederike Korte
Simon Sajnog
Sebastian Schramm
David Schulz
Fynn Schur
Tobias Guimaraes Zimmer

Hochschule Bremen
Flughafenallee 10, 28199 Bremen

26. März 2025

HSB
Hochschule Bremen
City University of Applied Sciences

Zusammenfassung

Wir untersuchen am Beispiel einer Sprachkommandoerkennung, wie man ein neuronales Netz in einer stark leistungsbeschränkten Umgebung am besten nutzen kann, indem man ein feldprogrammierbares Gate-Array (FPGA) einsetzt. Das im vorliegenden Bericht beschriebene Projekt baut auf Projekten in früheren Semestern auf, und es wird fortgesetzt werden. Im aktuellen Semester wurden die nötigen Verarbeitungsschritte an neue Audio-Hardware angepasst sowie detaillierter identifiziert und festgelegt. Aus einem aufgetretenen Schnittstellenproblem heraus ergab sich eine neue Einsicht zu einer Systemarchitektur für eine effiziente Verarbeitung auf einem FPGA. Bei den einzelnen Teilaufgaben wurden ansonsten bedauerlicherweise keine wesentlichen Fortschritte gegenüber dem Vorgängerprojekt erzielt. Im Ausblick am Ende fassen wir den aktuellen Stand der Ideen zusammen, wie die laufende Serie von Mini-Projekten fortgesetzt werden kann.

Inhaltsverzeichnis

Kapitel 1

Einleitung

geschrieben von Jan Bredereke (außer Kap. 1.6)

Dies ist der Projektbericht zum Mini-Projekt in der Lehrveranstaltung „Embedded Systems“ an der Hochschule Bremen im Wintersemester 2024/25. Dieses Kapitel führt zunächst in den Kontext ein und präsentiert dann die Forschungsfrage. Anschließend stellt es die Beispielanwendung und einschlägige vorhergehende Arbeiten an der Hochschule Bremen vor. Das Kapitel schließt mit dem konkreten Projektziel.

1.1 Kontext

Die Motivation für das Projekt entstammt der Raumfahrt. Auf Bordrechnern ist besonders wenig Rechenleistung verfügbar, und eine Verbindung zu einer Bodenstation ist meist nur zeitweise gegeben. 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 5 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.

Insbesondere möchte man gerne neuronale Netze nutzen, die üblicherweise in Rechenzentren mit leistungstarker und stromhungriger Spezialhardware eingesetzt werden. Sie können allerdings nicht einfach am Rande der Cloud („Edge“), d.h. nahe bei den Sensoren und Aktoren, oder gar autonom von Daten- und Stromversorgungsverbindungen, auf einem Mikrocontroller eingesetzt werden, denn die CPU eines Mikrocontrollers ist dafür zu rechenschwach.

Mehr Leistung, sowohl absolut als auch pro Stromverbrauch, verspricht der Einsatz eines feldprogrammierbaren Gate-Arrays (FPGA). Grundsätzlich ist ein FPGA für die hochparallele Struktur eines neuronalen Netzes sehr gut geeignet. In der Praxis ergeben sich aber viele Optimierungsaufgaben, die erst gelöst werden müssen, bevor das FPGA diesen Vorteil voll ausspielen kann.

1.2 Forschungsfrage

Unsere Forschungsfrage ist, wie man neuronale Netze auf FPGAs am besten nutzen kann, die so leistungsbeschränkt wie strahlungsfeste FPGAs sind. Etwas allgemeiner stellt sich diese Frage entsprechend

für neuronale Netze auf FPGAs für Edge-Computing.

1.3 Die Beispielanwendung

Ein autonomes Modellauto als konkretes Vehikel (siehe Abb. 1.1) dient uns als Stellvertreter für ein autonomes Raumfahrzeug. Es ist mit einer Kamera und einem SoC Zync-7020 ausgerüstet. Das SoC enthält außer einer Arm-CPU auch ein FPGA Artix-7, dessen Leistungsfähigkeit recht gut einem weltraumtauglichen FPGA entspricht. Das SoC ist auf einem PYNQ-Z2 Board (vormals: PYNQ-Z1) verbaut, in Abb. 1.1 an der altrosa („pinken“) Farbe zu erkennen.

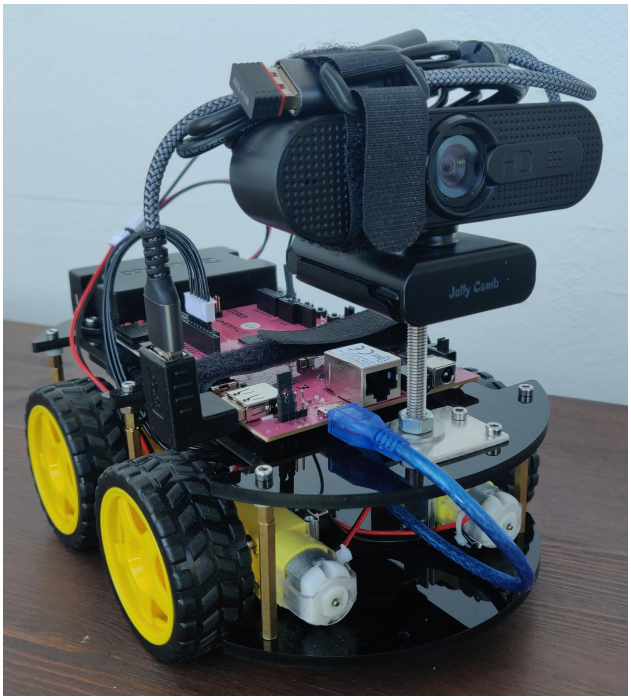


Abbildung 1.1: Das Fahrzeug aus vorhergehenden Lehrveranstaltungen mit optischer Objekterkennung auf einem FPGA. Foto: Felix Müller

1.4 Vorhergehende Arbeiten an der Hochschule Bremen

Die Lehrveranstaltung „Projekt: SoC-NN – FPGAs für neuronale Netze: Edge Computing auf autonomen Vehikeln“ im Wintersemester 2021/22 sowie einige Lehrveranstaltungen davor [Alt22; Alt21; Mül21; Mül21; Hut20] realisierten eine Bilderkennung: Das Fahrzeug soll autonom und in Echtzeit eine Person in ihrem Blickfeld erkennen, dieser Person folgen, wenn sie sich bewegt, und Fahrkommandos durch einfache Gesten gehorchen. Der Konferenzbeitrag [Bre23] fasst die Ergebnisse dieser Arbeiten zusammen.

Die Lehrveranstaltung „Embedded Systems“ im Wintersemester 2021/22 [Hag22] begann im Rahmen des zugehörigen Mini-Projekts, auf dem selben Vehikel eine Fahrsteuerung durch Sprachkommandos statt durch Gesten zu realisieren, ebenfalls autonom und in Echtzeit. Damit stiegen wir neu in den Bereich der akustischen Signale ein. In diesem Gebiet gibt es weniger fertige Lösungen für neuronale Netze als für die Bilderkennung. Dieses Mini-Projekt erstellte ein Konzept, und es produzierte Trainingsdaten, es kam aber auf Grund der beschränkten Zeit noch nicht weiter.

Die Lehrveranstaltung „Embedded Systems“ im Wintersemester 2022/23 [Bur23] begann mit der Umsetzung. Hardware-Grundlage war weiterhin das FPGA-Board PYNQ-Z1. Die Systemarchitektur wurde, aufbauend auf dem obigen Vorgängerprojekt, detailliert ausgearbeitet. Ein Teil der Audiodatenverarbeitung wurde auf dem FPGA realisiert. Die Qualität von Trainingsdaten, die aus dem Vorgängerprojekt

übernommen wurden, wurde verbessert, und es wurde ein systematisches Problem beim Aufnehmen von Trainingsdaten identifiziert und behoben. Es wurde die Möglichkeit geschaffen, einfacher verschiedene Strukturen für das neuronale Netz zu erstellen, um damit zu experimentieren. Es wurden bereits viele Schritte für die Implementierung des neuronalen Netzes auf dem FPGA ausgearbeitet; die Integration mit der Audiodatenverarbeitung konnte allerdings nicht mehr realisiert werden. Erste Zeitmessungen zeigten, dass der Einsatz eines quantisierten neuronalen Netzes die Verwendung von Fließkommazahlen bei der Inferenz überflüssig macht, welche ohne eine leistungshungrige Hardware nicht verfügbar sind. Zur Erkennungsrate für Sprachkommandos auf dem FPGA lagen bei Projektende noch keine aussagekräftigen Daten vor.

1.5 Projektziel

Während es im Projekt SoC-NN darum ging, dass das Fahrzeug einfache Arm-Gesten eines Menschen erkennen soll und damit gesteuert werden soll, sollte es nun Ziel sein, dass das Fahrzeug einfache Sprachkommandos erkennt und damit gesteuert wird.

Volle Spracherkennung ist eine komplexe Aufgabe. Sie reicht von der Analog-Digital-Wandlung über die zeitliche Fensterung über die Zerlegung in das Frequenzspektrum per schneller Fouriertransformation (FFT) über eine Filterung des resultierenden Frequenzspektrums mit nachfolgender Meta-Frequenzanalyse (Cepstrum) bis zu einer Mustererkennung mit Hilfe eines akustischen Modells, eines Aussprache-Wörterbuchs und eines Sprachmodells, um schließlich den Text in Schriftform zu erhalten. Dies setzt viel Wissen über den Aufbau von Sprache und auch viel Rechenleistung voraus.

Wir beschränkten uns daher auf das vordere Ende dieser Verarbeitungskette und begnügten uns als Ziel mit mit dem Erkennen von wenigen, einfachen Sprachkommandos für unser Fahrzeug, wie z.B. „links“, „rechts“, „gerade“ und „stopp“. Wesentliche selbst zu entwickelnde Verarbeitungsstufen sind dabei die zeitliche Fensterung, die schnelle Fouriertransformation (FFT) und die Klassifikation von Mustern in deren Ausgabe per neuronalem Netz.

Weiterhin setzen wir ab der jetzigen Arbeit ein PYNQ-Z2 Board anstelle des Vorgängermodells PYNQ-Z1 ein. Das PYNQ-Z1 hatte ein eingebautes, aber nur einfaches Elektret-Mikrofon mit sehr mäßiger Tonqualität. Der daraus resultierende Datenstrom war PDM-kodiert. Es stellte sich im Vorgängerprojekt heraus, dass dies aufwändig umzukodieren ist. Das aktuelle PYNQ-Z2 enthält weiterhin das gleiche System-on-Chip Xilinx Zynq-7020 und ist auch sonst sehr ähnlich. Aber es hat zur Audio-Eingabe nun eine Klinkenbuchse, an die ein gutes Mikrofon angeschlossen werden kann, und es enthält den 24-Bit Analog-Digital-Wandler ADAU1761. Damit wird eine Audioaufnahme in höherer Qualität möglich, was ein Problem aus dem Vorgängerprojekt beheben kann. Zu Beginn der jetzigen Arbeit war allerdings noch unklar, wie und wo genau die Audiodaten aus dem Analog-Digital-Wandler zur Verfügung stehen.

Das Projektziel soll in mehreren Zwischenschritten erreicht werden, von denen auch nur einige erste im aktuellen Semester umsetzbar sein werden. Zunächst werden wir Miniprojekt-eigene PYNQ-Z2 Boards ohne ein Fahrzeug darumherum verwenden. Wir werden die Anschlussmöglichkeit für ein gutes Mikrofon an den Analog-Digital-Wandler des Boards nutzen, um einen Audio-Datenstrom zu erhalten. Wir werden die frühen Audio-Verarbeitungsschritte im FPGA implementieren. Wir werden die praktische Erkennungsrate des neuronalen Netzes auf brauchbare Werte verbessern. Das neuronale Netz werden wir zunächst auf der CPU des SoCs realisieren. Das verschiebt die Einarbeitung in das doch recht komplexe Implementieren eines neuronalen Netzes auf einem FPGA auf später. Sobald die Verarbeitungsschritte verstanden und umgesetzt sind, optimieren wir die Verarbeitungsgeschwindigkeit, indem wir z.B. das neuronale Netz auch auf das FPGA bringen. Ebenso integrieren dann wir die Sprachkommandoerkennung in das reale Fahrzeug. Es wird nicht erwartet, dass schon im aktuellen Semester alle diese Schritte vollständig erreicht werden.

Weitere Schritte in Richtung auf ein fertiges System werden später nötig sein. Das Fahrzeug wird zunächst beim Fahren Lausch-Pausen einlegen müssen, um Sprachkommandos verstehen zu können. Möglicherweise müssen wir ein besseres Mikrofon als das auf dem Board eingebaute ergänzen, oder sogar mehrere Mikrofone für eine Richtcharakteristik, um ausreichend Reichweite zu erzielen und um mit

Störgeräuschen umgehen zu können.

Übergeordnetes Projektziel soll sein, die Leistung der Signalverarbeitung unter den vorhandenen Hardware-Randbedingungen massiv zu steigern und damit entsprechend auch die Leistung des Vehikels beim autonomen Fahren zu verbessern. Projektergebnis soll ein besseres Verständnis für die vielen Optimierungsmöglichkeiten sein, sowohl auf Seiten der Digitaltechnik als auch auf Seiten der neuronalen Netze.

1.6 Überblick über die in diesem Bericht behandelten Themen

geschrieben von Tobias Guimaraes Zimmer

Da das langfristige Projektziel, vermutlich nicht im Kontext einer einzelnen Instanz dieses Mini-Projektes zu erreichen ist, gilt es einen Überblick zu schaffen, was konkret innerhalb dieser durchgeführten Instanz aus diesem Bericht hervorgehen soll.

Übergeordnet gilt es das Gesamtprojekt, welches über mehrere Semester gehen soll, zu ermöglichen, indem die Erkenntnisse dieses Semester durch diesen Bericht und durch die Verfügbarkeit der erstellten Repositorys, geteilt werden. In dieser Instanz des Projektes wurde neben allgemeinen Erkenntnissen zur Arbeit mit dem neuen PYNQ-Z2 Board und der Ausarbeitung einer allgemeinen Architektur in den folgenden Bereichen gearbeitet:

Eine Neuerung des PYNQ-Z2 Boards ist der ADAU1761 Audio Codec Chip. In diesem Bericht werden Erkenntnisse zu dessen Verwendung und darüber hinaus eine erste Realisierung einer zeitlich gefensterten Audioaufnahme behandelt.

Im Bereich Audioverarbeitung: Wurde das Verarbeiten von Audiosignalen über die Bildung von MFCCs mittels einer konkreten (bibliothekslosen Python) Implementierung realisiert. Damit einhergehend wurde evaluiert, wie sinnhaft eine Auslagerung dieser Verarbeitung auf dem Mikrocontroller ist. Außerdem wurden erste Funktionalitäten in IP-Blöcken überführt und bereits bestehende IP-Blöcke wie der FFT-Block in Betrieb genommen und getestet.

Zur Erstellung des neuronalen Netzwerkes wurde eine Datenvorverarbeitung der bestehende Daten vorgenommen sowie der Rahmen geschaffen für das Erstellen des Netzwerkes. Konkret wurden hier zwei Trainingsansätze probiert, welche jedoch kein erfolgreiches Trainieren eines Neuronalen Netzes als Ergebnis hatten. Vermutete Ursachen hierfür sind inkonsistent verarbeitete Daten oder die verwendete Trainingshardware.

Zur Überführung des neuronalen Netzwerkes auf das FPGA, soll in diesem Bericht hervorgehen, wie eine Wandlung in einen nutzbaren IP-Block mithilfe des FINN Frameworks funktionieren kann. Hinsichtlich dessen wurde insbesondere die Inbetriebnahme des Frameworks sowie die Realisierung der Einbindung des mit FINN erstellten IP-Blockes in das Blockdesign des FPGAs behandelt.

Kapitel 2

Grundlagen

2.1 Audioanalyse und -verarbeitung

2.1.1 Hamming-Fensterung

geschrieben von Moses Dimmel

Wird ein Audiosignal in Frames unterteilt um diese gesondert zu analysieren, so ist es möglich, dass ein sonst kontinuierliches Sinussignal nicht genau mit dem Ende einer Phase abgeschnitten wird. Durch eine nicht Ganz-zahlige Periodenanzahl entsteht so ein Leck-Effekt in der Spektralanalyse [Ins]. In der Analyse tritt also nicht mehr nur die Frequenz der Sinusschwingung auf, sondern auch noch viele weitere Frequenzen welche dort nicht hingehören. Um dem entgegenzuwirken wird eine Fensterfunktion auf das Signal angewandt. Eine dieser Fenster-Funktionen ist die Hamming-Fensterung. Die Koeffizienten der Hamming-Fensterung lassen sich wie folgt berechnen [Sch]:

$$w(n) = 0{,}54 - 0{,}46 \cdot \cos\left(\frac{2 \cdot \pi \cdot n}{N-1}\right) \tag{2.1}$$

Hierdurch wird das Signal an den Enden des Fensters abgeschwächt und der Leck-Effekt wird verringert.

2.1.2 Fast Fourier Transformation (FFT)

geschrieben von Mohammad Alibrahim

Ein diskretes Signal liefert zunächst nur Informationen über seine Amplitude in Abhängigkeit von der Zeit. Allerdings sind zusätzlich Informationen über das Frequenz-Verhalten vom Signal wichtig, um eine Analyse von Tönen durchführen zu können. Von Bedeutung ist hier die diskrete Fourier Transformation (DFT). Dabei handelt es sich um eine Methodik, ein Signal vom Zeitbereich in den Frequenzbereich zu transformieren. Dadurch werden die im Signal enthaltenen Frequenzen erkennbar und es wird sichtbar, wie stark diese sind.

Die DFT wird durch folgende Formel beschrieben:

$$X[k] = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn} \tag{2.2}$$

mit:

- $X[k]$: Frequenzbereich-Signal

- x_n : Der n-te Werte vom Signal im Zeitbereich
- N : Zahl der Abtastwerte vom Signal
- k : Index vom Frequenzbereich
- n : Index vom Zeitbereich

Wie an dem Term $e^{-\frac{i2\pi}{N}kn}$ zu erkennen ist, beruht die DFT auf der Idee, dass jedes Signal mathematisch als eine Summe von Sinus- und Kosinusfunktionen ausgedruckt werden kann. Sie wird jedoch langsam berechnet, da sie rechenaufwändig ist und eine Zeitkomplexität von $O(N^2)$ hat. Eine zeiteffizientere Alternative dazu ist der FFT-Algorithmus, welcher eine optimierte und schnelle Version von DFT und für Fast-Fourier-Transformation steht. Im Gegensatz zum DFT-Algorithmus reduziert FFT den Berechnungsaufwand durch Zerlegung der DFT in kleinere Teilaufgaben (Teilen-Herrschen-Prinzip). Dadurch wird eine Zeitkomplexität von $O(N \log N)$ erreicht.

Konkret wird bei diesem Ansatz das diskrete Signal in zwei Blöcke unterteilt, von denen einer symmetrisch ungerade und der andere symmetrisch gerade Funktionen beinhaltet:

$$X[k] = \sum_{n=0}^{\frac{N}{2}-1} x_{2\cdot n} \cdot e^{-\frac{i2\pi}{N}k(2\cdot n)} + \sum_{n=0}^{\frac{N}{2}-1} x_{2\cdot n+1} \cdot e^{-\frac{i2\pi}{N}k(2\cdot n+1)} \tag{2.3}$$

Durch diese Zerlegung lässt sich die ursprüngliche DFT als Summe zweier kleinerer DFTs der Länge $N/2$ schreiben. Die Teilsummen können wiederum rekursiv weiter zerlegt werden, bis am Ende DFTs der Länge 2 erreicht sind, die sich trivial berechnen lassen. [BOU]

2.1.3 Inverse Fast Fourier Transformation (IFFT)

geschrieben von David Schulz

Die Inverse Fast Fourier Transformation (IFFT) wandelt einen Frequenzbereich zurück in den Zeitbereich um. Dabei ist diese verwandt mit der Fast Fourier Transformation, d. h. man muss um von der FFT zur IFFT zu kommen nur die Vorzeichen der Exponenten der komplexen Drehfaktoren invertieren und das Ergebnis mit dem Faktor $1/\mathrm{n}$ zu multiplizieren. Diese kann in der Praxis mittels einer Schiebeoperation umgesetzt werden, dabei muss allerdings der Faktor n der FFT eine zweier Potenz sein. [GR23]

2.1.4 Diskrete Kosinustransformation (DCT)

geschrieben von David Schulz

Die diskrete Kosiunstransformation ist ähnlich zur diskreten Fourier Transformation, da beide ein diskretes Signal vom Zeitbereich in den Frequenzbereich überführen. Der Unterschied zwischen der diskreten Kosiunstransformation und der Fourier Transformation besteht dadrin, dass die Fourier Transformation ein Signal durch Sinus- und Kosinus-Funktionen annähert, während die Kosiunstransformation das Signal nur durch Kosinus-Funktionen annähert. Die diskrete Kosiunstransformation wird vor allem bei Audio- und Videosignalen verwendet zur Dekorrelation von Daten. Es wird in der Literatur zwischen acht verschiedenen Typen der diskreten Kosinustransformationen unterschieden. Im folgenden werden die ersten vier Typen der diskreten Kosinustransformation betrachtet, die auch als DCTs bezeichnet werden. Diese können Eigenwertproblemen von reellwertigen Matrizen mit gerader Symmetrie zugeordnet werden. [Mer23, S. 199 f.]

Die orthonormalen Basisvektoren der ersten vier diskreten Kosinustransformationen werden aus der Sequenz der folgenden Terme gebildet [Mer23, S. 200 ff.].

DCT-I:

$$c_k^{\mathrm{I}}(n) = \sqrt{\frac{2}{N}}\,\gamma_k\,\gamma_n\,\cos\left(\frac{k\,n\,\pi}{N}\right), \quad \text{mit } k,n = 0,1,\ldots,N \tag{2.4}$$

DCT-II:

$$c_k^{\mathrm{II}}(n) = \sqrt{\frac{2}{N}}\,\gamma_k\,\cos\left(\frac{k\,\left(n+\frac{1}{2}\right)\,\pi}{N}\right), \quad \text{mit } k,n = 0,1,\ldots,N-1 \tag{2.5}$$

DCT-III:

$$c_k^{\mathrm{III}}(n) = \sqrt{\frac{2}{N}}\,\gamma_n\,\cos\left(\frac{\left(k+\frac{1}{2}\right)\,n\,\pi}{N}\right), \quad \text{mit } k,n = 0,1,\ldots,N-1 \tag{2.6}$$

DCT-IV:

$$c_k^{\mathrm{IV}}(n) = \sqrt{\frac{2}{N}}\,\gamma_n\,\cos\left(\frac{\left(k+\frac{1}{2}\right)\,\left(n+\frac{1}{2}\right)\,\pi}{N}\right), \quad \text{mit } k,n = 0,1,\ldots,N-1 \tag{2.7}$$

Dabei gilt für die Konstante γ_j

$$\gamma_j = \begin{cases} \frac{1}{\sqrt{2}} & \text{für } j = 0 \text{ oder } j = N \\ 1 & \text{sonst.} \end{cases} \tag{2.8}$$

2.1.5 Mel-Frequenz-Cepstrum-Koeffizienten (MFCC)

geschrieben von Mohammad Alibrahim

Mel-Frequency-Cepstral-Coefficients (kurz MFCC) bieten eine optimierte Möglichkeit zur Identifizierung von Sprache in Audiosignalen, indem sie das von FFTs erzeugte Amplitudenspektrum eines Signals in eine kompaktere Form überführen. Dies erfolgt mithilfe der sog. Mel-Filterbank. Sie hat den Zweck, die linearen Datensätze von FFT in eine nicht-lineare Repräsentation anhand der Mel-Skala mit Hilfe folgender Formel zu transformieren:

$$Mel(f) = 2595 \cdot \log_{10}(1 + \frac{f}{700}) \tag{2.9}$$

Der Sinn dieser Transformation besteht darin, dass Frequenzen in einem Bereich von niedrigen bis mittleren Tönen detaillierter dargestellt, wohingegen höhere Frequenzen komprimiert werden, was besser zur menschlichen Wahrnehmung von Audiosignalen passt. [Log]

2.2 Neuronales Netz

2.2.1 Allgemein

geschrieben von Sebastian Schramm

Neuronale Netze sind eine Art der Daten-Verarbeitung und -Bewertung, in Anlehnung an die Neuronen aus der Biologie. Sie finden in den verschiedensten Einsatzgebieten Anwendung. Beispiele lassen sich in der Objekterkennung von Bild- und Videodaten, in der Audio-Datenverarbeitung oder der Prognose und Optimierung von komplexen Systemen, wie z.B. in der Produktion, finden.

Bevor ein neuronales Netz jedoch in seinem Einsatzgebiet funktionieren kann, muss es erst einmal trainiert werden. Dafür werden Trainings- und Validierungsdaten benötigt, anhand derer eine sehr spezifische Intelligenz erarbeitet bzw. trainiert und anschließend getestet wird. Somit ist eine Verwendung des trainierten Netzes nur in seinem sehr spezifischen Bereich und auch nur in seiner spezifischen Aufgabe sinnvoll.

2.2.2 Vorhandener Datensatz

geschrieben von Friederike Korte

In diesem Projekt wird der vorliegende Datensatz des Vorgängerprojektes verwendet. Dieser besteht aus insgesamt 469 Audiodateien im mp3-Format. Die Anzahl der Audiodateien teilt sich dabei wie folgt auf die einzelnen Befehle auf:

- back - 84 Audiodateien
- forward - 74 Audiodateien
- left - 69 Audiodateien
- right - 92 Audiodateien
- stop - 75 Audiodateien
- kein valider Befehl - 75 Audiodateien

Weiterhin liegt noch eine Audiodatei mit Hintergrundgeräuschen des fahrenden Autos vor (ebenfalls aus dem Vorgängerprojekt).

Von uns hinzugefügt wurde eine leere Audiodatei (kein Ton), um die Audiodateien bei der Datenvorverarbeitung auf eine bestimmte Länge bringen zu können.

Den Datensatz findet man in unserem GitLab-Projekt unter wise2425/ki/data.

2.2.3 Training

geschrieben von Sebastian Schramm

Das Training eines neuronalen Netzes ist ein iterativer Prozess, bei dem das Modell aus den bereitgestellten Daten lernt. Dieser Prozess umfasst mehrere Schritte, die sicherstellen, dass das Modell die zugrunde liegenden Muster in den Daten erkennt und verallgemeinern kann.

Zunächst werden die Daten in Trainings- und Testdatensätze aufgeteilt. Der Trainingsdatensatz wird verwendet, um das Modell zu trainieren, während der Testdatensatz zur Validierung der Modellleistung dient. Ein spezielles `collate_fn` wird verwendet, um die Tensoren auf die gleiche Länge zu bringen und die Labels in numerische Werte umzuwandeln. Dies ist wichtig, um sicherzustellen, dass die Daten korrekt verarbeitet werden können und das Modell effizient trainiert werden kann. Weitere Details zur Aufteilung von Daten und Verarbeitungsschritten sind in [Gér19] und [Agg18] beschrieben.

Während des Trainingsprozesses wird das Modell in mehreren Epochen trainiert. In jeder Epoche werden die Eingaben durch das Modell geleitet, der Verlust berechnet und die Gewichte des Modells aktualisiert. Dies wiederholt sich über mehrere Epochen, um das Modell schrittweise zu verbessern. Nach jeder Epoche wird das Modell auf dem Testdatensatz validiert, um die Genauigkeit zu überprüfen. Dies hilft, Überanpassung zu vermeiden und sicherzustellen, dass das Modell gut generalisiert [GBC16].

Ein wichtiger Aspekt des Trainings ist die Wahl des Optimierungsalgorithmus. Häufig verwendete Algorithmen sind der Stochastic Gradient Descent (SGD) und Adam. Diese Algorithmen helfen dabei, die Gewichte des Modells so anzupassen, dass der Verlust minimiert wird. Die Lernrate ist ein weiterer

wichtiger Parameter, der die Geschwindigkeit des Lernprozesses beeinflusst. Eine zu hohe Lernrate kann dazu führen, dass das Modell nicht konvergiert, während eine zu niedrige Lernrate den Trainingsprozess unnötig verlängern kann. Eine detaillierte Erklärung der Optimierungsverfahren findet sich in [GBC16] sowie in Online-Ressourcen wie [La19].

Nach dem Training wird das Modell quantisiert, um die Effizienz zu verbessern. Die Quantisierung reduziert die Präzision der Modellparameter, was zu einer kleineren Modellgröße und schnelleren Inferenzzeiten führt. Dies ist besonders nützlich für den Einsatz auf mobilen Geräten und eingebetteten Systemen, wo Speicher- und Rechenressourcen begrenzt sind [IBM24].

Zusätzlich zur Quantisierung kann das Modell auch durch Techniken wie Pruning und Knowledge Distillation weiter optimiert werden. Pruning entfernt unwichtige Gewichte aus dem Modell, wodurch die Modellgröße weiter reduziert wird. Knowledge Distillation überträgt das Wissen eines großen Modells auf ein kleineres Modell, das dann effizienter eingesetzt werden kann [Gér19].

2.2.4 Quantisierte NN

geschrieben von Sebastian Schramm

Quantisierung ist eine Technik zur Reduzierung der numerischen Präzision in neuronalen Netzen, um Speicherverbrauch und Rechenzeit zu minimieren. Statt 32-Bit-Gleitkommazahlen (FP32) werden reduzierte Formate wie 16-Bit (FP16), 8-Bit (INT8) oder 4-Bit (INT4) verwendet.

Dabei werden die Gewichte und Aktivierungen der Netzwerkschichten auf die gewünschte Präzision abgerundet oder quantisiert. Dies führt zu Genauigkeitsverlusten, die durch Techniken wie Quantization-Aware Training (QAT) oder Hybrid-Quantisierung minimiert werden können.

Arten der Quantisierung

- **Post-Training Quantization (PTQ):** Konvertiert ein trainiertes Modell nachträglich auf eine niedrigere Präzision [IBM24].
- **Quantization-Aware Training (QAT):** Berücksichtigt Quantisierung bereits während des Trainings, um Genauigkeitsverluste zu minimieren [Gér19].
- **Hybrid-Quantisierung:** Nutzt unterschiedliche Präzisionsstufen für verschiedene Netzwerkschichten [Agg18].

Vorteile

- Reduzierter Speicherbedarf und Energieverbrauch.
- Schnellere Inferenzzeiten, besonders auf spezialisierten Hardware-Beschleunigern.
- Effizienter Einsatz auf mobilen und eingebetteten Geräten.

Herausforderungen

Genauigkeitsverluste, Hardware-Kompatibilität und Skalierungsprobleme [La19].

2.3 Eingesetzte Hardware

2.3.1 Entwicklungsboard PYNQ-Z2

geschrieben von Mohammad Alibrahim

Als Entwicklungsboard wird beim Projekt das PYNQ-Z2-Board eingesetzt. Dabei handelt es sich um ein von AMD entwickeltes Board, das als Nachfolger für das im Vorgängerprojekt eingesetzte PYNQ-Z1 Board betrachtet wird. Verbaut auf dem Board ist das System-On-Chip (SoC) Zynq-7020, welches ein **Processing System (PS)** und eine **Programmable Logic (PL)** Einheit vereint. Das PS ist im Grunde eine 2-Kern-CPU, welche mit einer Frequenz von 866 MHz taktet und ein DDR3-Arbeitsspeicher mit einer Größe von 512 MB hat. Die PL-Komponente auf der anderen Seite beinhaltet ein FPGA aus der Artix-7-Familie, dessen Leistungsfähigkeit in der Größenordnung von strahlungsfesten FPGAs ist. Ferner wird das Board mit dem PYNQ-Framework betrieben, das es ermöglicht, das FPGA einfach über Python statt über VHDL anzusprechen und zu steuern. Hierbei kommen sog. *Overlays* zum Einsatz. Diese sind bereits entworfene Bitstreams bzw. Hardware-Bibliotheken, die bestimmte Funktionen in der PL-Komponente bereitstellen und mit zusätzlicher Funktionalität erweitert werden können. Dazu zählt im Wesentlichen der Zugriff auf Protokolle sowie Peripherie-Schnittstellen (beispielsweise GPIO-Geräte, HDMI-Ein- und -Ausgabe). [AMD22c]

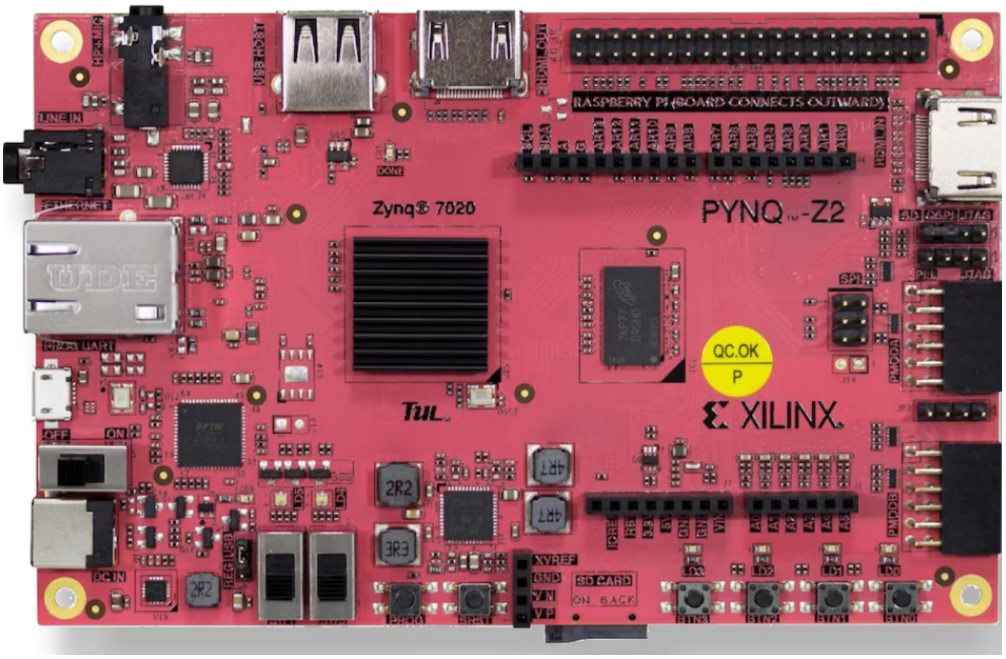


Abbildung 2.1: Bild vom Entwicklungsboard, Quelle: [AMDa]

2.3.2 AXI

geschrieben von Tobias Guimaraes Zimmer

Das „**A**dvanced e**X**tensible **I**nterface" (AXI) Protokoll [arm20] ist ein Protokoll, welches Teil der „Advanced Microcontroller Bus Architecture" (AMBA) [arm] ist. AMBA ist eine frei verfügbare Spezifikation der Firma Arm, welche im Raum der eingebetteten Systeme Anwendung findet. Insbesondere verwenden fast alle IP-Blöcke von AMD (ehemals Xilinx) dieses Protokoll.

Das Protokoll dient der Kommunikation und des Datenaustausches zwischen einer „Manager" (Master) und einer „Subordinate" (Slave) Komponente. Im Zusammenhang mit dem AXI Protokoll wird unterschieden zwischen Untertypen des Protokolls:

- **AXI4:** Die vollständige Funktionalität des Protokolls ist umgesetzt.
- **AXI4-Lite:** Eine eingeschränkte Variante des Protokolls ist umgesetzt. Hauptsächliche Einschränkung sind unter anderem eine feste Datenbreite von 32 oder 64 Bit sowie kein "bursting", siehe Abschnitt 2.3.2.
- **AXI4-Stream:** Stark eingeschränkte Variante des Protokolls, welche nur einen einseitigen Datentransfer umsetzt.

Zum Zeitpunkt dieses Berichts gibt es bereits AXI5, allerdings im Kontext des verwendeten Boards PYNQ-Z2, und der verfügbaren IP-Blöcke von AMD ist zunächst nur AXI4 von Relevanz.

Das hier und im Folgenden beschriebene kann im AXI Protokoll [arm20], im AXI-Stream Protokoll [arm21] und der „AXI Basics Series" [AMD23] (von AMD) weiter vertieft werden.

AXI4 Funktionsweise

In der Regel werden fünf Leitungen und zwei globale Signale im AXI Protokoll eingesetzt. Ausnahme ist hier der AXI4-Stream, dieser benötigt nur eine Leitung und die globalen Signale. Die globalen Signale sind `ACLK` und `ARESETn` diese sind jeweils das globale Taktsignal und ein globaler Reset. Diese gelten für alle der fünf Leitungen.

Die fünf Leitungen sind (in Klammern jeweils der Prefix den einzelne Signale dieser Leitung tragen i.e., das `VALID` Signal würde für die write address Leitung `AWVALID` heißen):

- Write address (AW): Leitung zur Steuerung einer Schreibtransaktion, hier werden Informationen zur Zieladdresse, Transaktionslänge etc. geteilt.
- Write data (W): Die eigentliche Datenleitung.
- Write response (B): Antwort Leitung des Slaves.
- Read address (AR): Leitung zur Steuerung von Lesetransaktionen prinzipiell wie die WR Leitung nur in Leserichtung.
- Read data (R): Die eigentliche Datenleitung zum lesen.

AXI Handshake Jede AXI Leitung setzt einen sogenannten Handshake ein, um den Datentransfer zu ermöglichen. Hierfür setzt jede Leitung ein `VALID` und ein `READY` Signal ein. Der Sender der Daten muss mit dem `VALID` Signal dem Empfänger signalisieren, dass die Daten bereitstehen. Der Empfänger wiederum signalisiert dem Sender mit dem `READY` Signal, dass dieser bereit ist, die Daten zu empfangen. Der Transfer geschieht dann bei der steigenden Flanke des globalen `ACLK` Signales, bei dem beide Signale auf *HIGH* sind. Dem Sender ist nicht erlaubt, das `VALID` Signal auf *LOW* zu setzen, solange kein Handshake stattgefunden hat.

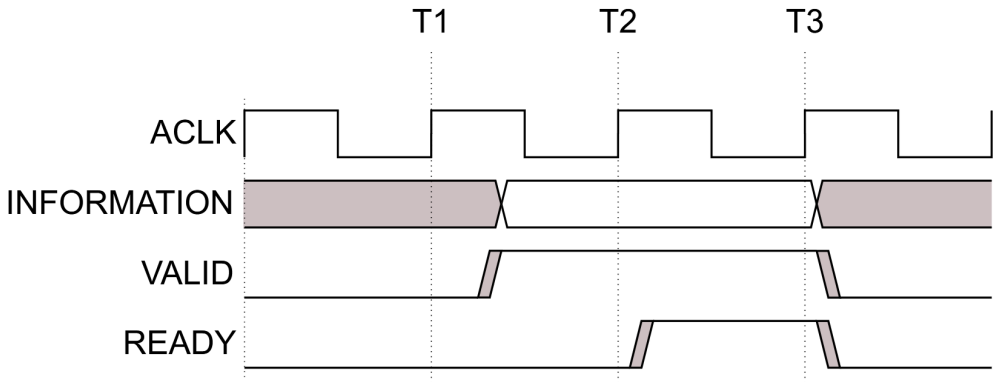


Abbildung 2.2: Beispiel AXI Handshake, Quelle: [arm20]

Aus der Abbildung geht hervor, wie ein Handshake vonstattengehen kann. Der Sender setzt seine Daten auf die Leitung und setzt `VALID` auf *HIGH*. Er ist nun verpflichtet diesen Zustand zu wahren bis der Empfänger erfolgreich die Daten erhalten konnte. Zur steigenden Flanke T2 hat der Empfänger noch nicht signalisiert, dass er bereit für den Empfang ist, dies geschieht zwischen den steigenden Flanken T2 und T3, indem dieser das `READY` Signal auf *HIGH* setzt. Zur steigenden Flanke T3 sind beide Signale entsprechend gesetzt und der Transfer kann stattfinden.

Die write data Leitung für den Datentransfer setzt zusätzlich zu dem `VALID` Signal ein `STRB` (write strobe) Signal. Dieses kann als eine Unterteilung des `VALID` Signales aufgefasst werden, es gibt Byteweise an welche Bytes valide sind. Das kann dann nützlich sein, wenn der Sender Daten senden möchte, die kleiner als die verwendete Datenbreite sind.

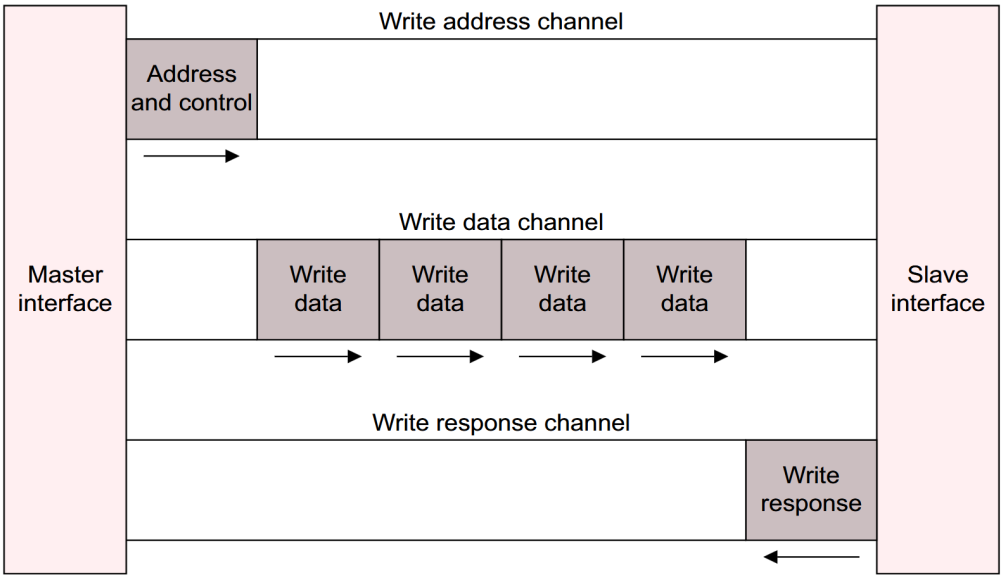


Abbildung 2.3: Beispiel AXI Schreibtransaktion, Quelle: [arm20]

AXI Lese- und Schreibtransaktionen Der generelle Ablauf einer Schreibtransaktion kann anhand der Abbildung nachvollzogen werden. Zunächst wird über die write address Leitung die Zieladresse sowie die Länge und Typ des Transfers festgelegt. Danach werden die eigentlichen Daten über die write data Leitung übertragen und anschließend über den write response channel bestätigt.

Hier ist ein Unterschied zu AXI4-Lite festzuhalten. Auf der Abbildung zu sehen ist der Transfer von vier Datenblöcken in einer Transaktion, das kann allgemein als „burst" verstanden werden. In AXI4-Lite ist dies nicht unterstützt, hier muss für jeden Datenblock eine volle Transaktion stattfinden. Aus diesem Grund besitzt die write address Leitung in der Lite Variante vier Signale, während es in der vollen Variante 13 sind.

AXI4-Streams sind prinzipiell umgesetzt durch eine write data Leitung, alle anderen Leitungen entfallen.

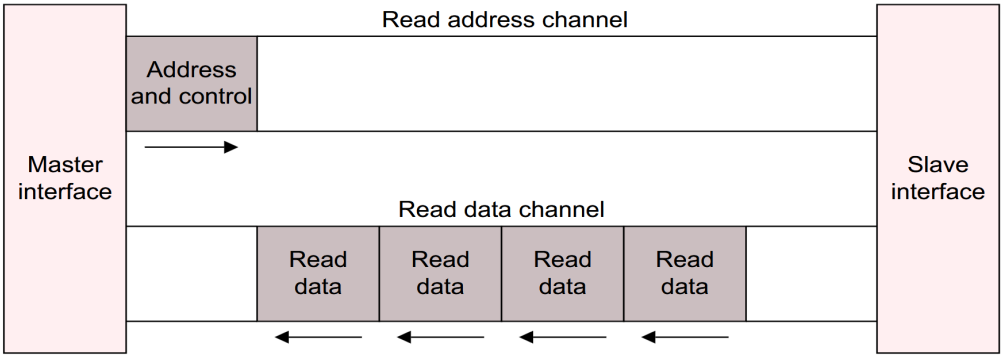


Abbildung 2.4: Beispiel AXI Lesetransaktion, Quelle: [arm20]

Lesetransaktionen sind im Grunde ähnlich zu Lesetransaktionen. Hier gilt zum Thema Bursting für AXI4-Lite das gleiche wie für Schreibtransaktionen.

2.3.3 AD Wandler

geschrieben von Mattes Bielefeld

[Ele18] lässt sich entnehmen, dass auf dem PYNQ-Z2-Board ein ADAU1761 verbaut ist. Er dient auf dem Board zur Aufnahme und Ausgabe von Audio. In [AMDa] sind eine 3,5mm Line-in Buchse und eine 3,5mm TRRS-Buchse als Audioschnittstellen des Boards angegeben. [Dev22] ist zu entnehmen, dass die Buchsen mit dem ADAU1761 verbunden sind. Daraus geht ebenfalls hervor, dass entsprechende Bus-Schnittstellen des Boards an die Ports des ADAU1761 angebunden sind. [Dev] ist zu entnehmen, dass der ADAU1761 für die digitale Kommunikation mit dem Board einen I2C/SPI-Port und einen Serial-Data-Port besitzt.

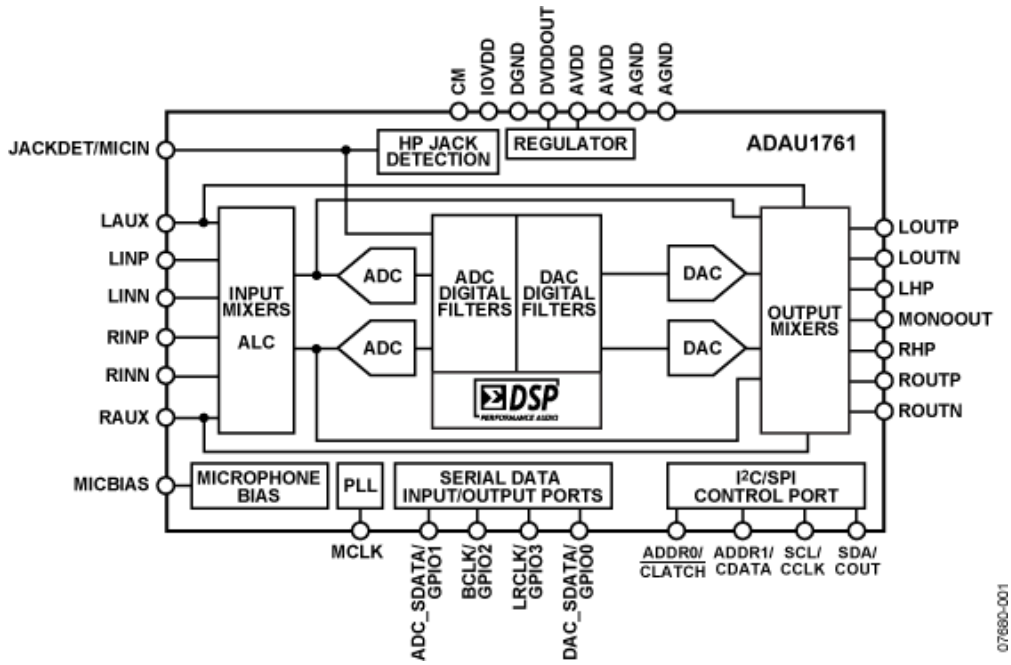


Abbildung 2.5: Verbindungen des ADAU1761, Quelle: [Dev]

In [Dev24] ist die Funktionsweise des ADAU1761 erläutert. Der ADAU1761 ist für verschiedene Zwecke konfigurierbar. Er besitzt 67 Register über die alle Funktionen des Chips gesteuert werden können. Zum Schreiben und Lesen der Register dient die I2C/SPI-Schnittstelle des ADUA1761, die an das Bussystem des Boards angeschlossen ist. Die Serial-Data-Schnittstelle dient für den Datenverkehr der Audiodaten zwischen dem ADAU1761 und dem Board.
Den Kern des ADAU1761 bilden je zwei AD- bzw. DA-Wandler. Die AD-Wandler dienen der Audioaufnahme, die DA-Wandler zum Wiedergeben von Audio. Für die Aufnahme von Audio gibt es zwei Lautstärkemixer, die vor je einen der AD-Wandler geschaltet sind. Der ADAU1761 bietet zusätzlich ALC (Automatic Level Control) für die Mixer. Bei einer Nutzung eines digitalen Mikrofons über die 3,5mm TRRS-Buchse werden jedoch die Mixer und AD-Wandler umgangen. Der ADAU1761 bietet außerdem einen Audio-Beipass, um aufgenommenes Audio ohne Verarbeitung direkt abzuspielen. Dabei wird das Signal von den Input-Mixern direkt zu den Output-Mixern übertragen, wodurch jegliche AD-Konvertierung umgangen wird. Gleichermaßen wie für die Audioaufnahme sind bei der Wiedergabe Lautstärkemixer hinter die DA-Wandler geschaltet. Zudem bietet der ADAU1761 über die 3,5mm TRRS-Buchse Mono- und Stereo-Audiowiedergabe.
Die Abtastrate der AD-Wandler kann zwischen 8KHz und 96KHz eingestellt werden, standardmäßig ist sie bei 48KHz. Das digitale Audiosignal wird in 4-Byte-Blöcken intern weitergegeben, wobei die obersten 8 Bit lediglich Puffer ohne tatsächlichen Inhalt sind.

2.4 Eingesetzte Software

2.4.1 Python

geschrieben von Serkay-Günay Celik

Python ist eine einfach zu erlernende, vielseitige und leistungsfähige Programmiersprache, die sowohl für Anfänger als auch für erfahrene Entwickler geeignet ist. Sie wurde 1991 von Guido van Rossum entwickelt und ist der Nachfolger der ABC-Programmiersprache. Die einfache und klare Syntax von Python erleichtert das Schreiben, Verstehen und Debuggen von Code [MV24].

Ein herausragendes Merkmal von Python ist seine Dynamik. Es handelt sich um eine interpretierte Sprache, was bedeutet, dass der Code nicht vorab kompiliert werden muss. Dadurch wird der Entwicklungsprozess beschleunigt, insbesondere bei Tests und Debugging. Allerdings läuft Python in der Regel langsamer als kompilierte Sprachen wie C++ oder Java [MV24].

Python bietet eine Vielzahl an eingebauten Datenstrukturen wie Listen, Dictionaries, Sets und Tupel, die eine effiziente Datenverwaltung ermöglichen. Im Gegensatz zu Sprachen wie C oder Java müssen Variablen in Python nicht explizit deklariert werden, da es um eine dynamische typisierte Sprache handelt. Dies bedeutet, dass der Datentyp einer Variablen während der Laufzeit geändert werden kann [MV24].

Ein weiteres wichtiges Merkmal ist die Unterstützung verschiedener Programmierparadigmen, einschließlich prozeduraler, objektorientierter und funktionaler Programmierung. Die Code-Struktur basiert auf Einrückungen anstelle von geschweiften Klammern, was zur besseren Lesbarkeit beiträgt [MV24].

Python ist plattformunabhängig und läuft auf verschiedenen Betriebssystemen wie Windows, macOS, Linux und Raspberry Pi. Aufgrund seiner einfachen Handhabung und der großen Anzahl an Bibliotheken ist Python eine der führenden Sprachen in Bereichen wie Data Science, künstliche Intelligenz, Webentwicklung und Automatisierung [MV24].

2.4.2 PyDub und ffmpeg

geschrieben von Friederike Korte

Bei PyDub handelt es sich um eine Python-Bibliothek, mit der verschiedene Formate von Audiodateien (mp3, wav ...) verarbeitet werden können. Sie bietet unter anderem Funktionen zum Importieren, Exportieren und Schneiden sowie Verketten und Überlagern von mehreren Audiodateien. [Sid23]

FFmpeg (Fast Forward MPEG) ist ein Programm, welches zur Konvertierung und Transkodierung von Mediendateien eingesetzt wird. [Cre] FFmpeg wird benötigt, weil PyDup bei vielen seiner Operationen auf dieses Programm zurückgreift.

2.4.3 PyTorch

geschrieben von Serkay-Günay Celik

PyTorch ist ein leistungsfähiges Framework für maschinelles Lernen und Deep Learning, das insbesondere für die Verarbeitung großer Datenmengen und komplexer Berechnungen auf GPUs optimiert wurde. Entwickelt von der Forschungsabteilung für künstliche Intelligenz von Facebook, bietet PyTorch eine flexible und dynamische Umgebung für die Modellierung neuronaler Netze. Im Gegensatz zu statischen Frameworks ermöglicht PyTorch eine intuitive und interaktive Modellentwicklung, wodurch Anpassungen während der Laufzeit vorgenommen werden können [Mis23].

Ein zentrales Konzept in PyTorch ist die Verwendung von Tensors, einer optimierten Datenstruktur, die sich ähnlich wie NumPy-Arrays verhält, jedoch für GPU-Beschleunigung optimiert ist. Die Tensor-Operationen von PyTorch unterstützen numerische Berechnungen auf mehreren Plattformen, einschließlich CPUs und GPUs, was die Leistung beim Training neuronaler Netzwerke erheblich verbessert [Mis23].

Darüber hinaus bietet PyTorch ein Autograd-Modul, das die automatische Berechnung von Gradienten ermöglicht, was für die Optimierung von neuronalen Netzwerken entscheidend ist. Mit der integrierten torch.nn-Bibliothek stellt PyTorch eine Vielzahl an vordefinierten Schichten und Funktionen bereit, die die Implementierung von Modellen wie Convolutional Neural Networks (CNNs) oder Recurrent Neural Networks (RNNs) erleichtern [Mis23].

Dank dieser Eigenschaften eignet sich PyTorch hervorragend für experimentelle Forschung, schnelle Prototypenentwicklung und den produktiven Einsatz von Deep-Learning-Modellen.

2.4.4 Brevitas

geschrieben von Serkay-Günay Celik

Brevitas ist eine auf PyTorch basierende Bibliothek, die speziell für die Entwicklung von quantisierten neuronalen Netzwerken entwickelt wurde. Sie ermöglicht die Anwendung von Quantization Aware Training (QAT), wodurch Modelle mit reduzierter numerischer Präzision trainiert werden können. Dies ist besonders wichtig für hardwareeffiziente Implementierungen auf Field Programmable Gate Arrays (FPGAs), da eine reduzierte Präzision zu geringeren Speicher- und Rechenanforderungen führt [Her24].

Ein zentraler Vorteil von Brevitas ist die nahtlose Integration mit PyTorch, wodurch bestehende neuronale Netzwerkarchitekturen einfach um Quantisierung erweitert werden können. Die Bibliothek bietet Unterstützung für verschiedene Quantisierungsmethoden und ermöglicht es, sowohl Gewichte als auch Aktivierungen in einem neuronalen Netz mit geringer Präzision darzustellen. Dies trägt dazu bei, Modelle für den Einsatz in ressourcenbeschränkten Umgebungen zu optimieren [Her24].

2.4.5 Docker Engine

geschrieben von Marvin Hoyer

Die Docker Engine ist für die Verwendung des im folgenden Abschnitts beschriebenen FINN-Compilers notwendig. Bei dieser Engine handelt es sich nach [Doca] um eine Open-Source-Technologie zur Isolierung von Programmanwendungen mittels virtueller Container. Diese Container enthalten alle notwendigen Dateien und Ressourcen und erlauben somit eine replizierbare Ausführung einer Anwendung auf unterschiedlichen Rechnern.

2.4.6 FINN Projekt

geschrieben von Marvin Hoyer

Das FINN Projekt bezeichnet ein experimentelles Framework des Integrated Communications and AI Lab der Firma AMD Research & Advanced Development. Dieses Framework bietet einen End-to-end-Workflow zur Untersuchung und Implementierung quantisierter neuronaler Netze auf FPGA-Boards [siehe dazu FIN24].

Zum FINN-Projekt gehören neben dem FINN Framework auch das bereits erwähnte Brevitas und der FINN Compiler [FIN24]. Im FINN-Projekt findet Brevitas Anwendung zur Umsetzung quantisierter neuronaler Netze. Nach der Umsetzung der Quantisierung mittels Brevitas kann dieses damit im „Open Neural Network Exchange“-Format (ONNX) exportiert werden. Dieses Format dient laut [ONN] dem Austausch von neuronalen Netzen.

Im nächsten Schritt kann über den FINN-Compiler das Netzwerk im ONNX-Format eingelesen und kompiliert werden. Dabei werden nach [FINc] die quantisierten neuronalen Netze unter Verwendung von Vivado-HLS und verschiedenen RTL-Modulen auf die Datenflussarchitekturen des verwendeten FPGA-Boards abgebildet und ein Bitfile generiert. Dieses Bitfile beschreibt die Umsetzung des neuronalen Netzes auf dem FPGA und enthält zusätzlich weiteren Code für das verwendete PYNQ-Board. Wie in

[FINb] beschrieben können durch die modulare Architektur von FINN zudem auch Zwischenausgaben wie IP-Blöcke ausgegeben und gespeichert werden.

2.4.7 Jupyter Notebook

geschrieben von Mohammad Alibrahim

Jupyter Notebook ist eine webbasierte Entwicklungsumgebung, die vor allem im Umfeld von Datenanalyse und dem Trainieren von Machine-Learning-Modellen beliebt ist. Sie erlaubt es, Quellcode in getrennten Zellen zu schreiben, sodass sich Quellcode-Blöcke individuell durchführen lassen. Zudem unterstützt Jupyter Notebook die Dokumentation mit Markdown, wodurch Texte und Überschriften leicht eingefügt werden können. [Tea]

2.4.8 Vitis HLS

geschrieben von Moses Dimmel

„Vitis HLS“ ist eine Entwicklungsumgebung, welche mittels High-Level Synthesis (HLS) die Entwicklung von IP-Blöcken ermöglicht, welche anschließend in Vivado importiert werden können. Dabei wird kein VHDL geschrieben, sondern die Entwicklung findet mittels in einer höheren Sprache, namentlich „C++“ statt. Der „C++“-Code wird dann von Vitis nach HDL (Verilog oder VHDL) kompiliert, wobei einige automatische Optimierungsverfahren angewandt werden um Speicher- und Rechenzeit zu sparen. Der Entwicklungsvorgang sieht wie folgt aus:

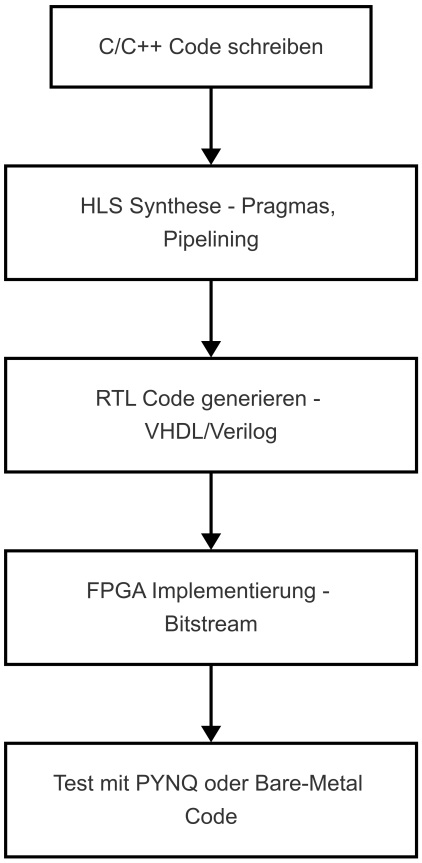


Abbildung 2.6: Entwicklungsabfolge bei der Implementierung eines IP-Blocks mittels HLS

Im Anschluss kann der erstellte IP-Block in ein Blockdiagramm in Vivado eingebunden werden.

Kapitel 3

Geplante Architektur

3.1 Geplante Architektur

geschrieben von Tobias Guimaraes Zimmer

Das PYNQ-Z2 Board stellt einen Mikrocontroller und ein FPGA Board zur Verfügung, daher müssen in einer Architektur beide dieser Seiten berücksichtigt werden. Die Mikrocontroller Seite, beschrieben als „Processing System" (PS), soll hier aber nur auf minimale weise genutzt werden, denn ein langfristiges Ziel wäre es, die gesamte Architektur auf einem FPGA Board umzusetzen. Während diese spezifisch gezeigte geplante Architektur mit dem Hintergrund des PYNQ-Z2 entstanden ist, so wurde dabei auch bedacht sie so allgemein zu halten, dass es auf vergleichsweisen Boards umsetzbar wäre.

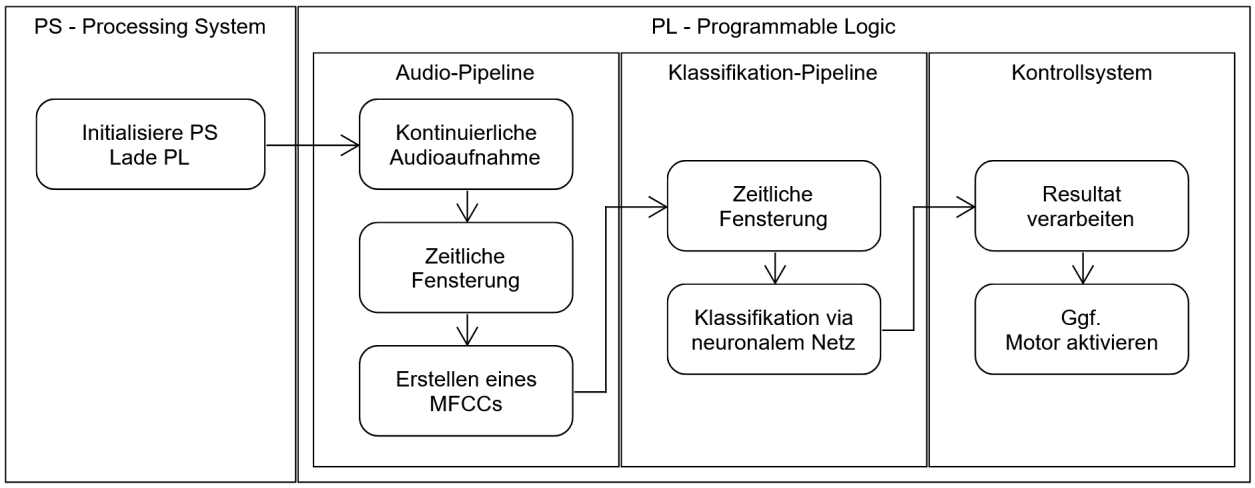


Abbildung 3.1: Geplante Architektur des Gesamtsystems

Die minimale Nutzung der PS Seite wird in der geplanten Architektur deutlich, hier ist die einzige Aufgabe der PS Seite das FPGA zu programmieren. Die FPGA Seite des Boards, beschrieben als „Programmable Logic" (PL), soll idealerweise die gesamte Logik des Programmes beinhalten, welche in drei Teile geteilt werden kann.

Audio-Pipeline: Die Audio-Pipeline soll kontinuierlich Audio aufnehmen und in gewissen zeitlichen Intervallen an die MFCC (Mel Frequency Cepstral Coefficients) Erstellung weitergeben, welche dann eins oder mehrere MFCCs erstellt und der Klassifikation-Pipeline zur Verfügung stellt. Die Audioaufnahme soll mittels des auf dem PNYQ-Z2 Board verbauten ADAU1761 [AMD22b] geschehen, denkbar ist eine Umsetzung allerdings auch mit anderen Boards und Hardware. Die Kontinuität der Aufnahme kann auf verschiedene Art und Weise umgesetzt werden, so könnte z.B. eine Art Schieberegister eingesetzt werden. Sind die zeitlichen Rahmenbedingungen der anderen Teile der Pipeline bekannt, so ist dies

allerdings nicht mehr nötig. Bei einer Fensterung von 20 ms reicht es dann aus, wenn in diesem Intervall der allokierte Buffer (z.B. durch Block RAM auf dem FPGA Board), die neuen Audiodaten beinhaltet. Sollte der Block zum Erstellen der MFCCs länger als eine akzeptable Fensterlänge (i.d.R. 20-40 ms) benötigen, so ist es empfehlenswert, die zeitliche Fensterung zu verdoppeln und in diesem Block eine weitere Fensterung zu realisieren.

Klassifikation-Pipeline: Die Klassifkation-Pipeline besteht aus den wesentlichen Teilen der Fensterung und der eigentlichen Klassifikation. Da die Audio-Pipeline MFCCs zu ihrer entsprechenden Fensterung bereitstellt (i.d.R. 20-40 ms) benötigt es einer Systematik diese sinnhaft (z.B. bis ein Bereich von zwei Sekunden abgedeckt ist) zu sammeln und dann durch ein neuronales Netz klassifizieren zu lassen. Auch hier kann entsprechend der Dauer des neuronalen Netzes die Fensterung angepasst werden, um so oft wie möglich eine Klassifizierung durchzuführen. I.e., dauert die Klassifizierung 200 ms so kann das zwei Sekunden Fenster alle 200 ms geschoben werden.

Kontrollsystem: Das Kontrollsystem hat zur Aufgabe, das Resultat der Klassifikation zu bewerten und auszuführen. Konkret ist das System dafür verantwortlich, die Motoren entsprechend erkannter Sprachkommandos anzusprechen. Daher besteht es aus zwei Teilbereichen, die Verarbeitung des Resultats und die eigentliche Aktivierung des Motors. Dies kann je nachdem, wie das Resultat des neuronalen Netzes aussieht auch in einem Teil durchgeführt werden, es ist allerdings denkbar, dass eine gewisse Vorverarbeitung nötig sein wird und eine eins zu eins Umsetzung an die Motoren eher schwierig sein sollte.

Die Kommunikation zwischen den Pipelines und auch zwischen den einzelnen Komponenten ist grundsätzlich über mehrere Methoden denkbar. Für das PYNQ-Z2 Board wird dies entweder durch eine Verbindung mittels AXI (z.B. AXI4-Streams) geschehen oder durch geteilte Speicherbereiche.

Hier ist anzumerken, dass es durchaus sein kann, dass die zur Verfügung stehenden Komponenten auf dem FPGA Board nicht ausreichend sind, um diese Architektur in ihrer Gesamtheit zu realisieren. Sollte dies der Fall sein, dürfte es kein Problem darstellen, Teile dieser Architektur auf die PS Seite zu verlagern. So kann zum Beispiel das Kontrollsystem oder die Erstellung eines MFCCs auch auf der PS Seite stattfinden. Hierbei ist zu vermeiden, die Klassifikation auszulagern, da dies nicht dem Gesamtziel des Projekts entsprechen würde. Außerdem ist es fraglich, die kontinuierliche Audioaufnahme auszulagern, da diese, zumindest auf dem PYNQ-Z2 Board, direkt mit der PL Seite verbunden ist. Ein Weg, die Verbindung zwischen PS und PL Seite herzustellen für das PYNQ-Z2 ist mittels „AXI Direct Memory Access" [AMD24e] weiteres hierzu kann im Kapitel 4.2 gefunden werden.

Kapitel 4

Arbeit mit Board

4.1 PYNQ Inbetriebnahme

geschrieben von Jesse Gollub

4.1.1 Vorraussetzungen

Für die Inbetriebnahme des PYNQ-Z2 werden folgende Komponenten benötigt:

- PYNQ-Z2 Board
- Netzteil
- MicroSD-Karte (mindestens 8 GB empfohlen)
- Computer mit MicroSD-Kartenleser

4.1.2 Betriebssystem auf SD-Karte schreiben

Das neuste PYNQ-Image kann von der offiziellen Webseite heruntergeladen werden:
`https://www.pynq.io/boards.html`
Zur Installation auf der MicroSD-Karte unter Linux sind folgende Schritte zu gehen:

1. Image entpacken

   ```
   $ unzip pynq_z2_v3.0.1.zip
   ```

2. SD-Karte anschließen und das entsprechende Gerät ermitteln

   Zur Ermittlung, unter welcher Adresse die SD-Karte erreichbar ist, kann der folgende Befehl verwendet werden:

   ```
   $ sudo blkid
   ```

   Dies listet alle angeschlossenen Block-Geräte auf (Dateisysteme etc.). Im weiteren Verlauf wird davon ausgegangen, dass die SD-Karte an `/dev/sda` angeschlossen ist.

3. Image auf die SD-Karte schreiben:

   ```
   $ sudo dd if=/path/to/pynq_z2_v3.0.1.img of=/dev/sda
   status=progress bs=1M
   ```

Hier wird mithilfe des `dd`-Befehls (diskdump) das Image auf das Gerät `/dev/sda` mit einer Blockgröße von 1 Megabyte pro Lese/Schreib-Operation geschrieben.

4. Nach Abschluss die SD-Karte sicher entfernen und in das PYNQ-Board einsetzen.

Um das Board zu starten, muss der *Boot*-Jumper auf *SD*-Position gesetzt sein, das Netzteil verbunden und der dazugehörige *Power*-Jumper auf *REG* gesetzt sein. Das Ethernetkabel muss wie im nächsten Abschnitt erklärt verbunden werden. Danach kann das Board gestartet werden.

Nach kurzer Zeit nach dem Anschalten des Boards sollten die LEDs auf dem Board anfangen, grün zu leuchten. Das bedeutet, dass das Board erfolgreich hochgefahren ist. Weitere Informationen zur Inbetriebnahme des Boards sind auf dem Wiki der Python-Bibliothek *PYNQ* zu finden (siehe [Dev22]).

4.1.3 Netzwerkverbindung herstellen

Nach dem Hochfahren kann das Board mit dem Computer verbunden werden.

Verbindung im Heimnetz

Im privaten Heimnetz kann das Board einfach per Ethernet-Kabel in das gleiche Netzwerk wie der eigene Computer angeschlossen werden. Um die vergebene Adresse herauszufinden kann beispielsweise `nmap` oder das Menü des Routers verwendet werden.

Verbindung in der Hochschule

Eine direkte Verbindung zwischen Computer und ist nur dann verlässlich möglich, wenn der Computer über einen Ethernetanschluss verfügt. Ist dies der Fall, muss die Netzwerkkarte des Computers auf den gleichen Adressbereich wie das Board gesetzt werden. Wenn das Board zuvor nicht mit einem Heimnetz eines Studenten verbunden war, hat das Board die Adresse `192.168.2.99` mit der Subnetzmaske `255.255.255.0`, weshalb die Netzwerkkarte des Laptops auf eine Adresse wie `192.168.2.XXX`, z.b. `192.168.2.2`, gesetzt werden muss.

Falls der Computer keinen Ethernetanschluss hat, kann ein USB-Ethernet-Adapter genutzt werden. Allerdings kann eine direkte Verbindung mit dem Board zu hoher Paketverlust-Rate führen. Ein zwischengeschalteter DHCP-Server kann helfen, die Netzwerkperformance deutlich zu verbessern.

Überprüfung der Verbindung

Nach dem Hochfahren und Konfiguration des Netzwerks kann die Verbindung mit `ping` getestet werden:

```
$ ping 192.168.2.99
```

Falls das Board antwortet, kann es über SSH oder Jupyter angesprochen werden.

4.1.4 Arbeit mit dem Board über SSH

SSH-Zugang zum Board

Das PYNQ-Board kann über SSH mit folgendem Befehl angesprochen werden:

```
$ ssh xilinx@192.168.2.99
```

Nutzung der pynq-Bibliothek in Python

Für die Nutzung von Overlays muss Python mit der pynq-Bibliothek verwendet werden. Da für den Zugriff auf die Hardware eine privilegierte Sitzung notwendig ist, sollte der Root-User verwendet werden. Standardmäßig hat dieser jedoch keinen Zugriff auf die pynq-Bibliothek. Diese kann durch die Aktivierung der virtuellen Python-Umgebung von pynq geladen werden:

```
$ source /etc/profile.d/pynq_venv.sh
```

Danach können alle Python-Skripte mit pynq-Abhängigkeiten normal ausgeführt werden.

4.1.5 Arbeit mit dem Board mit Jupyter Notebooks

Zugriff auf Jupyter Notebook

Nachdem das Board erfolgreich mit dem Netzwerk verbunden ist, kann auf Jupyter Notebook zugegriffen werden. Dazu kann beispielsweise ein Webbrowser geöffnet werden und auf den Webserver, der auf dem Board läuft zugegriffen werden, indem die IP-Adresse des Boards eingegeben wird:

```
http://192.168.2.99:9090
```

Auf dem Port 9090 ist der Webserver von Jupyter erreichbar.

Die Anmeldedaten sind standardmäßig xilinx für den Benutzernamen und ebenfalls xilinx für das Passwort.

Die Weboberfläche bietet Nutzern Zugriff auf die Notebooks, welche im folgenden Verzeichnis liegen:

```
/home/xilinx/jupyter_notebooks
```

In einem Notebook kann dann beispielsweise ein Overlay, welches zuvor mit Vivado erstellt wurde, geladen werden. Dazu müssen die `.bit` und `.hwh` Dateien in Jupyter hochgeladen werden, um in Python darauf zugreifen zu können:

```
1 from pynq import Overlay
2 overlay = Overlay("base.bit")
```

Hier wird ein Overlay aus den Dateien `base.bit` und `base.hwh` geladen.

Wenn das Board sicher heruntergefahren werden soll, kann in Jupyter ein Terminal gestartet werden und von dort mit folgendem Befehl heruntergefahren werden:

```
$ shutdown now
```

4.2 Arbeit mit Vivado

geschrieben von Tobias Guimaraes Zimmer

Diese Sektion soll zeigen, wie eine mittels Vivado erstellte Logik für das FPGA, auf dem PYNQ-Z2 ausführbar gemacht werden kann. Sowie wie ein exemplarischer IP-Block auf der Mikrocontroller Seite zur Verfügung gestellt werden kann. Für diese Sektion wurde die *Vivado Design Suite 2024.2* genutzt, es kann aber auch in älteren Versionen durchgeführt werden. Wenn eine Version vor 2022 genutzt wird, so müssen alle *AXI SmartConnects* [AMD22d] ersetzt werden durch *AXI Interconnects* [AMD22a], diese verhalten sich äquivalent zueinander und der SmartConnect ist als eins zu eins Ersatz für den Interconnect gedacht.

Vivado ist ein relativ umfangreiches Produkt und bietet an verschiedenen Stellen an, Blöcke selbst zu verbinden. Diese Funktion kann genutzt werden, ist aber mit Vorsicht zu genießen, da an manchen Stellen falsche Verbindung und oder auch falsche oder unsinnige Blöcke generiert werden.

4.2.1 Projekterstellung

Die Projekterstellung kann im weitesten Sinne durch dem Folgen der Anweisung der Vivado Oberfläche vonstattengehen. Bei der Auswahl des Projekttyps wird *RTL Project* ausgewählt. Wichtig ist im letzten Schritt das korrekte Board auszuwählen:

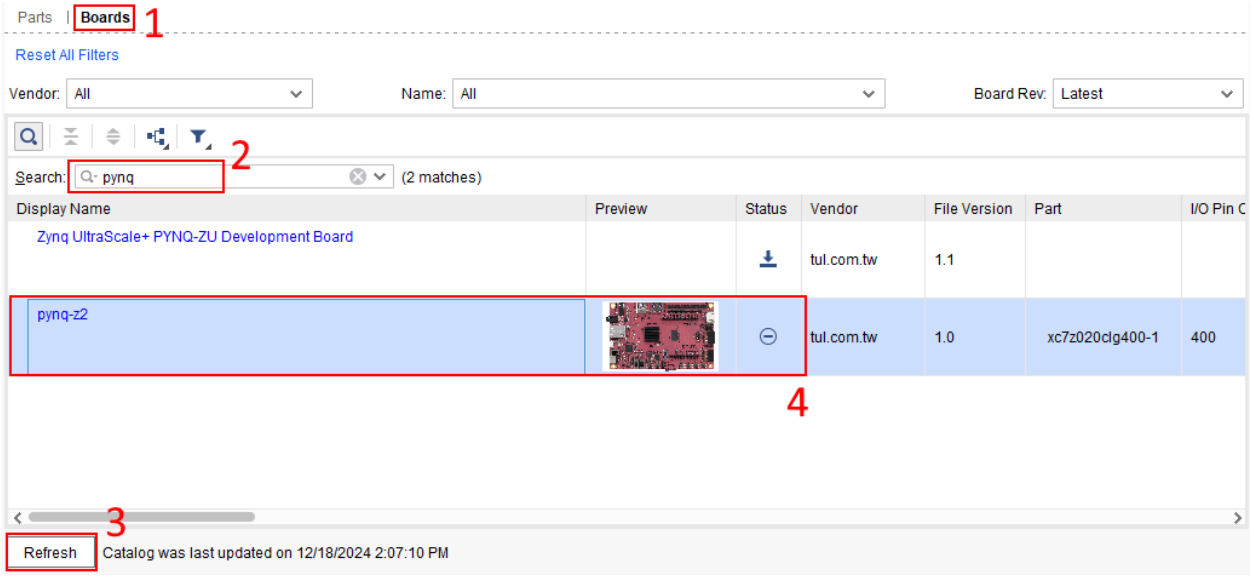


Abbildung 4.1: Board Auswahl in Vivado

Hier muss zu dem Reiter *Boards* (1) navigiert werden, dann kann nach dem Board *pynq* (2) gesucht werden. Sollte es nicht auftauchen in der Liste wie in der Abbildung zu sehen, muss der Katalog aktualisiert werden durch Betätigen des *Refresh* (3). Das Board sollte nun wie gezeigt (4) auswählbar sein. Sollte die Aktualisierung des Katalogs nicht erfolgreich sein, gibt es auch die Möglichkeit das Board manuell hinzuzufügen. An dieser Stelle kann in den Projekteinstellungen die *Target language* auf *VHDL* gestellt werden.

4.2.2 Erstellen eines Blockdesigns für das PYNQ-Z2

In dem erstellten Projekt kann nun ein Blockdesign erstellt werden. Bevor IP-Blöcke zu diesem Design addiert und verbunden werden können, ist es empfehlenswert, die verfügbaren Schnittstellen zu kennen:

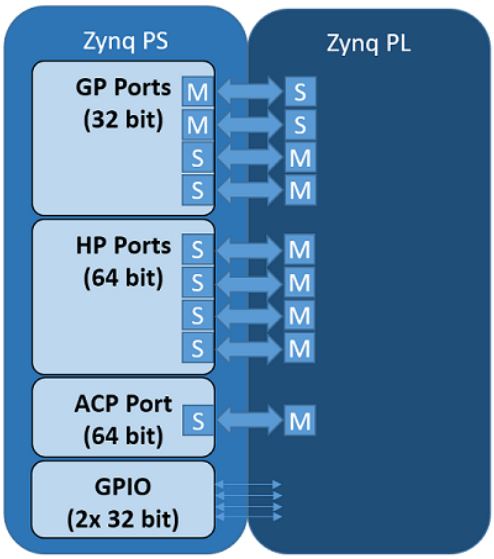


Abbildung 4.2: PS-PL Schnittstellen PYNQ-Z2. Quelle: [AMD22b]

In der Abbildung werden die möglichen Schnittstellen des PYNQ-Z2 gezeigt, relevant sind hier vor allen die *General Purpose* (GP) und die *High Performance* (HP) Ports. Wenn beispielsweise eine Interaktion von der PS Seite (i.e., aus dem Jupyter Notebook) mit der PL Seite stattfinden soll, muss einer der GP Master Ports genutzt werden. In die andere Richtung kann auch einer der HP Ports genutzt werden, mit entsprechend größerer Datenbreite. Es handelt sich bei jedem dieser Schnittstellen um das AXI Protokoll, weiteres hierzu wurde im Kapitel 2.3.2 beschrieben.

Verpflichtende IP-Blöcke

Die PS Seite des Systems wird durch den IP-Block *ZYNQ7 Processing System* repräsentiert (ist dieser nicht vorhanden, wurde das Board nicht korrekt ausgewählt). Über diesen Block können eine Vielzahl von Einstellungen getroffen werden wie z.B. Einstellungen zum Takt, Pins, Spannung etc. Relevant ist zunächst nur die Optionen um die oben genannten Ports zu aktivieren, diese sind zu finden unter der Rubrik *PS-PL Configuration*.

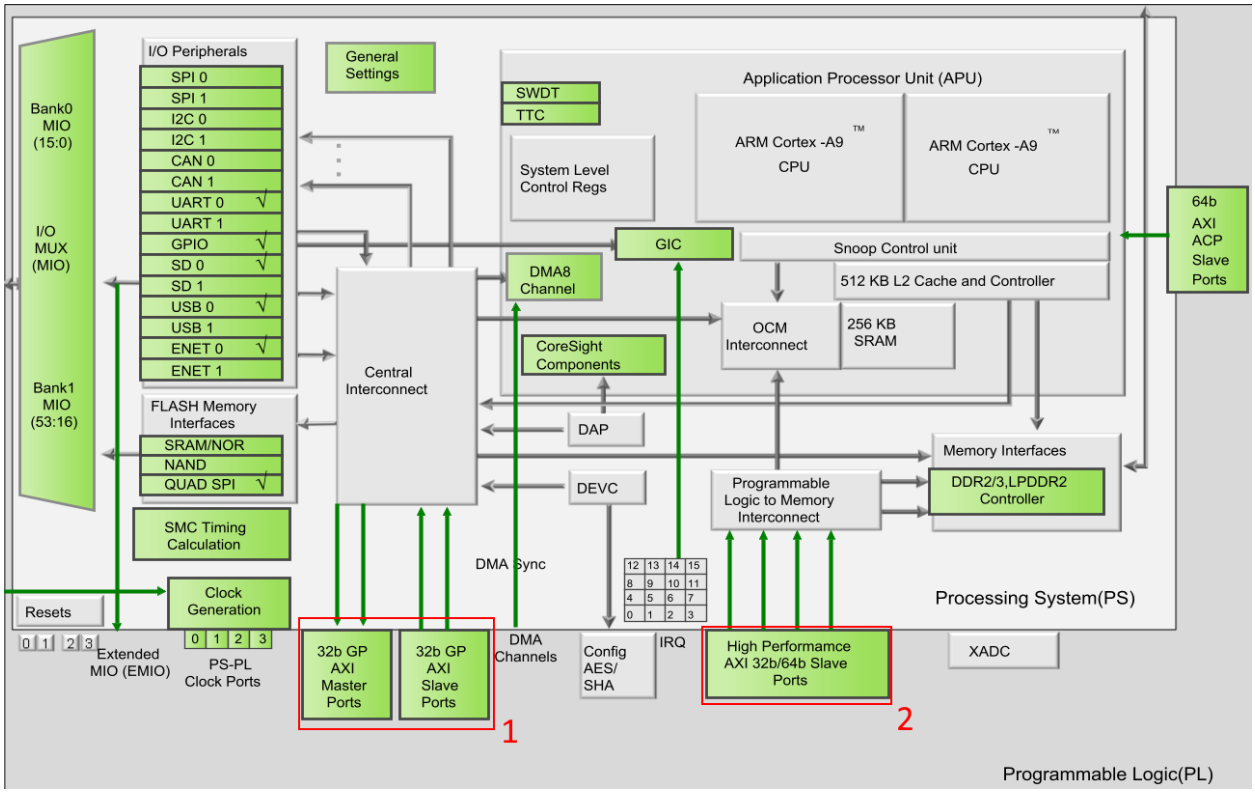


Abbildung 4.3: *ZYNQ7 Processing System* Block Konfiguration

Wenn die Einstellungen zu dem *ZYNQ7 Processing System* Block aufgerufen werden, so wird man mit einer Darstellung wie in der Abbildung gezeigt konfrontiert. Diese zeigt die Gesamtstruktur des Systems auf. Alle mit grüner Farbe hinterlegten Komponenten sind welche, die an dieser Stelle konfiguriert werden können, so beispielsweise auch die GP Ports (1) und die HP Ports (2).

Ein weiterer Block, der Verwendung findet, ist der *Processor System Reset*. Er stellt den Reset für alle weiteren Komponenten und ist entsprechend verbunden mit dem Reset des *ZYNQ7 Processing System*. Da in diesem Fall alle Komponenten den gleich Takt von der PS Seite bekommen ist der Block nicht zwingend verpflichtend, aber dennoch empfehlenswert zu benutzen.

Verbinden von AXI Komponenten

Da das AXI Protokoll als eine eins zu eins Verbindung definiert ist, benötigt es einen Weg einen oder mehrere Master zu einem oder mehreren Slaves zu verbinden. Diese Möglichkeit wird durch sogenannte *Interconnects* geschaffen, welche auch als Part des AXI Protokolls behandelt wird [arm20].

In Vivado ist diese Komponente entweder der *AXI Interconnect* IP-Block oder in neueren Versionen der *AXI SmartConnect* IP-Block. Ein solcher Block kann entsprechend der Anzahl der gewünschten Slave bzw. Master Komponenten konfiguriert werden. Er wird wie alle im Folgenden beschriebenen Blöcke an den Takt der PS Seite und an den Reset des *Processor System Reset* Blocks verbunden.

Mindestens zwei solcher *AXI SmartConnect* Blöcke werden benötigt, eins, um einen Block anzusprechen von der PS Seite und einen weiteren für das Ansprechen der PS Seite von der PL Seite.

AXI DMA

Der *AXI Direct Memory Access* [AMD24e] Block ermöglicht Speicherinhalte der PS Seite (PS DRAM) einem IP-Block zur Verfügung zu stellen via eines AXI4-Streams, sowie die Möglichkeit AXI4-Streams eines Blockes in den DRAM der PS-Seite zu schreiben.

In der Konfiguration des Blockes ist die *Scatter Gather Engine* sowie *Allow unaligned transfers* auszuschalten, da das PYNQ-Z2 diese Funktionen nicht unterstützt. Weiterführend:

1. *Width of Buffer Length Register:* Bestimmt die Anzahl an **Bytes**, die aus dem Speicher gelesen werden können, via $2^n - 1$, i.d.R. kann dies auf 26 gestellt werden.
2. *Address Width:* Muss, wenn dieser Block Speicher der PS Seite auslesen soll, auf **32** gestellt werden.
3. *Memory Map Data Width:* Breite der Daten aus dem Speicher, ist entsprechend dem gewählten Port auszuwählen, i.e., wenn ein HP Port genutzt wird, 64 Bit, sonst 32. Die entsprechenden Kanäle (Lese und Schreibkanäle) können an dieser Stelle auch aktiviert werden.
4. *Stream Data Width:* Breite des anzusprechenden IP-Blockes.

In diesem und auch in anderen Blöcken möchte Vivado durch die *Auto* Option selbst die Datenbreite entscheiden, das klappt i.d.R. nicht und sollte deswegen ausgeschaltet werden.

Sollten beide Kanäle (Lese und Schreibkanäle) aktiviert sein, so müssen außerhalb des Taktes und des Resets fünf Leitungen verbunden werden:

1. *S_AXI_LITE:* Konfigurations-Verbindung des DMA Blocks, diese wird über einen entsprechenden SmartConnect an den Master GP Port des *ZYNQ7 Processing System* verbunden.
2. *M_AXI_MM2S und M_AXI_S2MM:* Die *Mapped Memory to Stream* bzw. *Stream to Mapped Memory* Kanäle erlauben das schreiben zu Speicher der PS Seite sowie das Lesen. Diese werden über einen SmartConnect an die Slave Ports des *ZYNQ7 Processing System* verbunden. Hier ist es empfehlenswert zwei Ports zu nutzen, einen für das Schreiben und einen für das Lesen zwecks der Paralellität. Das geht über einen SmartConnect, sollte ein Interconnect stattdessen verwendet werden, so benötigt es zwei.
3. *S_AXIS_S2MM und M_AXIS_MM2S:* Entsprechende Leitungen um die gelesenen Speicherbereiche zu dem IP-Block, welcher angesprochen werden soll, zu streamen. Bzw. den von diesem empfangenen Stream entsprechend in den Speicher zu schreiben.

Erstellen des Bitstreams

Sollte das Blockdesign erstellt und validiert sein, so kann ein Bitstream generiert werden. Wenn dies geschehen ist, kann unter *File* → *Export* → *Export Bitstream File* die Bitstream Datei exportiert werden. Darüber hinaus wird auch die *Hardware Hand off File* benötigt, diese ist nach erfolgreichen generieren des Bitstreams zu finden unter:
`<Projektordner>/<Projekt-Name>.gen/sources_1/bd/<Blockdesign-Name>/hw_handoff`.

Eine beispielhafte Verwendung dieser Methodik anhand der Nutzung des FFT-Blocks kann im Git-Repository *library* unter *MFCC* gefunden werden. Die Verwendung dieses Block kann in dem Kapitel 6.2.2 nachvollzogen werden.

Kapitel 5

Audioinbetriebnahme

geschrieben von Fynn Schur

In diesem Kapitel wird beschrieben, wie der Zugriff auf den ADAU1761 realisiert wurde und wie die aufgenommenen Audiodaten zur weiteren Nutzung verarbeitet wurden. Zunächst wird erläutert, wie die Audioaufnahme mithilfe des BaseOverlay-Moduls erfolgt. Anschließend wird der Ansatz zur Entwicklung eines eigenen Overlays vorgestellt, das eine erweiterte Funktionalität für die Audioverarbeitung auf dem FPGA-Teil des Systems ermöglicht. Zum Abschluss wird erläutert, wie die zeitliche Fensterung der Audiodaten implementiert wurde.

5.1 Audioaufnahme über das BaseOverlay

geschrieben von Fynn Schur

5.1.1 Allgemeines Vorgehen

Um mit dem BaseOverlay-Modul Audioaufnahmen durchführen zu können, muss dieses zunächst importiert werden. Dies erfolgt über folgendes Import-Statement:

```
from pynq.overlays.base import BaseOverlay
```

Das BaseOverlay-Modul ermöglicht über `base.audio` (wobei `base = BaseOverlay("base.bit")` gesetzt wird) den Zugriff auf die Klasse AudioADAU1761. Diese stellt die Methode `record()` bereit, mit der Audio für eine definierte Zeitdauer aufgezeichnet und in einem Puffer abgelegt werden kann. Auf diesen Puffer kann anschließend über `audio.buffer` zugegriffen werden.

Da der ADAU1761-Chip sowohl über einen Line-In- als auch einen Kombianschluss verfügt, muss vor der Aufnahme festgelegt werden, welcher Eingang verwendet werden soll. Dies kann durch Aufruf von `audio.select_microphone()` für den Kombianschluss oder `audio.select_linein()` für den Line-In erfolgen.

5.1.2 Projektspezifische Implementation

Im Rahmen der projektspezifischen Umsetzung wurde die Audioverarbeitung in mehrere Klassen und Dateien aufgeteilt, um eine eigenständige Audiobibliothek zu entwickeln. Der Quellcode ist unter folgendem Link abrufbar:

`https://gitlab.on.hs-bremen.de/labor-bredereke/embeds/wise2425/library`

Zentrales Element der Implementierung ist die Klasse Audio, die auf die AudioADAU1761-Instanz innerhalb des übergebenen Overlays zugreift. Zur Vereinfachung der Auswahl des Audioeingangs wurde

eine Enumeration (`AudioSource`) eingeführt. Die Methode `record()` ermöglicht nun die Aufnahme von Audiodaten für eine angegebene Dauer. Das aufgenommene Signal wird weiterhin im Audiopuffer der AudioADAU1761-Klasse gespeichert, welcher als Eigenschaft der Audio Klasse übernommen wurde.

Zur weiteren Verarbeitung dieser Audiodaten steht die IO-Datei zur Verfügung, die eine `write()`- sowie eine `read()`-Methode implementiert. Diese Methoden ermöglichen das Schreiben und Lesen von Daten an bzw. von einer Speicheradresse über das MMIO-Protokoll. Falls die Speicheradresse eines IP-Blocks bekannt ist, können Audiodaten direkt in dessen Speicherbereich geschrieben werden, sofern der IP-Block über einen AXI-Slave-Konnektor verfügt.

5.2 Audioaufnahme im eigenen Overlay

geschrieben von Fynn Schur

5.2.1 Arbeiten mit einem eigenen Overlay

Da die Audioverarbeitung auf dem FPGA-Teil des Boards erfolgen soll, reicht die Nutzung des BaseOverlay-Moduls allein nicht aus. Stattdessen ist eine Anpassung des Overlays erforderlich. Da die darunterliegenden IP-Blöcke für die Audioverarbeitung weiterhin verfügbar bleiben müssen, wird das BaseOverlay um eigene Overlay-Elemente erweitert und anschließend neu kompiliert. Dieser Prozess wird im nachfolgenden Kapitel näher erläutert.

Durch dieses Vorgehen bleibt der Zugriff auf die bestehenden Klassen des BaseOverlay erhalten, ohne dass sämtliche Funktionen neu implementiert werden müssen. Dies ist von entscheidender Bedeutung, da die standardmäßig in Vivado verfügbaren IP-Blöcke keine direkte (funktionierende) Schnittstelle zum Audio-Chip bereitstellen.

Zudem erfordert der Audio-Chip eine Initialisierung beim Systemstart, um funktionsfähig zu sein. Diese Initialisierung wird im ursprünglichen Overlay mittels C++-Dateien durchgeführt. Daher wurde das BaseOverlay gezielt um eigene IP-Blöcke erweitert, um die notwendige Funktionalität bereitzustellen und eine nahtlose Integration mit dem bestehenden System zu gewährleisten.

5.2.2 Neukompilierung des BaseOverlay

Um das **BaseOverlay** neu zu kompilieren, sind die folgenden Schritte erforderlich:

1. Bereitstellung einer geeigneten Systemumgebung
2. Installation von **Vivado**
3. Beschaffung und Modifikation der benötigten Dateien
4. Anpassung des **BaseOverlay**-Inhalts
5. Ausführen der Neukompilierung
6. Nutzung des **Vivado**-Projekts zur **Generierung des Bitstreams**

Systemumgebung

Im Verlauf der Projektarbeit stellte sich heraus, dass die Neukompilierung des Overlays unter **Windows** nicht funktionierte. Daher wurde ein **Linux-basiertes Betriebssystem**, in diesem Fall NixOS, verwendet.

Ein wesentlicher Vorteil von **NixOS** ist die **vollständige Konfigurierbarkeit über Konfigurationsdateien**, wodurch eine reproduzierbare Systemumgebung sichergestellt wird. Die vollständige Systemkonfiguration

zum Zeitpunkt des letzten erfolgreichen Overlays-Builds kann im folgenden **GitLab-Repository** eingesehen werden:

GitLab: "dotfiles"

Im weiteren Verlauf wird davon ausgegangen, dass eine solche **NixOS-Umgebung** bereitsteht. Unter anderen **Linux-Distributionen** kann der nachfolgende Prozess geringfügig abweichen.

Installation von Vivado

Zur Neukompilierung des **BaseOverlay** werden **TCL-Skripte** in Kombination mit **Vivado** verwendet. Im Projekt wurde die folgende Version eingesetzt:

- **Vivado v2024.2 (64-bit)**
- **Detaillierte Spezifikationen:**
  - **SW Build:** 5239630
  - **IP Build:** 5239520
  - **SharedData Build:** 5239561

Da unter **NixOS** Programme nicht über herkömmliche **Installer** installiert werden können, sind einige zusätzliche Schritte erforderlich. Eine Anleitung zur Installation von **Vivado** unter **NixOS** ist unter folgendem Link verfügbar:

Nix-Xilinx GitLab Repository

Da **Vivado** manuell installiert wird und somit nicht in der **Lock-Datei der NixOS-Flake** fixiert ist, muss für eine maximale **Reproduzierbarkeit** exakt die gleiche **Vivado-Version** verwendet werden.

Beschaffung und Modifikation der benötigten Dateien

Das **BaseOverlay** besteht aus einer Vielzahl von Dateien, die in einem öffentlichen **GitHub-Repository** von **Xilinx** verfügbar sind. Diese können mit folgendem **Befehl** lokal geklont werden:

```
git clone https://github.com/Xilinx/PYNQ
```

Für das **PYNQ-Z2 Board** befinden sich die relevanten Dateien unter folgendem Pfad:

```
../PYNQ/boards/Pynq-Z2/base/
```

In diesem Verzeichnis muss die Datei **base.tcl** angepasst werden, um die **Vivado-Version** auf **2024.2** zu setzen. Andernfalls wird das **TCL-Skript** nicht korrekt ausgeführt.

Anpassung des BaseOverlay-Inhalts

Um eigene Komponenten in das **BaseOverlay** zu integrieren, müssen diese im folgenden Verzeichnis abgelegt werden:

```
../PYNQ/boards/ip
```

Alle **benutzerdefinierten IP-Blöcke**, die in diesem Verzeichnis abgelegt werden, stehen im daraufhin generierten **Vivado-Projekt** zur Verfügung.

Ausführen der Neukompilierung

Zur **Neukompilierung** des **BaseOverlay** müssen die folgenden Befehle im Verzeichnis

`../PYNQ/boards/Pynq-Z2/base/`

ausgeführt werden:

```
nix shell nixpkgs#xsct
make all
```

Der anschließende **Build-Vorgang** kann einige Zeit in Anspruch nehmen.

Nutzung des Vivado-Projekts zur Generierung des Bitstreams

Nach erfolgreichem **Build-Vorgang** wird ein **Vivado-Projekt** erzeugt. Beim Öffnen des Projekts ist die **Standardkonfiguration** des **FPGA-Boards** sichtbar.

In diesem Schritt kann das **BaseOverlay** an die spezifischen Anforderungen angepasst werden. Sobald die gewünschten Änderungen vorgenommen wurden, muss der **Bitstream** aus dem Projekt generiert und anschließend auf das **FPGA-Board** übertragen werden.

5.3 Zeitliche Fensterung

geschrieben von Jesse Gollub

Das Aufnehmen der Audiosignale erfolgt in Python über die `record`-Funktion der `AudioADAU1761`-Klasse aus der `pynq`-Bibliothek. Diese Funktion nimmt eine Fließkommazahl als Parameter für die Aufnahmedauer in Sekunden entgegen und speichert das Ergebnis im Buffer des `audio`-Objekts. Allerdings bietet diese Methode keine Möglichkeit, kontinuierlich Audiodaten aufzunehmen, weshalb eine zeitliche Fensterung notwendig wird.

Um dies zu ermöglichen, wird ein `Deque` (Double-Ended Queue) verwendet, der als Ringpuffer fungiert. Der Puffer speichert eine festgelegte Anzahl an Audiodaten. Sobald der Puffer voll ist, werden die ältesten Daten am Ende des Puffers entfernt, während die neuen Daten am Anfang eingefügt werden.

Die Fensterung erfolgt in einer Schleife, die fortlaufend kleine Audioschnipsel aufnimmt und diese in den Puffer einfügt. Sobald der Puffer die maximal festgelegte Größe erreicht hat, werden die gesammelten Daten verarbeitet und über das `MMIO` (Memory-Mapped I/O) an eine definierte Adresse weitergeleitet, um sie im weiteren Verlauf zu analysieren (siehe Kapitel 6).

Das Verpacken der kleinen Audioschnipsel in einen Puffer ist entscheidend, weil für eine präzise Audioanalyse stets der vollständige Kontext eines Tones benötigt wird. Einzelne, isolierte Datenpunkte sind oft nicht aussagekräftig genug, weshalb das Sammeln von Audiodaten über einen festgelegten Zeitraum hinweg ermöglicht, den gesamten Ton und seine Eigenschaften zu erfassen.

Kapitel 6

Audioverarbeitung

6.1 Implementierung via Python

geschrieben von David Schulz

mit geschrieben von Tobias Guimaraes Zimmer

Die Bestimmung der „Mel Frequency Cepstral Coefficients“ ist nötig damit das Neuronale Netz Wörter erkennen kann. Die Implementation dieser Berechnung geschah mittels Python, da das PYNQ-Z2 Board eine Python Umgebung bereits anbietet. Ziel dieser Implementation ist zusätzlich zur eigentlichen Berechnung ein Fundament zu bieten, welches dazu verwendet werden kann, die einzelnen Schritte der Berechnung in IP-Blöcke für das FPGA-Board zu überführen. Aus diesem Grund wurde bei der Implementation keine Python-Bibliothek verwendet, da diese es nicht erlaubt hätten nur Teile der Berechnung in IP-Blöcke für das FPGA-Board auszulagern. Um ein grundlegendes Verständnis und Vergleichswerte zu erlangen, wurde zunächst mittels der Python-Bibliothek Librosa die Berechnung umgesetzt. Danach wurden mehrere Implementationen erstellt, da auch mehrere Ansätze denkbar wären z.B. bei dem Erstellen der Filterbänke oder beim Dekorrelieren der Werte. Zwei dieser Implementationen versuchen den Ergebnissen der Python-Bibliothek Librosa nahezukommen, in dem eine eigene Berechnung der MFCCs erstellt wurde. Der Unterschied zwischen den Implementationen liegt darin, dass in der einen Implementation „Mel Frequency Cepstral Coefficients“ durch eine diskrete Kosinustransformation bestimmt werden und in der anderen durch eine äquivalente Berechnung mittels einer erneuten Verwendung der Fast Fourier Transformation. Die dritte Implementation beruht auf der Rekonstruktion der Python-Bibliothek Librosa, wobei der Quellcode auf das wesentliche reduziert wurde um eine Implementation durch IP-Blöcke im Nachhinein zu ermöglichen. Es wurde sich auf die Implementation geeinigt, in der die diskrete Kosinustransformation durch die Fast Fourier Transformation berechnet wurde, da in dieser Implementation der vorhandene FFT-IP-Block verwendet werden kann. Dadurch entfällt die eigene Implementation der diskreten Kosinustransformation in VHDL als IP-Block.

Alle Implementationen zur Berechnung der „Mel Frequency Cepstral Coefficients“ erhalten als Eingangssignal die Werte der aufgenommenen Tonspur. Dieses Signal wird zuerst in kleinere Blöcke mit jeweils 20 ms Länge und der Schrittgröße von 10 ms unterteilt, siehe dazu Quellcode 6.1. Auf diese Blöcke wird die Hamming-Fensterfunktion angewendet um Kanteneffekte zu reduzieren. Diese Fensterfunktion erreicht dies, indem sie die mittleren Werte hervorhebt und die am Rand liegenden Werte abschwächt. Eine Tonspur muss mindestens 20 ms lang sein, damit die Implementierung funktioniert.

```
1 windowed_matrix = segment_signal(raw_audio, sample_rate)
```

Listing 6.1: Quellcode für die Segmentierung des Eingabesignals in kleinere Fenster.

Anschließend wird für jeden Block die diskrete Fourier Transformation (mittels FFT) bestimmt, siehe Quellcode 6.2.

```
1 fft_frames = np.fft.fft(fft_frames)
```

Listing 6.2: Quellcode für Berechnung des FFTs für das segmentierte Eingabesignal.

Aus dem Ergebnis der diskrete Fourier Transformation wird das Betragsspektrum erzeugt, siehe dazu Quellcode 6.3.

```
power_spectrum = np.abs(fft_frames) ** 2
```

Listing 6.3: Quellcode für die Bestimmung des Betragsquadrats

Das resultierende Betragsspektrum wird anschließend durch Mel-Filterbank gefiltert, siehe Quellcode 6.4. Zur Bestimmung dieser Filter wurden verschiedene Ansätze probiert. Es handelt sich prinzipiell um Dreiecksfilter, die über gewisse Frequenzen gespannt werden. Diese werden in Abhängigkeit der maximalen und minimalen präsenten Frequenzen berechnet. Sowohl die Bartlett-Funktion als auch eine einfache Verwendung einer Dreiecksfunktion wurden implementiert, um eine Überführung in IP-Blöcke zu vereinfachen. Die Entscheidung fiel allerdings auf eine Konstruktion der Filter mittels der Numpy `linspace` Funktion, da diese immer sicher Dreiecke liefert, die einen Höhepunkt bei exakt 1 haben. Bei den anderen Filterfunktionen ist dies nicht zwingend der Fall, insgesamt kann dem entgegengewirkt werden durch das Normalisieren der Filter. Darauf wurde in dieser Implementation aber zunächst verzichtet, um die nachfolgende Implementation in IP-Blöcken nicht weiter zu erschweren.

```
# filter points (start and stop) and corresponding frequencies
filter_points, frequency = get_filter_points(min_freq, max_freq,
mel_filter_num, n_fft, sample_rate)
filters = get_filters(filter_points, n_fft)
# take right side of spectrum
right = power_spectrum[:, :len(power_spectrum[0]) // 2 + 1]
# convolution with filters
filtered_frames = np.dot(filters, right.T)
```

Listing 6.4: Quellcode für Berechnung der Filter und der Filterung des Betragsspektrums.

Anschließen wird der Logarithmus von den gefilterten Werten gebildet, siehe Quellcode 6.5.

```
filtered_frames = 10.0 * np.log10(filtered_frames)
```

Listing 6.5: Quellcode für Berechnung des Logarithmus von dem Segmentierten Eingabesignal.

Zuletzt muss eine Dekorrelation durchgeführt werden. Die Implementationen unterscheiden sich im Wesentlichen in diesem Schritt, da hier entweder die diskrete Kosinustransformation verwendet wurde oder durch eine erneute Fourier Transformation die Dekorrelation umgesetzt wurde, siehe Quellcode 6.6. Es ist möglich die diskrete Kosinustransformation durch eine Fourier Transformation zu bestimmen, indem man den Realteil der Fourier Transformation bestimmt, falls es ein reellwertiges Signal ist [Mer23].

```
filtered_frames = filtered_frames.T
filtered_frames_copy = np.flip(filtered_frames, 1)

prepared_filtered_frames = np.append(filtered_frames, filtered_frames_copy, 1)

second_fft = np.fft.fft(prepared_filtered_frames).real
```

Listing 6.6: Quellcode für die Berechnung der diskreten Kosinustransformation durch die Fast Fourier Transformation.

6.2 Implementierung via VHDL für das FPGA

geschrieben von Moses Dimmel

Um die Geschwindigkeit der Audioverarbeitung zu erhöhen muss die Berechnung der "Mel Frequency Cepstral Coefficients"via VHDL auf dem FPGA umgesetzt werden. Dafür wurden die einzelnen Schritte

der Berechnung in eigene IP-Blöcke aufgeteilt um die Komplexität zu verringern und für die Berechnung der FFT den von Vivado bereitgestellten IP-Block zu verwenden (siehe 6.1). Implementiert wurde vorerst die Hamming-Fensterung, die Berechnung des Betrages und die DCT. Die FFT wurde mittels eines von Vivado bereits gestellten IP-Blöcke berechnet.
Allerdings traten erhebliche Schwierigkeiten bei der Rechnung mit Floating-Point Werten auf, da die Verwendung des Datentypen „float“ oder „double“ für die Entwicklung auf dem FPGA zu ressourcenintensiv ist. Benötigt wurde ein Fixed-Point Datentyp. Hier wird zuvor im Datentyp selbst angegeben, wie viele Bits für den Integer-Teil, und wie viele für den irrationalen Teil zugeteilt werden sollen. Eine offizielle Implementation von Fixed-Point für VHDL ist jedoch leider nicht bekannt. In einem Thread des AMD-Forums [Mem] konnte eine Implementierung gefunden werden, die Abhängigkeiten dazu konnten auf Github [Aut] gefunden werden. Jedoch traten bei der Verwendung dieser Implementierung kritische Fehler auf und war demnach erfolglos. Imzugedessen wurde an dieser Stelle auf die Implementierung der "Mel Frequency Cepstral Coefficients"mittels HLS umgeschwenkt.

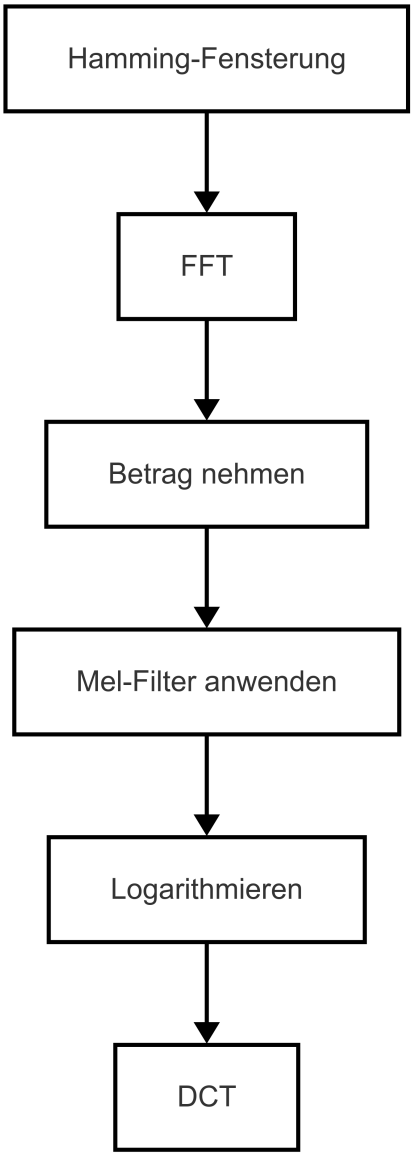


Abbildung 6.1: Abbildung zeigt die einzelnen Schritte in die die Berechnung der "Mel Frequency Cepstral Coefficientsäufgeteilt wurde.

6.2.1 Hamming-Fensterung

geschrieben von Moses Dimmel

Das vom Mikrofon aufgenommene Audiosignal wird zunächst in Frames aufgeteilt und als AXI-Stream an den entwickelten IP-Block zur Fensterung des Signal nach Hamming weitergegeben (siehe B.1.3). Bei einer Sample-Rate von 48 000 Hz und und Audioframes von 20ms würde sich beispielsweise eine Fenstergröße von 960 Samples pro Frame bilden. Diese Samples werden jeweils mit den entsprechenden Hamming-Koeffizienten multipliziert. Dadurch werden die äußeren Werte innerhalb eines Frames abgeschwächt, wodurch mögliche spektrale Leckagen durch die Aufteilung in Frames verringert werden können.

```
m_axis_tdata <= std_logic_vector(to_signed((to_sfixed(signed(s_axis_tdata), 15,
    -16)) * HAMMING_COEFF(index), 32));
```

Listing 6.7: VHDL-Code für die Anwendung der Hamming-Fensterung

Da auf FPGA nur wenige Ressourcen verfügbar sind und die Rechenleistung des FPGA nicht sehr groß ist, müssen die Hamming-Koeffizienten statt live berechnet zu werden zuvor durch bspw. ein Python-Script B.1.3 berechnet werden und in VHDL im BRAM abgelegt werden.

Die Hamming-Fensterung agiert mittels eines Index-Zählers, welcher sich beim Erreichen des Endes der Fenstergröße zurücksetzt und somit immer die korrekten Hamming-Koeffizienten verrechnet, ohne das Audiosignal zu buffern. Nach der Verrechnung mit den Hamming-Koeffizienten würde ein Audio-Sample an den vorgefertigten FFT-Block über die AXI-Schnittstelle weitergeleitet werden.

6.2.2 FFT Block

geschrieben von David Schulz

Zur Berechnung der Fourier Transformation wurde ein fertiger IP-Block verwendet. Dieser FFT-IP-Block implementiert den Cooley-Tukey FFT-Algorithmus um die diskrete Fourier Transformation zu berechnen. Dabei ist es möglich den FFT-IP-Block so zu konfigurieren, dass dieser entweder die Fast Fourier Transformation (FFT) oder die Inverse Fast Fourier Transformation (IFFT) durchführt. Zur Berechnung der Inverse Fast Fourier Transformation (IFFT) verwendet der IP-Block das Verfahren bei dem erst die Fast Fourier Transformation gebildet wird und dann eine komplexe Konjugation des „Phase Faktors“ vorgenommen wird, damit erhält man nicht die tatsächliche Inverse Fast Fourier Transformation, da hierbei noch mit $1/\mathrm{n}$ die Werte skaliert werden müssen (dabei ist n die Anzahl der Datenpunkte). Dies kann allerdings über die Einstellung des Skalierungsfaktors vorgenommen werden. Im Kern des FFT-IP-Blocks wird entweder ein Radix-4 oder Radix-2 Verfahren verwendet, welches von beiden verwendet wird, wird entweder automatisch entschieden oder kann manuell eingestellt werden. Der FFT-IP-Block ist außerdem für die Verwendung mit AXI4-Streams ausgelegt. Dabei sind alle Eingabeleitungen und Ausgabeleitungen AXI4-Streams außer `aclk`, `aclken` und `aresetn` und die Event Signale.[AMD24a] [AMD24b] [AMD24c]

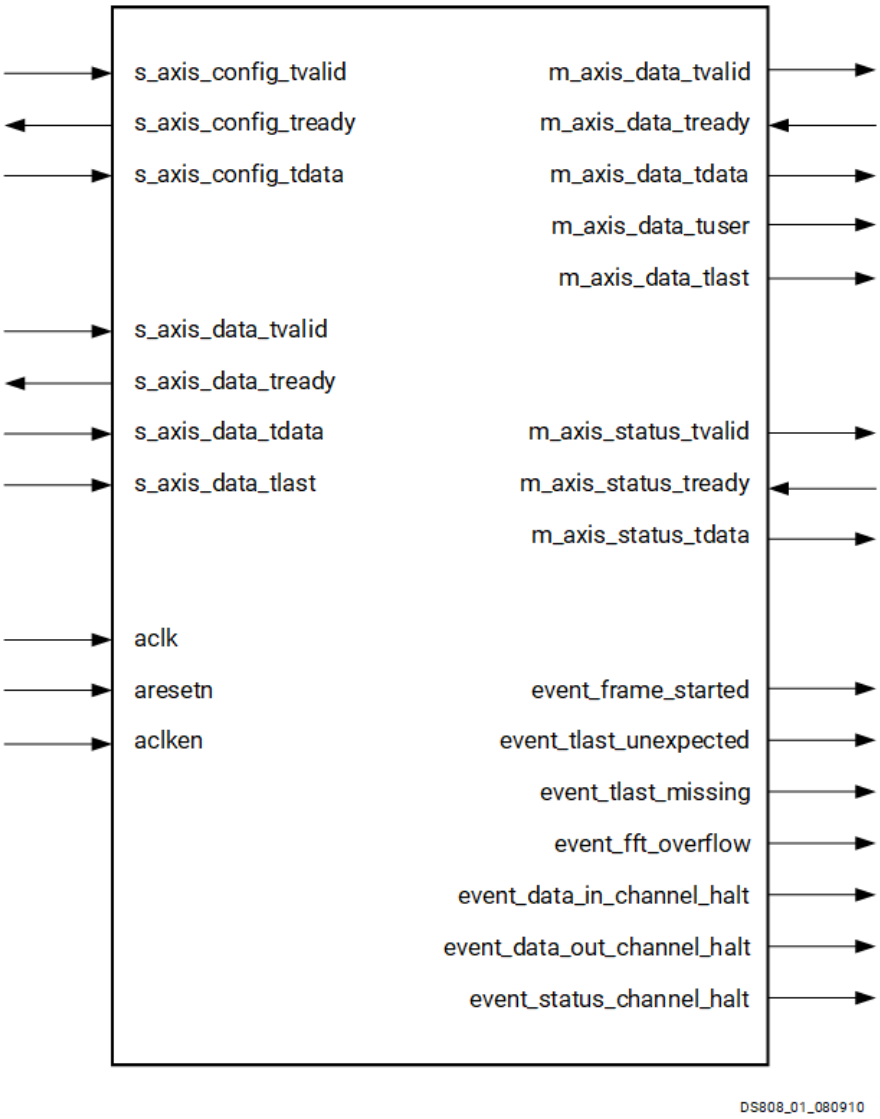


Abbildung 6.2: Die Abbildung zeigt den FFT-IP-Block mit seinen Inputs und Outputs, dabei sind alle Inputs und Outputs AXI4-Streams außer `aclk`, `aclken` und `aresetn` und die Event Signale.

Der FFT-IP-Block kann zwar auch zur Laufzeit konfiguriert werden, das ist in diesem Anwendungsfall allerdings nicht nötig. [AMD24d]

Der FFT-IP-Block wurde in einem Overlay für das FPGA-Board hinzugefügt, wo dann die Eingabeleitungen und Ausgabeleitungen mit den AXI4-Streams verbunden wurden, siehe Abbildung C.1. Außerdem wurde Python-Quellcode geschrieben um diesen IP-Block vom Mikrocontroller aus anzusprechen und um die FFT für Daten bilden zu können.

Die Implementierung des IP-Blocks geschah wie in Kapitel 4.2 beschrieben und kann im Git-Repository *library* unter *MFCC* gefunden werden. Weiterführend kann ein Blockdiagramm im Anhang gefunden werden, Abbildung C.1.

6.2.3 Berechnung des Betrages

geschrieben von Moses Dimmel

Nach der Berechnung der FFT empfängt der Block zur Berechnung des Betrages dieser einen reellen und imaginären Anteil der Analyse (siehe B.1.3). Da die Berechnung einer Wurzel auf dem FPGA sehr rechenintensiv ist, wird aus diesen Werten anschließend der Betrag lediglich angenähert durch die Summe der einzelnen Beträge des Real- und Imaginäranteils.

```
abs_re <= unsigned(abs(re));
abs_im <= unsigned(abs(im));

-- Magnitude Berechnen (Angenaehert durch: |Re| + |Im|)
magnitude <= abs_re + abs_im;
```

Listing 6.8: VHDL-Code für die Berechnung des Betrages.

Im Anschluss wird der gebildete Betrag über den AXI-Stream an den Block für die Mel-Filterung und das Logarithmieren weitergeleitet werden.

6.2.4 Berechnung der DCT

geschrieben von Moses Dimmel

Die Implementierung der DCT-Berechnung wurde nicht fertiggestellt, sondern es wurde stattdessen wie beschrieben zur HLS-Implementierung umgeschwenkt. Nichtsdestotrotz ist der bestehende Code im Anhang B.1.3 zu finden. Auch hier wurden die Kosinus-Werte durch ein Hilfsscript B.1.3 generiert, sodass sie im BRAM des FPGA gespeichert werden können da die Berechnung des Kosinus für das FPGA zu aufwändig wäre.

6.3 Implementierung via HLS für das FPGA

geschrieben von Moses Dimmel

Als ein zweiter Ansatz wurde die Berechnung der „Mel Frequency Cepstral Coefficients“ wurde die High-Level Synthesis mithilfe der Entwicklungsumgebung Xilinx Vitis HLS gewählt. In C++ wurde hier die Logik für die einzelnen Schritte der Berechnung durchgeführt (siehe B.1.4). Dabei konnten spezielle von Vitis optimierte Methoden für bspw. das Ziehen einer Wurzel (hls::sqrt()) verwendet werden und weitere Optimierungsverfahren wie Pipelining umgesetzt werden. Der C++-Code wurde anschließend vom Compiler in VHDL bzw. Verilog umgewandelt und ein IP-Block erstellt. Dazu muss in Vitis HLS die Synthese gestartet werden und anschließend von der Menüleiste die „Register Transfer Language (RTL)“ exportiert werden um einen IP-Block zu exportieren. Dieser kann in Vivado eingebunden werden, indem im IP-Katalog per Rechtsklick die Option „Add Repository“ gewählt wird und der entsprechende Ordner angegeben wird, indem sich der exportierte IP-Block befindet. In den folgenden Unterkapiteln wird die Implementierung der „Mel Frequency Cepstral Coefficients“ beschreiben.

6.3.1 Hamming-Fensterung

geschrieben von Moses Dimmel

Das Audiosignal wird zunächst per AXIS-Stream eingelesen. Dieser ist mit folgendem Typ in der Header-Datei (siehe B.1.4) deklariert:

```
typedef ap_axis<32,2,5,6> axis_t;
typedef hls::stream<axis_t> axis_stream;
```

Listing 6.9: C++-Code für die Typdefinition der Eingabe- und Ausgabeparametern

Nach dem Einlesen eines Audio-Samples wird dieses mit dem entsprechenden Hamming-Koeffizienten multipliziert, sodass pro Fenster, hier bspw. 512 Samples groß, jeweils die Werte an den Rändern weniger signifikant und die Werte zur Mitte des Fensters hin signifikanter gewertet werden um Leckage im

Spektralbereich zu vermeiden.

```
windowed_audio[i] = in_sample.data * hamming_window[i];
```

Listing 6.10: C++-Code für die Hamming-Fensterung

Die vorberechneten Koeffizienten des Hamming-Fensters sind vorberechnet in einem Array abgelegt, um Rechenleistung zu sparen.

6.3.2 Berechnung der FFT und des Betrages

geschrieben von Moses Dimmel

Für die Berechnung der FFT wurde bei der HLS-Implementierung ebenfalls zum Sparen von Ressourcen der FFt IP-Block von Vivado verwendet. Dazu wird jeder Sample-Wert über den 64 Bit breiten AXIS-Stream an den FFT-Block gesendet. Die oberen 32 Bit beinhalten den imaginären Anteil, hier 0. Die unteren 32 Bit enthalten die Audio-Fensterung.

```
// Ausgabe an den FFT-Block
in_sample.data.range(63, 32) = 0;
in_sample.data.range(31, 0) = windowed_audio;
fft_in_axis.write(in_sample);

...

// Verarbeiten der Eingabe vom FFT-Block
real.range(31, 0) = out_sample.data.range(31, 0);
imag.range(31, 0) = out_sample.data.range(63, 32);
real = real * factor;
imag = imag * factor;
fft_magnitude[i] = hls::sqrt(real * real + imag * imag);
```

Listing 6.11: C++-Code für die FFT-Berechnung und Berechnung des Betrages

Nach Erhalt der Ergebnisse der Berechnung des FFT-Blocks wird der reelle und imaginäre Anteil der Frequenzinformationen jeweils extrahiert und mit dem Faktor multipliziert, mit dem der FFT-Block die Ergebnisse zur Vermeidung von Overflow herunterskaliert, um wieder das korrekte Ergebnis zu erhalten. Dieser Faktor muss vor der Berechnung im FFT-Block konfiguriert werden.

Schlussendlich wird nun der Betrag der komplexen Zahl ermittelt.

6.3.3 Anwendung der Mel-Filterbank und Logarithmierung

geschrieben von Moses Dimmel

Im Anschluss an die Betragsberechnung der FFT wird die Mel-Filterbank angewandt und der Logarithmus berechnet. Um Rechenfehler zu vermeiden wird der Eingabe der Logarithmusfunktion zuvor ein sehr kleiner Wert > 0 addiert. Wie bei der Hamming-Fensterung sind die vorberechneten Werte der Mel-Filterbank in einem Array abgespeichert um Rechenleistung zu sparen.

```
for (int k = 0; k < FFT_SIZE; k++) {
  mel_energy[m] += fft_magnitude[k] * mel_filterbank_lut[m][k];
}
```

```
mel_energy[m] = hls::log(hls::abs(mel_energy[m]) + fixed_t(1e-6));
```

Listing 6.12: C++-Code für die Mel-Filterung und Logarithmierung

6.3.4 DCT-Berechnung

geschrieben von Moses Dimmel

Die DCT (Diskrete Kosinus-Transformation) kann effizient mithilfe der FFT errechnet werden. Hierfür werden zunächst die logarithmierten Werte als realen Anteil an den FFt-Block gesendet. Der Array wird allerdings dabei dupliziert, wobei die zweite Hälfte gespiegelt gesendet wird. Das Ergebnis dieser Rechnung sind nach Rückgängigmachung der Skalierung die Mel-Koeffizienten. Da für die Spracherkennung nur die unteren Koeffizienten benötigt werden, werden hier die ersten 20 an die Ausgabe gesendet und per AXIS-Streams ausgegeben.

```
in_sample.data.range(63, 32) = 0;
in_sample.data.range(31, 0) = mel_energy[i];
fft_in_axis.write(in_sample);

...

real.range(31, 0) = out_sample.data.range(31, 0);
real = real * factor;

if (i < MEL_BANDS) {
  mfcc[i] = real;
}
```

Listing 6.13: C++-Code für DCT-Berechnung

Kapitel 7

Erstellen und Trainieren des NN

geschrieben von Friederike Korte

Von der Vorgängergruppe ist bereits ein umfassendes Python-Projekt angelegt worden, in dem die Definition der Modelle, das Training und auch das anschließende Quantisieren in das ONNX-Format definiert ist. Da das Einlesen in den Code aufgrund fehlender Docstrings und Type-Hinting jedoch mit einem erheblichen Zeitaufwand verbunden ist und wir zusätzlich auch die Daten etwas anders vorverarbeiten wollen, haben wir ein komplett neues Python-Projekt angelegt.

7.1 Struktur des Neuronalen Netzes

geschrieben von Serkay-Günay Celik

7.1.1 Design der Superklasse für Neuronale Netze

Die Implementierung neuronaler Netze erfordert eine strukturierte und modulare Herangehensweise, um die Wiederverwendbarkeit und Erweiterbarkeit des Codes zu verbessern. Im Rahmen dieses Projekts wurden zwei Superklassen entwickelt, um verschiedene Arten von Netzwerkarchitekturen effizient zu verwalten:

1. `BaseModelCustom`: Diese Klasse dient als flexible Basis für Modelle, die eine individuelle `forward`-Methode benötigen. Sie ermöglicht die Erstellung nicht-sequenzieller oder komplexer Netzwerkarchitekturen, die nicht durch eine einfache Schichtabfolge beschrieben werden können.
2. `BaseModelSequential`: Diese Klasse ist für sequentielle Modelle gedacht, bei denen die Reihenfolge der Schichten bereits festgelegt ist. Sie erlaubt eine effiziente und unkomplizierte Implementierung, da die `forward`-Methode automatisch durch das `self.model`-Attribut definiert wird.

Durch die Nutzung dieser Superklassen wird die Entwicklung neuer neuronaler Netzwerke erheblich erleichtert, da die Grundstruktur bereits vorgegeben ist und sich Entwickler auf die spezifischen Modellarchitekturen konzentieren können.

7.1.2 Die `BaseModelCustom`-Klasse

Die `BaseModelCustom`-Klasse ist für Modelle mit einer individuellen Netzwerkstruktur konzipiert. Sie dient als abstrakte Basisklasse und zwingt jede Unterklasse dazu, eine eigene `forward`-Methode zu definieren. Dies ist besonders nützlich für rekurrente Netzwerke (RNNs), mehrspurige Architekturen oder Graph Neural Networks (GNNs), die sich nicht durch eine einfache sequentielle Schichtenstruktur ausdrücken lassen.

Der Hauptvorteil dieser Klasse ist ihre Flexibiltät, da sie keine festen Annahmen über die Modellarchitektur trifft. Stattdessen bietet sie eine leere `forward`-Methode, die in jeder Unterklasse individuell implementiert werden muss. Dadurch können Entwickler den Datenfluss innerhalb des Netzwerks vollständig anpassen.

Falls eine Unterklasse keine eigene `forward`-Methode implementiert, wird automatisch die Exception `NotImplementedError` ausgelöst, um Fehler frühzeitig zu erkennen.

7.1.3 Die `BaseModelSequential`-Klasse

Im Gegensatz zur `BaseModelCustom`-Klasse ist die `BaseModelSequential`-Klasse für vordefinierte sequenzielle Netzwerke optimiert. Sie vereinfacht die Implementierung von klassischen Feedforward-Netzwerken, Convolutional Neural Networks (CNNs) und einfachen Multilayer Perceptrons (MLPs).

Die `BaseModelSequential`-Klasse basiert auf der Annahme, dass das Modell eine feste Reihenfolge von Schichten hat. Anstatt eine eigene `forward`-Methode zu schreiben, müssen Entwickler lediglich ein `self-model`-Attribut in der Unterklasse definieren. Dieses Attribut enthält die Schichten des Netzwerkes als `nn.Sequential`-Objekt, wodurch die Verarbeitung automatisch im `forward`-Schritt durchgeführt wird.

Ein weiterer Vorteil dieser Struktur ist die Reduzierung von Boilerplate-Code, da Entwickler nicht manuell eine Forward-Methode schreiben müssen. Sollte `self.model` in einer Unterklasse nicht definiert sein, wird beim Aufruf der `forward`-Methode eine `AttributeError` ausgelöst, um eine korrekte Implementierung sicherzustellen.

7.1.4 Beschreibung des Modells `PreviousModel`

geschrieben von Friederike Korte

Im Vorgängerprojekt ist bereits ein sehr erfolgreiches Modell definiert und trainiert worden. Mit den vorliegenden Audiodateien hat es eine Erkennungsrate von ca. 90 Prozent erreicht. Allerdings gab es erhebliche Probleme bei der Erkennung von Befehlen, die das Model direkt über das Mikrofon bekommen hat (nur 5% Erkennungsrate).

Die Vermutung ist, dass dies auf die Audioschnittstelle des dort verwendeten FPGA-Boards zurückzuführen ist. Da wir in diesem Projekt das Nachfolgemodell des dort verwendeten FPGA-Boards zur Verfügung haben, das eine verbesserte Audioschnittstelle besitzt, definieren und trainieren wir dieses Modell hier ebenfalls.

Das Modell besteht dabei aus insgesamt 13 Layern mit vier unterschiedlichen Layer-Typen, die sich sequentiell wiederholen. Die genau Beschreibung der Layer und des Modells ist in [Bur23] in Kapitel 5.1 nachzulesen.

7.1.5 Beschreibung des Modells `Model2RNN`

geschrieben von Serkay-Günay Celik

Das in diesem Projekt verwendete neuronale Netz, Model2RNN, kombiniert Convolutional Neural Networks (CNNs) mit Recurrent Neural Networks (RNNs), insbesondere mit einer Long Short-Term Memory (LSTM)-Schicht. Diese hybride Architektur wurde gewählt, um sowohl lokale Muster in den Eingabedaten als auch deren zeitliche Abhängigkeiten effizient zu erfassen.

Das Modell basiert auf der BaseModelCustom-Superklasse, was eine hohe Flexibilität ermöglicht und eine individuelle `forward`-Methode zur Verarbeitung der Daten erfordert.

Architektur des Modells

Das Modell besteht aus drei Hauptkomponenten:

1. **CNN-Teil (Feature-Extraktion)**
    - Besteht aus zwei quantisierten Linear-Schichten (QuantLinear) mit einer Quantized ReLu (QuantReLu)-Aktivierungsfunktion dazwischen
    - Diese Schichten helfen dabei, wichtige Merkamle aus den Eingangsdaten zu extrahieren und zu transformieren
    - Eine Dropout-Schicht wird genutzt, um Overfitting zu vermeiden
2. **RNN-Teil (LSTM für Sequenzverarbeitung)**
    - Die aus dem CNN extrahierten Features werden als Eingabe für eine LSTM-Schicht verwendet
    - Diese Schicht verarbeitet die zeitlichen Abhängigkeiten innerhalb der Sequenz, um eine bessere Sprachkommandoerkennung zu ermöglichen
    - Die $batch\_first = True$-Option sorgt dafür, dass die Batch-Dimension an erster Stelle steht, was das Training effizienter macht
3. **Klassifikations-Teil (Fully Connected Layer + BatchNorm)**
    - Der letzte LSTM-Ausgabewert wird an eine quantisierte vollverbundene Schicht (QuantLinear) weitergegeben.
    - Diese Schicht reduziert die dimensionalen Merkmale auf die Anzahl der Ausgabekategorien (z.B. verschiedene Sprachkommandos)
    - Eine Batch-Normalization-Schicht stabilisiert die Ausgabe

Forward-Pass Ablauf

Der Forward-Pass beschreibt, wie die Eingabedaten durch das neuronale Netzwerk verarbeitet werden, bis eine endgültige Klassifikation erfolgt. In unserem Fall kombiniert das Modell CNN-Schichten zur Merkmalsextraktion mit einer LSTM-Schicht, um zeitliche Abhängigkeiten zu erfassen. Anschließend erfolgt die Klassifikation durch eine quantisierte vollverbundene Schicht. Der genaue Ablauf ist wie folgt:

1. Der Eingangstensor (Batch, Features, Time) wird durch die CNN-Schichten transformiert, um relevante Merkmale aus den Rohdaten zu extrahieren.
2. Die Daten werden so umgeordnet (`permute`), dass sie für die LSTM-Schicht geeignet sind.
3. Das LSTM-Netzwerk verarbeitet die Daten sequenziell und generiert eine zeitliche Repräsentation.
4. Der letzte Zeitschritt des LSTM wird als endgültige Feature-Darstellung extrahiert.
5. Eine quantisierte vollverbundene Schicht gibt die endgültigen Klassifikationswerte aus.
6. Batch-Normalization wird angewendet, um stabile Vorhersagen zu ermöglichen und Schwankungen im Modell zu reduzieren.

Warum diese Architektur?

- CNNs helfen dabei, wichtige Merkmale aus Sprachdaten zu extrahieren.
- LSTMs sind besonders gut darin, zeitliche Muster in der Sequenz zu erkennen.
- Die Kombination aus beiden Modellen verbessert die Genauigkeit der Sprachkommandoerkennung, da sowohl lokale als auch globale Abhängigkeiten berücksichtigt werden.
- Die Verwendung von quantisierten Schichten sorgt dafür, dass das Modell für die FPGA-Implementierung optimiert ist.

7.1.6 Weitere Modelle

geschrieben von Friederike Korte

Zusätzlich zu den beiden bereits beschriebenen Modellen haben wir einige weitere Modelle definiert, die teilweise noch weitere Layer-Typen implementieren, als die bisher verwendeten im `PreviousModel` und `Model2RNN`.

Diese Modelle sind jedoch nicht trainiert worden, da beide Trainingsansätze nicht erfolgreich verlaufen sind und der Fokus auf den beiden beschriebenen Modellen lag.

7.2 Datenvorverarbeitung für das Training

geschrieben von Friederike Korte

Bevor ein Neuronales Netz mit dem vorliegenden Datensatz trainiert werden kann, müssen diese Daten vorverarbeitet und in die richtige Struktur gebracht werden. Dabei wird in der Regel auch der Datensatz bereinigt, indem z.B. fehlerhafte Daten aus dem Datensatz entfernt werden. Da dies bereits erfolgreich von unserer Vorgängergruppe umgesetzt wurde haben wir keine weiteren Daten aussortiert. (Nachzulesen in [Bur23] in Kapitel 5.2.3)

Wir beginnen damit, dass wir den kompletten Datensatz einlesen und diesen dann in die benötigte Struktur bringen. Jede Audiodatei wird mithilfe der PyDup-Bibliothek AudioSegment eingelesen. Es wird die Sample-Rate ausgelesen, eine Extrahierung der MFCC-Merkmale durchgeführt sowie die Audiolänge auf genau 2 Sekunden verlängert und Hintergrundgeräusche des fahrenden Autos über die Audiodatei gelegt.

Verändern der Audiolänge

Alle Audiodateien sind deutlich kürzer als 2 Sekunden. Da beim späteren Einsatz auf dem Board auch immer Audiodateien mit der gleichen Länge reingegeben werden, sollen die Daten beim Trainieren ebenfalls die gleiche Länge haben. Damit der Befehl nicht immer am Anfang (oder Ende) der 2s-Audio ist, wird der Audiobefehl per Zufall an eine Stelle innerhalb der 2-Sekunden leeren Audiodatei gesetzt.

Hinzufügen von Hintergrundgeräuschen

Beim späteren Einsatz auf dem Board wird es nicht unerhebliche Geräusche vom Motor des Autos geben. Dies soll beim Trainieren bedacht werden, daher werden hinter die 2s-langen Audiodateien Hintergundgeräusche des fahrenden Autos gelegt.

Speicherung

Jedes der drei Audiosignale (Original, 2s-Länge, mit Hintergundgeräuschen) wird dann in ein PyTorch-Tensor-Format konvertiert. Anschließend wird ein AudioData-Objekt mit folgenden Attributen (Daten) erstellt:

- Abtastrate (*sample_rate*)
- Tensor-Objekt der Original-Audio (*waveform_raw*)
- Tensor-Objekt mit 2s Länge ohne Hintergrundgeräusche (*waveform_without_bg_noise*)
- Tensor-Objekt mit 2s Länge ohne Hintergrundgeräusche (*waveform_with_bg_noise*)
- Dateiname (*file_name*)

- Extrahierte MFCC-Merkmale (*features*)

Dabei werden nicht alle dieser Daten verwendet (aktuell sind z.B. *waveform_raw* und *file_name* obsolet), allerdings könnten Sie in der Zukunft ggfs. relevant werden, weshalb sie hier direkt mitgespeichert werden.

7.3 Training

geschrieben von Friederike Korte

In diesem Kapitel werden die zwei implementieren Ansätze zum Training eines Neuronalen Netzes beschrieben. Der erste Ansatz ist im Projekt implementiert und wird bei Aufrufen der main-Methode verwendet. Da dieser Trainingsansatz nicht erfolgreich verlief, wurde kurzfristig noch ein zweiter Ansatz getestet, der in dem Jupyter-Notebook `Neural_Network_Models` im Verzeichnis `ki` liegt. Dieser orientiert sich an einem Tutorial von PyTorch zum Trainieren eines Sprach-Modells. In diesem Ansatz wurden Fortschritte im Bezug zum ersten Ansatz erzielt, jedoch konnte auch hier kein Modell komplett trainiert werden.

7.3.1 Ansatz 1

geschrieben von Serkay-Günay Celik

Trainingsstrategie

Für das Training des neuronalen Netzes wird ein überwachtes Lernverfahren verwendet, das darauf abzielt, Sprachkommandos in verschiedene vordefinierte Kategorien einzuordnen. Die Modelle werden auf einer GPU trainiert, sofern eine CUDA-fähige Grafikkarte verfügbar ist, andernfalls erfolgt das Training auf der CPU. Das Training selbst umfasst die beiden Neuronalen Netze `Model2RNN` und `PreviousModel`, die in einer Schleife nacheinander getestet werden, um deren Leistung zu vergleichen und das bestmögliche Modell für die Klassifikationsaufgabe zu identifizieren.

Ein zentraler Bestandteil der Trainingsstrategie ist die effiziente Verarbeitung der Eingabedaten. Da diese in Form von Sequenzen vorliegen, die unterschiedliche Längen aufweisen, muss eine geeignete Methode zur Stapelverarbeitung implemenentiert werden. Hierzu wird eine Custom Collate Function genutzt, die sicherstellt, dass alle Eingaben innerhalb eines Mini-Batches auf die gleiche Länge gepadded werden. Dadurch kann das Modell die Daten effizient verarbeiten, ohne dass durch variierende Sequenzlängen unnötige Speicher- oder Rechenprobleme enstehen.

Neben der Datenverarbeitung ist auch die Wahl der richtigen Optimierungsstrategie von großer Bedeutung. Nach jedem Trainingsschritt wird der Verlust berechnet, woraufhin die Modellparameter mittels Backpropagation aktualisiert werden. Dieser iterative Prozess wird über mehrere Epochen hinweg durchgeführt, wobei nach jeder Epoche eine Evaluierung auf den Testdaten erfolgt. Dadurch kann überprüft werden, ob das Modell Fortschritte macht und wie gut es auf zuvor ungesehene Daten generalisiert.

Um die Effektivität des Trainingsprozesses weiter zu verbessern, wurden verschiedene Maßnahmen getroffen. Dazu gehört unter anderem das Shuffling der Trainingsdaten, um sicherzustellen, dass das Modell keine unerwünschten Muster aus der Reihenfolge der Daten lernt. Zudem wird mit einer moderaten Batch-Größe 32 gearbeitet, wodurch ein Gleichgewicht zwischen Stabilität und Geschwindigkeit des Trainings erreicht werden soll.

Optimierungsansatz

Zur Optimierung des neuronalen Netzes wurde als Verlustfunktion die CrossEntropyLoss verwendet. Diese eignet sich besonders gut für Mehrklassen-Klassifikationen, da sie nicht nur die Abweichung zwischen vorhergesagten und tatsächlichen Klassen berücksichtigt, sondern auch stark fehlerhafte Vorhersagen mit

höheren Fehlerwerten bestraft. Dadurch wird das Modell gezielt darauf trainiert, möglichst präzise und eindeutige Klassifikationen durchzuführen.

Für die Aktualisierung der Modellgewichte wurde der Adam-Optimizer eingesetzt. Adam kombiniert die Vorteile von zwei etablierten Optimierungsverfahren: dem klassischen Stochastic Gradient Descent (SGD) und dem RMSprop-Algorithmus. Durch adaptive Lernraten für jede einzelne Modellparameter-Aktualisierung sorgt Adam dafür, dass das Training effizient verläuft und schneller konvergiert als herkömmliche Optimierungsmethoden. Die Lernrate wird dabei auf 0.001 gesetzt, da sich dieser Wert in vielen neuronalen Netzen als guter Ausgangspunkt bewährt hat.

Ein weiterer Vorteil des Adam-Optimizers ist seine Fähigkeit, mit nicht-stationären Daten umzugehen, da er in der Lage ist, vergangene Gradienteninformationen zu speichern und darauf basierend adaptive Updates durchzuführen. Dies soll dazu führen, dass das Modell bei variierenden Trainingsmustern robuste Ereignisse liefert. Um sicherzustellen, dass das Training stabil verläuft und das Modell sich nicht zu stark an die Trainingsdaten anpasst (Overfitting), wird während des gesamten Prozesses regelmäßig die Validierungsgenauigkeit überwacht.

Datenverarbeitung & Batch-Handling

Ein wesentlicher Bestandteil des Trainigsprozesses ist die effiziente Verarbeitung der Eingabedaten, da die Sequenzen innerhalb des Datensatzes unterschiedliche Längen aufweisen. Um sicherzustellen, dass das Modell die Daten in Mini-Batches verarbeiten kann, wurde eine Custom Collate Function implementiert. Diese Funktion übernimmt das Padding der Sequenzen innerhalb eines Batches, indem alle Eingaben auf Länge der längsten Sequenz des jeweiligen Mini-Batches erweitert werden. Das Padding erfolgt mit Nullen, sodass die ursprünglich Informationen der Sequenz erhalten bleibt, während das Modell dennoch in der Lage ist, die Eingaben einheitlich zu verarbeiten.

Zusätzlich werden die Labels der Daten als String vorgegeben, weshalb sie vor dem Training in numerische Indizes umgewandelt werden. Diese Indizierung erleichtert die Verarbeitung durch das neuronale Netz und ermöglicht eine effiziente Berechnung des Loss-Werts. Um eine robuste und ausgewogene Trainingsverteilung zu gewährleisten, werden die Trainingsdaten vor jedem Durchlauf zufällig durchmischt, während die Testdaten in einer festen Reihenfolge bleiben.

Für die eigentliche Verarbeitung der Daten wird der PyTorch DataLoader verwendet, der die Mini-Batches organisiert und durch die Custom Collate Function sicherstellt, dass alle Eingaben in der richtigen Form vorliegen. Dabei wird eine Batch-Größe von 32 gewählt, um eine Balance zwischen Trainingsstabilität und Rechengeschwindigkeit zu finden. Während des Trainings erfolgt die Datenverarbeitung direkt auf der GPU, falls eine CUDA-fähige Grafikkarte verfügbar ist. Andernfalls wird die Berechnung auf der CPU durchgeführt. Diese dynamische Anpassung ermöglicht eine optimale Nutzung der verfügbaren Hardware-Ressourcen und sorgt für eine effiziente Modellaktualisierung.

Trainingsablauf

Das Training des neuronalen Netzes erfolgt über einen Zeitraum von 10 Epochen, in denen das Modell iterativ auf die Trainingsdaten angepasst wird. Jede Epoche besteht aus mehreren Trainingsschritten, bei denen das Modell aus den Eingabedaten lernt, seine Gewichte aktualisiert und anschließend auf den Testdaten validiert wird. Der Ablauf innerhalb einer Epoche gliedert sich in mehreren Phasen: den Vorwärtsdurchlauf (Forward Pass), die Berechnung des Verlusts (Loss Berechnung), die Backpropagation und den anschließenden Optimierungsschritt.

Im Vorwärtsdurchlauf werden die Eingabedaten durch das Modell geleitet, um Vorhersagen zu generieren. Anschließend wird der Verlustwert berechnet, indem die vorhergesagten Werte mit den tatsächlichen Labels verglichen werden. Dazu wird die CrossEntropyLoss-Funktion verwendet, die sich besonders für Mehrklassen-Klassifikation eignet, da sie hohe Fehlerwerte für falsche Vorhersagen stärker bestraft.

Nach der Verlustberechnung folgt die Backpropagation, ein zentrales Element des Trainingsprozesses. Dabei werden die Gradienten der Gewichte berechnet, um herauszufinden, in welche Richtung die

Modellparameter aktualisiert werden müssen, um den Fehler zu minimieren. Mithilfe des Adam-Optimizers wurden die Gewichte dann schrittweise angepasst.

Nach jedem vollständigen Durchlauf über die Trainingsdaten wird eine Evaluierung auf den Testdaten durchgeführt, um zu überprüfen, wie gut das Modell auf bisher ungesehene Daten generalisiert. Dazu wird das Modell in den Evaluierungsmodus versetzt, sodass keine Gradienten berechnet oder Modellgewichte verändert werden. Die Testdaten werden durch das trainierte Netz geleitet, und anhand der vorhergesagten Klassen wird die Validierungsgenauigkeit bestimmt. Zusätzlich wird berechnet, wie viele der Vorhersagen korrekt sind, um eine erste Einschätzung über die Modellleistung zu erhalten.

Während des gesamten Trainingsprozesses wird der Verlustwert kontinuierlich überwacht, um mögliche Probleme wie Overfitting frühzeitig zu erkennen. Bei Bedarf können durch nachträgliche Anpassungen von Hyperparametern wie Lernrate oder Batch-Größe weitere Optimierungen vorgenommen werden, um die Leistungsfähigkeit des Modells weiter zu verbessern.

Visualisierung der Modellvorhersagen & Konfusionsmatrix

Nach dem Training des Modells soll dessen Leistung anhand verschiedener Visualisierungsmethoden analysiert werden können. Dabei kommen zwei zentrale Ansätze zum Einsatz: die Visualisierung einzelner Modellvorhersagen und die Erstellung einer Konfusionsmatrix. Diese Methoden ermöglichen eine detaillierte Bewertung der Klassifikationsgüte und helfen dabei, potenzielle Schwächen des Modells zu identifizieren.

Visualisierung der Modellvorhersagen

Um die Entscheidungsfindung des Modells besser nachvollziehen zu können, werden einzelne Testbeispiele herangezogen und deren Klassifikationsausgabe detailliert analysiert. Hier wird die Wahrscheinlichkeitsverteilung über alle möglichen Klassen hinweg als Balkendiagramm dargestellt. Die Softmax-aktivierten Ausgangswerte des Modells zeigen für jedes Beispiel an, mit welcher Wahrscheinlichkeit das Modell eine bestimmte Klasse zugeordnet hat.

Diese Visualisierung ermöglicht es, nicht nur richtige Klassifikationen zu bestätigen, sondern auch zu untersuchen, in welchen Fällen das Modell unsicher ist oder falsche Vorhersagen trifft. Wenn das Modell beispielsweise mit hoher Wahrscheinlichkeit eine falsche Klasse vorhersagt, kann dies auf problematische Muster in den Trainingsdaten oder auf eine unzureichende Modellgeneralisierung hindeuten.

Erstellung und Analyse der Konfusionsmatrix

Zur Bewertung der Gesamtleistung des Modells wird eine Konfusionsmatrix erstellt. Diese zeigt auf, wie oft jede Klasse korrekt vorhergesagt wird und welche Klassen besonders häufig miteinander verwechselt werden. Die Matrix wird mit der `ConfusionMatrixDisplay`-Funktion aus `sklearn.metrics` visualisiert und als Heatmap dargestellt, wobei die Zeilen die tatsächlichen Klassen und die Spalten die vorhergesagten Klassen repräsentieren.

Die Analyse der Konfusionsmatrix liefert wertvolle Erkenntnisse darüber, welche Kategorien vom Modell besonders gut erkannt werden und wo häufige Fehler auftreten. So können systematische Verwechslungen identifiziert werden, die z.B. darauf hindeuten, dass sich bestimmte Klassen in ihren Eigenschaften stark ähneln. Falls einige Klassen signifikant schlechter abschneiden, könnte dies auf ein Ungleichgewicht in den Trainingsdaten oder eine unzureichende Repräsentation bestimmter Muster im Trainingsprozess zurückzuführen sein.

Durch die Kombination dieser beiden Visualisierungsmethoden ist es möglich, nicht nur die Modellgenauigkeit quantitativ zu bewerten, sondern auch qualitative Einblicke in das Entscheidungsverhalten des neuronalen Netzes zu gewinnen. Diese Informationen können als Grundlage für zukünftige Optimierungsmaßnahmen dienen, beispielsweise durch gezieltes Nachtrainieren des Modells mit schwierigeren Beispielen oder durch Anpassungen an der Modellarchitektur.

Herausforderung und fehlgeschlagene Trainingsversuche

Während des Trainings treten wiederholt Probleme mit der Dimensionierung der Matrizen im neuronalen Netz auf, die dazu führen, dass das Modell nicht erfolgreich trainiert werden kann. `RuntimeError: mat1 and mat2 shapes cannot be multiplied (32x46260 and 1990x3980)`

Dieser Fehler deutet darauf hin, dass die Matrizenmultiplikation zwischen zwei Schichten des neuronalen Netzes nicht möglich ist, da die Dimensionen der Eingabedaten und der Gewichtsmatrizen nicht kompatibel sind. In unserem Fall hat die Eingabematrix die Form 32x46260, während die erwartet Gewichtsmatrix die Form 1990x3980 aufweist.

Mögliche Ursachen für diesen Fehler sind eine fehlerhafte Architektur des neuronalen Netzes oder eine inkonsistente Datenverarbeitung. Trotz intensiver Analyse und Anpassungen an der Datenverarbeitung konnte bisher keine Lösung des Problems gefunden werden. Dies führt dazu, dass diese Trainingsmethode nicht erfolgreich eingesetzt werden kann.

Dieses Problem verdeutlicht die Herausforderungen bei der Arbeit mit neuronalen Netzen. Eine weitere Analyse der Datenverarbeitung und der Schichtenarchitektur ist erforderlich, um das Problem zu lösen und ein funktionierendes Modell zu trainieren.

7.3.2 Ansatz 2

geschrieben von Friederike Korte

Da der erste Trainingsansatz leider nicht erfolgreich verlief, wurde kurzfristig noch ein weiterer Ansatz getestet. Dieser befindet sich im Jupyter-Notebook *Neural Network Models*, das im GitLab im Verzeichnis wise2425/ki liegt. Grundlage für diesen Ansatz ist das Beispiel *Speech Command Classification with torchaudio* von PyTorch. [PyTc]

Die Audiodaten werden hier genau wie beim ersten Ansatz eingelesen und in einen 80/20-Train/Test-Split aufgeteilt. Danach unterscheidet sich das Vorgehen jedoch.

In diesem Ansatz werden die Tensor-Objekte auf die gleiche Abtastrate resampled und die Basis von int16 zu float32 transfomiert. Grund hierfür ist eine bessere Kompatibilität mit Neuronalen-Netz-Modellen, die oftmals einen normalisierten Float-Wert erwarten.

Bevor mit dem Trainieren gestartet werden kann, werden noch einige Hilfsfunktionen und Variablen sowie das Model definiert:

pad_sequence()

In dieser Hilfsfunktion werden alle Tensor-Objekte aus einem Batch auf die gleiche Länge gebracht. Dabei werden kürzere Tensor-Objekte mit Nullen aufgefüllt, bis sie die Länge des längsten Tensor-Objektes erreicht haben.

collate_fn()

geschrieben von Sebastian Schramm

Die Funktion **collate_fn()** wird verwendet, um die Daten in einem Batch so zu verarbeiten, dass alle Tensoren die gleiche Länge haben. Dies ist notwendig, da die Eingabedaten, wie zum Beispiel Audio Wellenformen, unterschiedliche Längen haben können. Die Funktion nimmt als Eingabeparameter **batch**, eine Liste von Tupeln, wobei jedes Tupel aus einem Label und einem Tensor besteht. Zunächst werden die Labels und Tensoren aus dem Batch extrahiert und in separate Listen **labels** und **tensors** gespeichert. Die Labels, die als Strings vorliegen, werden in numerische Indizes umgewandelt, indem die Position des Labels in der Liste der Kategorien gesucht wird. Anschließend wird die maximale Länge der Tensoren im Batch ermittelt, um die Tensoren auf diese Länge zu bringen. Die Tensoren werden dann mit Nullen

aufgefüllt, sodass alle Tensoren im Batch die gleiche Länge haben. Schließlich werden die gepaddeten Tensoren und die numerischen Labels als Tupel zurückgegeben. Dadurch stellt diese Funktion sicher, dass alle Tensoren im Batch die gleiche Länge haben, was für die Verarbeitung in neuronalen Netzen notwendig ist.

batch_size

Dieser Parameter gibt die Anzahl der Trainingsbeispiele an, die in einem einzelnen Durchlauf durch das neuronale Netzwerk verarbeitet werden. Eine größere **batch_size** kann die Trainingszeit pro Epoche verkürzen, da mehr Daten parallel verarbeitet werden, erfordert jedoch mehr Speicher. Eine kleinere **batch_size** kann zu stabileren Gradienten führen, da die Gewichte häufiger aktualisiert werden, benötigt jedoch mehr Epochen, um das gleiche Maß an Konvergenz zu erreichen.

device

Mittels des **device** kann bestimmt werden, auf welcher Hardware das neuronale Netzwerk trainiert werden soll. In der Regel kann dies entweder die CPU oder GPU sein. Wenn eine GPU verfügbar ist, wird diese bevorzugt, da sie die Berechnungen erheblich beschleunigen kann. In diesem Fall wird überprüft, ob eine GPU verfügbar ist, welche **cuda** unterstützt, andernfalls wird die CPU verwendet.

train_loader und test_loader

Der **train_loader** und **test_loader** sind DataLoader-Objekte, die verwendet werden, um die Trainings- und Testdaten in Batches zu laden und zu verarbeiten. Der **train_loader** lädt die Trainingsdaten in zufälliger Reihenfolge, um das Modell während des Trainings zu optimieren, während der **test_loader** die Testdaten in gleicher Reihenfolge lädt, um die Leistung des Modells zu bewerten. Beide DataLoader verwenden die **collate_fn()**-Funktion, um die Tensoren in Batches gleicher Länge zu bringen. Die Batchgröße, die Anzahl der parallelen Arbeitsprozesse und die Speicherzuweisung werden basierend auf dem verwendeten Gerät konfiguriert.

model

mit geschrieben von Friederike Korte

Das **model** ist ein neuronales Netzwerk, das für die Klassifikation von Sprachbefehlen verwendet wird. Es ist exakt aus dem Tutorial von PyTorch übernommen worden. Es besteht aus mehreren Schichten, darunter Faltungsschichten, Batch-Normalisierungsschichten und Pooling-Schichten. Diese Schichten extrahieren Merkmale aus den Eingabedaten und reduzieren deren Dimensionen, um die wichtigsten Informationen zu behalten. Am Ende des Netzwerks befindet sich eine voll verbundene Schicht, die die extrahierten Merkmale in Klassenvorhersagen umwandelt.

Bei diesem Modell handelt es sich weder um ein sehr tiefes (insgesamt vier Convolutional Layer) noch ein sehr komplexes Neuronales Netz. Es eignet sich für eher weniger komplexe Audio-Klassifikationsaufgaben und wurde daher auch exakt übernommen.

optimizer und scheduler

Der **optimizer** ist ein Optimierungsalgorithmus, der verwendet wird, um die Gewichte des neuronalen Netzwerks während des Trainings anzupassen. In diesem Ansatz wird der Adam Optimizer verwendet, der für seine Effizienz und gute Konvergenzeigenschaften bekannt ist. Der **scheduler** ist ein Lernratenplaner, der die Lernrate des Optimizers nach einer bestimmten Anzahl von Epochen reduziert, um die Konvergenz zu verbessern und Überanpassung zu vermeiden.

train und test

Die **train**-Funktion führt das Training des Modells durch. Sie iteriert über die Trainingsdaten, berechnet die Vorhersagen des Modells, den Verlust und aktualisiert die Gewichte des Modells basierend auf dem Gradientenabstieg. Die **test**-Funktion bewertet die Leistung des Modells auf den Testdaten, indem sie die Genauigkeit der Vorhersagen berechnet. Beide Funktionen verwenden den DataLoader, um die Daten in Batches zu laden und die **collate_fn()**-Funktion, um die Tensoren auf die gleiche Länge zu bringen.

predict

Die **predict**-Funktion verwendet das trainierte Modell, um die Klasse eines gegebenen Audio-Samples vorherzusagen. Sie transformiert das Audio-Sample, führt es durch das Modell und gibt die vorhergesagte Klasse zurück. Diese Funktion ist nützlich, um die Leistung des Modells auf neuen, ungesehenen Daten zu testen.

Evaluation

Nach dem Training wird das Modell auf einem separaten Validierungsdatensatz evaluiert. Dabei werden Metriken wie Genauigkeit, Präzision, Recall und F1-Score berechnet, um die Leistung des Modells zu bewerten. Diese Metriken geben Aufschluss darüber, wie gut das Modell in der Lage ist, die verschiedenen Sprachbefehle korrekt zu klassifizieren.

Hyperparameter-Tuning

Um die Leistung des Modells weiter zu optimieren, wird ein Hyperparameter-Tuning durchgeführt. Dabei werden verschiedene Kombinationen von Hyperparametern wie Lernrate, Batchgröße und Anzahl der Schichten getestet, um die besten Einstellungen für das Modell zu finden. Dies geschieht in der Regel durch eine Grid- oder Random-Suche.

Modell-Speicherung

Nach dem erfolgreichen Training und der Evaluation wird das Modell gespeichert, um es später wiederverwenden zu können. Das Modell wird in einem ONNX-Format gespeichert, das eine plattformübergreifende Nutzung ermöglicht. Dies erleichtert die Integration des Modells in verschiedene Anwendungen und Systeme.

Herausforderungen und fehlgeschlagene Trainingsversuche

geschrieben von Sebastian Schramm

mit geschrieben von Friederike Korte

Bei den ersten Trainingsversuchen auf der CPU ist der Rechner schnell an seine Grenzen gekommen. Nach nur wenigen Minuten Trainingszeit ist der Fehler `DefaultCPUAllocator: not enough memory` aufgetreten. Daher haben wir die `size` von urspünglich 256 auf 16 reduziert und zusätzlich auf einem Rechner mit GPU trainiert (Training ist hier in der Regel deutlich schneller). Trotz der Vorteile und der Kompilierung auf der GPU stieß der Trainingsprozess auf erhebliche Herausforderungen. Nach einer Laufzeit von 1 1/2 bis 2 Stunden zeigte sich, dass das Modell nicht weiterkam. Dies könnte auf verschiedene Faktoren zurückzuführen sein, wie z.B. unzureichende Datenqualität, suboptimale Hyperparameter oder Modellarchitektur. Es ist wichtig, diese Aspekte zu analysieren und gegebenenfalls Anpassungen vorzunehmen, um die Trainingsleistung zu verbessern. In diesem Fall war es jedoch nicht möglich, das Problem zu lösen, und das Training musste abgebrochen werden.

7.3.3 Training mit einer AMD GPU

geschrieben von Sebastian Schramm

PyTorch ist ein beliebtes Framework für das Training von neuronalen Netzen. Es bietet eine Vielzahl von Funktionen und ist sowohl für CPUs als auch für GPUs optimiert. Allerdings ist die Installation von PyTorch für AMD GPUs nicht so einfach wie für NVIDIA GPUs, da PyTorch standardmäßig auf CUDA basiert, das nur von NVIDIA-GPUs unterstützt wird. Für AMD GPUs wird stattdessen **ROCm** (Radeon Open Compute) verwendet, das eine ähnliche Funktionalität wie CUDA bietet. In diesem Kapitel wird beschrieben, wie PyTorch mit einer AMD GPU unter Verwendung von ROCm installiert und konfiguriert werden kann.

Systemvoraussetzungen

In diesem Kapitel wird **Ubuntu 24.04 LTS** als Betriebssystem sowie **Python 3.10** verwendet. Die Installation von PyTorch und ROCm erfolgt über **pip**. Es wird vorausgesetzt, dass die AMD GPU bereits korrekt installiert und erkannt wurde. Die notwendigen Treiber werden im Installationsprozess erwähnt. Laut der offiziellen ROCm-Dokumentation wird die Installation von ROCm im Consumer-Bereich aktuell nur mit der RX 7900 Serie unterstützt [ROCb]. In diesem Fall wird eine **AMD Radeon RX 7900 XTX** verwendet.

Einrichtung des Systems

Um ein neuronales Netz mit **PyTorch** unter Verwendung einer **AMD GPU** zu trainieren, muss **ROCm** (Radeon Open Compute) installiert werden. Die folgende Anleitung beschreibt die notwendigen Schritte zur Einrichtung des Systems.

Installation von ROCm Die Installation von ROCm umfasst mehrere Schritte, die sicherstellen, dass alle benötigten Komponenten korrekt eingerichtet sind [ROCa].

1. **System aktualisieren:** Bevor neue Software installiert wird, sollte das System auf den neuesten Stand gebracht werden.

```
sudo apt update
```

2. **Erforderliche Kernel-Module installieren:** Diese Module sind notwendig, um die AMD GPU korrekt zu nutzen.

```
sudo apt install "linux-headers-\$(uname -r)" "linux-modules-extra-\$(uname -
    r)"
```

3. **Python-Pakete installieren:** PyTorch benötigt bestimmte Python-Pakete zur Ausführung.

```
sudo apt install python3-setuptools python3-wheel
```

4. **Benutzer zu den erforderlichen Gruppen hinzufügen:** Damit der Benutzer auf die GPU zugreifen kann, muss er in die Gruppen **render** und **video** aufgenommen werden.

```
sudo usermod -a -G render,video \$LOGNAME
```

5. **AMD GPU Treiber herunterladen und installieren:** Der AMD GPU-Treiber muss manuell heruntergeladen und installiert werden.

```
wget https://repo.radeon.com/amdgpu-install/6.3.2/ubuntu/noble/amdgpu-
    install_6.3.60302-1_all.deb
sudo apt install ./amdgpu-install_6.3.60302-1_all.deb
```

6. **System aktualisieren:** Nach der Installation des Treibers sollte erneut ein Update durchgeführt werden.

```
sudo apt update
```

7. **ROCm installieren:** Nun kann ROCm installiert werden.

```
sudo apt install amdgpu-dkms rocm
```

8. **System neustarten:** Nach der Installation aller Komponenten ist ein Neustart erforderlich, damit die Änderungen wirksam werden.

```
sudo reboot
```

Installation der benötigten Python-Pakete

Nach der erfolgreichen Einrichtung von ROCm müssen die benötigten PyTorch-Pakete installiert werden, um die GPU für das Training nutzen zu können. Dazu wird die neueste ROCm-Version von PyTorch aus dem offiziellen Repository installiert [PyTb]:

```
pip3 install --pre torch torchvision torchaudio --index-url https://download.
    pytorch.org/whl/nightly/rocm6.2/
```

Dieser Befehl stellt sicher, dass die korrekten Versionen der PyTorch-Bibliotheken für ROCm installiert werden.

Überprüfung der Installation

Um zu testen, ob die Installation erfolgreich war, kann ein einfaches Python-Skript verwendet werden, das prüft, ob die GPU erkannt wird. Dazu wird folgender Code verwendet:

```
import torch
print(torch.cuda.is_available())
```

Falls `True` ausgegeben wird, ist CUDA bzw. ROCm aktiv.

Überprüfung, ob die GPU erkannt wurde

Zusätzlich kann getestet werden, ob die GPU korrekt erkannt wurde:

```
if torch.cuda.is_available():
    print(f"CUDA Device: {torch.cuda.get_device_name(0)}")
elif torch.version.hip:
    print(f"ROCm Device: {torch.cuda.get_device_name(0)}")
else:
    print("Keine GPU erkannt")
```

Falls eine GPU erkannt wird, sollte deren Name in der Ausgabe erscheinen. Andernfalls muss die Installation nochmals überprüft werden.

7.4 Quantisieren des NN

geschrieben von Sebastian Schramm

Die Quantisierung von neuronalen Netzen ist ein Verfahren, bei dem die Präzision der Modellparameter und -operationen reduziert wird, um die Effizienz und Leistung zu verbessern. Dies ist besonders nützlich für die Implementierung auf Hardware mit begrenzten Ressourcen, wie z.B. FPGAs oder mobile Geräte. Durch die Reduzierung der Bit Breite der Gewichte und Aktivierungen können Speicherbedarf und Rechenleistung erheblich verringert werden, ohne die Genauigkeit des Modells wesentlich zu beeinträchtigen.

7.4.1 Vorteile der Quantisierung

1. Reduzierten Speicherbedarf: Durch die Verwendung von niedrigeren Bit-Breiten (z.B. 8-Bit anstelle von 32-Bit) wird der Speicherbedarf für die Modellparameter reduziert. Dies ist besonders vorteilhaft für eingebettete Systeme und mobile Geräte, die oft über begrenzten Speicher verfügen.
2. Schnellere Berechnungen: Quantisierte Modelle können schneller ausgeführt werden, da die Berechnungen mit niedrigeren Bit-Breiten effizienter sind. Dies führt zu einer schnelleren Inferenzzeit, was besonders in Echtzeitanwendungen wichtig ist.
3. Geringerer Energieverbrauch: Besonders wichtig für mobile und eingebettete Systeme, da weniger Energie für die Berechnungen benötigt wird. Dies verlängert die Batterielaufzeit und reduziert die Wärmeentwicklung.
4. Einfache Implementierung auf Hardware: Quantisierte Modelle sind besser für die Implementierung auf spezialisierten Hardwareplattformen wie FPGAs geeignet. Diese Hardware kann oft effizienter mit niedrigeren Bit-Breiten arbeiten.

7.4.2 Exportieren des Neuronalen Netzes

Um ein quantisiertes neuronales Netz nach ONNX zu exportieren, kann der folgende Ansatz verwendet werden:

1. Modell in den Evaluierungsmodus versetzen: Das Modell wird in den Evaluierungsmodus versetzt, um sicherzustellen, dass keine Gradienten berechnet werden.
2. Dummy-Input erstellen: Ein Dummy-Input-Tensor mit der gleichen Größe wie der Modelleingang wird erstellt.
3. Modell nach ONNX exportieren: Das Modell wird nach ONNX exportiert, wobei die Modellparameter, die ONNX-Operatoren-Version und die Eingabe- und Ausgabeknoten definiert werden.

Dieser Prozess ermöglicht die Verwendung des quantisierten Modells auf verschiedenen Plattformen und Hardwarebeschleunigern [PyTa] [ONN].

Kapitel 8

Transfer des NN auf das FPGA

geschrieben von Marvin Hoyer

In diesem Kapitel wird zunächst auf die Installation des FINN-Frameworks selbst sowie die dafür notwendigen Anwendungen genauer eingegangen. Anschließend ist aufgeführt, wie ein quantisiertes neuronales Netz in Form einer ONNX-Datei analysiert werden kann, und wie eine erste Abschätzung der Laufzeiten des Models oder notwendiger Ressourcen für ein ausgewähltes FPGA-Board mittels FINN generiert werden kann. Außerdem wird die Erstellung eines IP-Blocks aus diesem neuronalen Netz mittels FINN genauer erläutert und die Einbindung des daraus entstehenden Blocks in die Gesamtstruktur genauer beschrieben.

8.1 Einrichtung des FINN Frameworks

geschrieben von Marvin Hoyer

Dieser Abschnitt beschreibt die relevanten Schritte, um den FINN-Compiler im Anschluss an diesen Abschnitt auf einem Computer ausführen zu können. Zu diesen Schritten zählen das Vornehmen verschiedener Einstellungen sowie die Installation notwendiger Programme. Es werden außerdem während des Einrichtungsprozesses aufgetretene Probleme erwähnt und die in diesem Projekt dafür angewendeten Lösungsschritte vorgestellt.

Sollten sich während der Einrichtung oder Installation andere, in diesem Projekt nicht aufgetretene Probleme ergeben, wird empfohlen, sich auf die verwiesenen offiziellen Installationsanleitungen zu beziehen. Diese enthalten weitere Informationen zu den einzelnen Installationsschritten und stellen i. d. R. einige Hinweise zur Behebung häufig auftretender Fehler bereit.

8.1.1 Errichten eines nutzbaren Linux-Systems

geschrieben von Thomas Klein

Dieser Abschnitt beschreibt die Erstellung eines bootfähigen Geräts, um auf diesem eine Linux-Distribution installieren und ausführen zu können. Sollte bereits ein Linux-Gerät vorhanden sein, ist es für die Nutzung des FINN-Compilers notwendig, dass das root-Passwort bekannt ist. Außerdem sollte das vorhandene Linux-System die Installationsanforderungen der AMD Vivado Design Suite [siehe AMD24f], von Docker [siehe Docb] und des FINN-Compilers [siehe FIN24] erfüllen. Wenn all dies zutrifft, kann mit dem Schritt in Abschnitt 8.1.2 fortgefahren werden.

Im Falle dieses Projektes wurde das Linux-Betriebssystem in einem Dual-Boot-Kontext neben einem Windows Betriebssystem installiert. Sollte Linux als einziges Betriebssystem auf dem zu nutzenden Computer installiert werden kann der folgende Absatz übersprungen werden.

Bevor mit der Einrichtung eines nutzbaren Linux-Systems begonnen wird sei an dieser Stelle darauf

hingewiesen, dass in diesem Fall für Windows der Bitlocker-Schlüssel bekannt sein muss. Sollte nach Abschluss dieses Schrittes das Linux-System gestartet werden und im Anschluss wieder zu Windows zurückgekehrt werden, kann es vorkommen, dass Windows nach dem Bitlocker-Schlüssel fragt. Unter Umständen ist dieser mit dem Microsoft-Account des verwendeten Geräts verlinkt und kann daher unter [Mica] gefunden werden, andernfalls sind weitere Möglichkeiten zum Auffinden des Schlüssels unter [Micb] gelistet. Sollte Bitlocker ein Problem darstellen kann es für den Zeitraum des Projektes auf eigene Gefahr hin deaktiviert werden. Auf Windows 11 muss dazu in den Einstellungen unter „Datenschutz und Sicherheit“ > „Geräteverschlüsselung“ die Geräteverschlüsselung auf „Aus“ gestellt werden. Sollte die Gerätesicherung aktiviert sein, kann das Deaktivieren je nach Größe und Anzahl der Dateien auf der Festplatte mehrere Stunden dauern.

Für das Aufsetzen eines Linux-Systems wird zunächst ein genügend großes Speichermedium wie z.B. eine (portable) SSD oder ein USB-Stick benötigt. Aus Erfahrungswerten des Projekts ist dafür eine Speichergröße von mindestens 256 GB ausreichend. Dieses Medium muss nun bootfähig gemacht werden. Dazu wird empfohlen, ein neues bzw. unbenutztes Speichermedium zu benutzen, da bei der Erstellung des bootfähigen Geräts alle Daten auf diesem Medium vollständig gelöscht werden.

Es wird empfohlen den Compiler auf einer Ubuntu-Linux-Distribution auszuführen. In diesem Projekt wird Linux Mint dafür verwendet. Hierbei handelt es sich wie in [Lina] beschrieben um eine von Ubuntu abgeleitete Distribution. Dazu wird die Downloadseite von Linux Mint unter [Linb] aufgerufen und dort die „Cinnamon-Edition“ der Version 22.1 ausgewählt. Auf der nächsten Seite befinden sich verschiedene Mirror-Images dieser Version, von denen eine beliebige ausgewählt werden kann. Hier wird die Version der „Hochschule Esslingen University of Applied Sciences“ verwendet. Durch einen Klick auf den Namen des Anbieters wird die ISO-Datei „linuxmint-22-cinnamon-64bit.iso“ heruntergeladen.

Nach erfolgreichem Download der ISO-Datei kann das Speichermedium bootfähig gemacht werden. Dafür können verschiedene Anwendungen genutzt werden, hier wird sich für „Rufus“ entschieden. Dazu wird die Anwendung über die offizielle Seite unter [Ruf] heruntergeladen und als Version die aktuelle portable Version für Windows, in diesem Fall „rufus-4.6p.exe“ gewählt. Die heruntergeladene Anwendungsdatei kann nun per Doppelklick gestartet werden, wobei sich ein Fenster mit verschiedenen Einstellungsmöglichkeiten öffnet. Zunächst muss ein Laufwerk ausgewählt werden, welches bootfähig gemacht werden soll (Nr. 1). Als nächstes muss unter „Startart“ das ISO-Linux-Image ausgewählt werden. Standardmäßig ist hier bereits „Laufwerk oder ISO-Image (Bitte wählen Sie ein Image)“ ausgewählt (Nr. 2). Rechts daneben befindet sich ein Button „Auswahl“ (Nr. 3), mit welchem die heruntergeladene ISO-Datei ausgewählt werden kann. Dazu wird im sich nach Klick auf den Button öffnenden Dateiexplorer zum Speicherort der ISO-Datei navigiert und diese ausgewählt. Anschließend kann der Prozess über den Button „START“ gestartet werden (Nr. 4). Zum Vergleich sind die vorzunehmenden Einstellungen in Abbildung 8.2 noch einmal abgebildet. Nach erfolgreicher Installation kann Rufus geschlossen und der Computer neu gestartet werden.

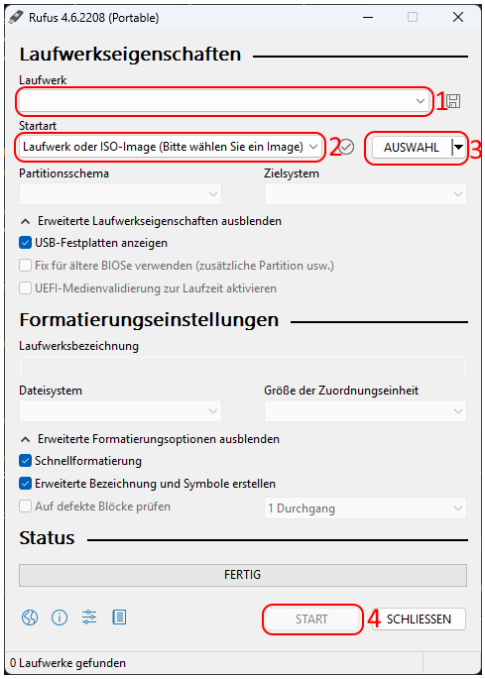


Abbildung 8.1: Benutzeroberfläche der Rufus-Anwendung

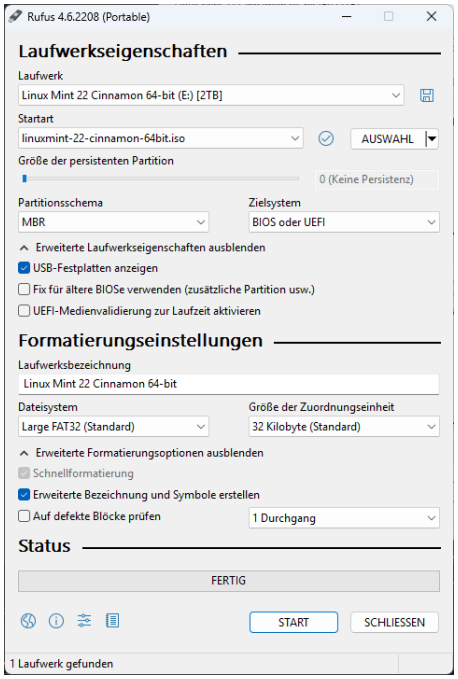


Abbildung 8.2: In Rufus vorzunehmende Einstellungen

Um nun das Linux Mint-System zu starten, muss das Boot-Menü des Geräts aufgerufen werden. Je nach verwendetem Gerät ist die Herangehensweise unterschiedlich und kann meistens auf der jeweiligen Herstellerseite nachgeschaut werden. Darauf öffnet sich das Boot-Menü und das Gerät mit dem Linux Betriebssystem sollte angezeigt werden. Nach Auswahl dieses startet sich das Installationssystem. Von hier aus kann nun das Linux-System installiert werden, worauf in diesem Bericht aber nicht weiter eingegangen wird. Weitere Hinweise zur konkreten Installation von Linux Mint können der Dokumentation des Betriebssystems unter [Linc] entnommen werden.

8.1.2 Vorbereitungen zur Installation der AMD Vivado Design Suite

geschrieben von Simon Sajnog

Um nun den Installer für die AMD Vivado Design Suite herunterladen zu können wird zunächst ein AMD-Konto benötigt. Dazu kann auf der offiziellen Downloadseite von AMD unter [AMDb] oben rechts in der Webseiten-Kopfzeile auf das Person-Icon geklickt und anschließend „My Account“ ausgewählt werden. Sollte bereits ein Konto bestehen, kann sich in dieses durch Eingeben der Zugangsdaten eingeloggt werden, andernfalls muss ein neues Konto erstellt werden.

Nach erfolgreicher Authentifizierung kann nun die Installationsdatei heruntergeladen werden. Dazu muss in den angezeigten Reitern der Reiter „Vivado (HW Developer)“ ausgewählt werden. Danach kann links auf der Webseite unter „Version“ eine gewünschte Version ausgewählt werden, wobei sich hier für die zum Zeitpunkt des Projekts aktuellste Version „2024.2“ entschieden wird. Im Anschluss kann die gewünschte Installationsdatei entsprechend dem verwendeten Betriebssystem ausgewählt und heruntergeladen werden. Hier muss darauf geachtet werden, dass es sich bei der Installationsdatei um die „Full Product Installation“ handelt. Da als Betriebssystem Linux verwendet wird, wird der „Linux Self Extracting Web Installer“ ausgewählt. Durch einen Klick auf den Namen der Installationsdatei wird diese heruntergeladen und kann an einem beliebigen Ort gespeichert werden. Hier ist es die BIN-Datei „FPGAs_AdaptiveSoCs_Unified_2024.2_1113_1001 _Lin64.bin“, welche im Verzeichnis „Downloads“ des aktuellen Benutzers gespeichert wird.

Bevor diese Datei nun zur Installation gestartet werden kann, müssen zusätzlich einige Anpassungen vorgenommen werden. Für die Installation der Vivado Design Suite 2024.2 wird die Bibliothek „libtinfo5“

benötigt, andernfalls schlägt die Installation fehl. Um „libtinfo5“ erfolgreich installieren zu können, sind nun die folgenden Schritte notwendig.

Zunächst muss die source-List für das Advanced Packaging Tool (apt) angepasst werden, da libtinfo5 in der im Projekt verwendeten Linux-Version nicht mehr in den Standard-Paketquellen vorhanden ist. Dazu muss der folgende Befehl ausgeführt werden:

```
$ sudo nano /etc/apt/sources.list
```

Anschließend muss dort folgende Zeile eingetragen werden:

```
$ deb http://archive.ubuntu.com/ubuntu/ lunar universe
```

Diese Änderungen müssen nun mittels `STRG + x`, anschließend `y` und `ENTER` gespeichert werden. Nun kann apt mittels des folgenden Befehls aktualisiert werden:

```
$ sudo apt update
```

Im Anschluss daran kann „libtinfo5“ mittels des folgenden Befehls installiert werden:

```
$ sudo apt install libtinfo5
```

8.1.3 Installationsprozess der AMD Vivado Design Suite

geschrieben von Simon Sajnog

Nun kann der AMD Vivado Design Suite Installer ausgeführt werden. Dazu muss im Terminal zunächst zu der heruntergeladenen BIN-Datei Datei navigiert werden. Anschließend muss die Datei ausführbar gemacht werden. Der Befehl dazu lautet:

```
$ chmod +x FPGAs_AdaptiveSoCs_Unified_2024.2_1113_1001_Lin64.bin
```

Nun kann der AMD Vivado Design Suite Installer mit folgendem Befehl gestartet werden:

```
$ ./FPGAs_AdaptiveSoCs_Unified_2024.2_1113_1001_Lin64.bin
```

Nach Ausführung dieses Befehls öffnet sich der Installer. Im sich öffnenden Willkommen-Fenster werden verschiedene Hinweise angezeigt. Durch einen Klick auf den Button „Next >“ erscheint eine neue Seite. Hier müssen im ersten Schritt die Zugangsdaten des AMD-Kontos eingegeben werden. Auch kann nun ausgewählt werden, ob der Download und die Installation jetzt stattfinden sollen („Download and Install Now“) oder zunächst nur das Image heruntergeladen werden soll („Download Image (Install Separately)“). An dieser Stelle wird die zweite Option empfohlen, da bei einer fehlschlagenden Installation ansonsten die Installationsdateien erneut heruntergeladen werden müssten. Dafür muss nun ein Verzeichnis ausgewählt werden, in welchem die Installationsdateien gespeichert werden sollen. Wichtig dabei ist, dass das ausgewählte Verzeichnis keine Leerzeichen enthalten darf. Anschließend muss „Linux“ als Plattform und als Image Contents „Selected Product Only“ ausgewählt werden.

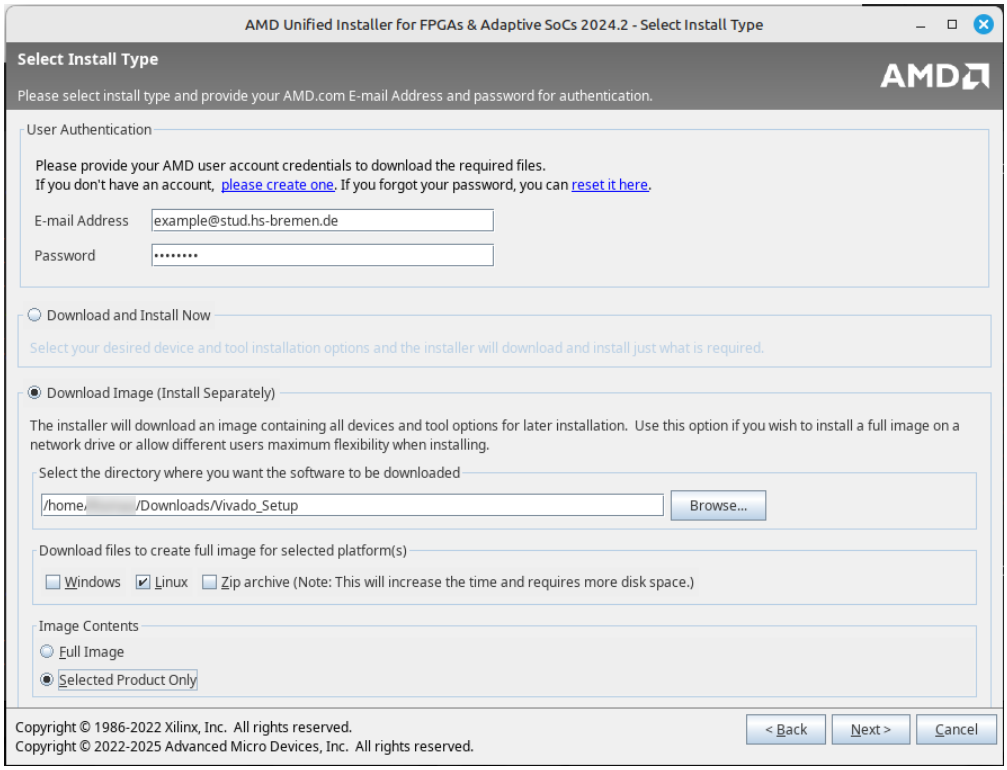


Abbildung 8.3: Auswahl des Vivado Installationstyps

Im nächsten Schritt muss wie in Abbildung 8.4 zu sehen zunächst Vivado als das zu installierende Produkt ausgewählt werden.

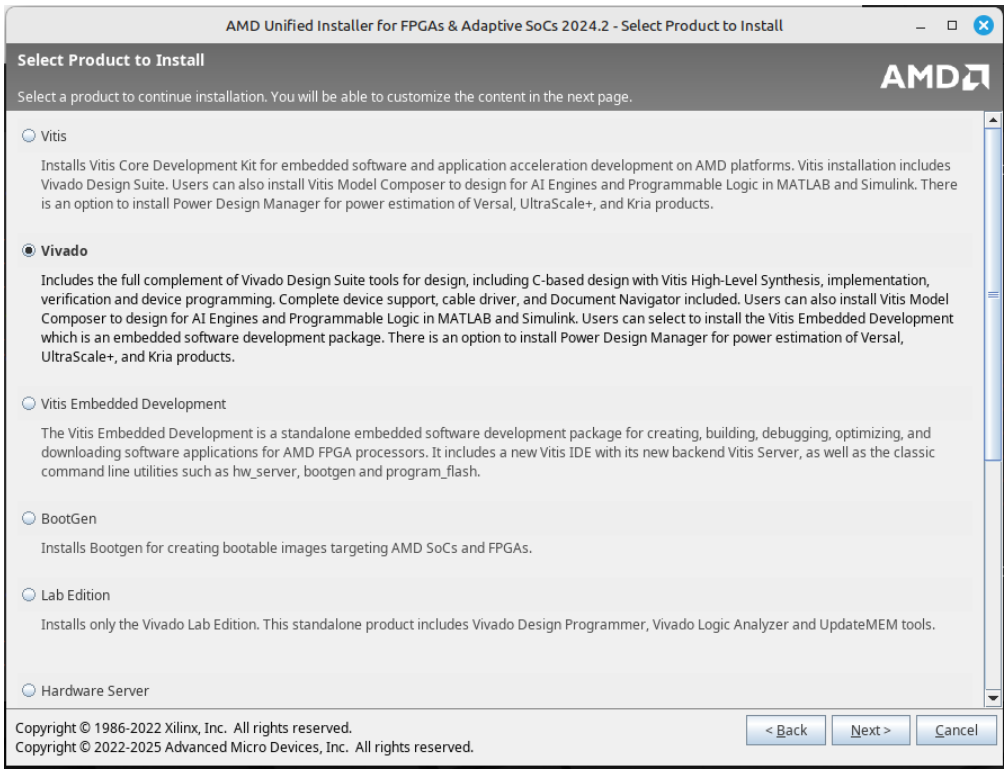


Abbildung 8.4: Auswahl des zu installierenden Produktes

Anschließend muss auf der darauffolgenden Seite die zu installierende Vivado-Edition ausgewählt werden. Dabei ist die kostenfreie „Vivado ML Standard“-Edition zu wählen. Dies ist in Abbildung 8.5 dargestellt.

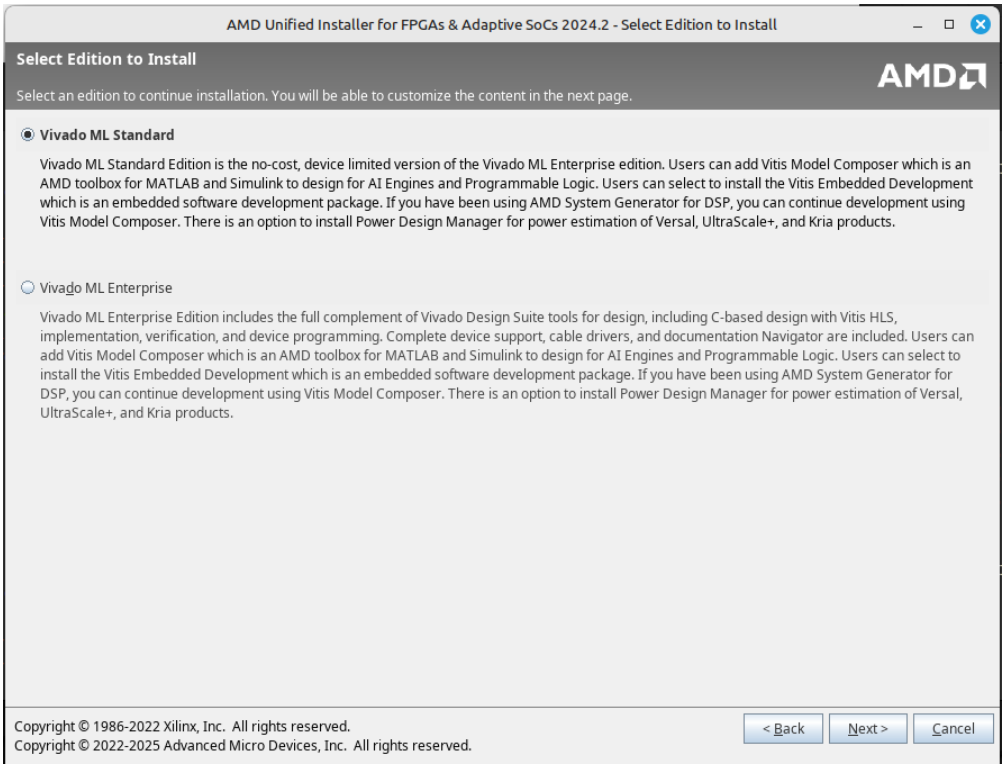


Abbildung 8.5: Auswahl der zu installierenden Vivado-Edition

Danach können verschiedene herunterzuladende Komponenten ausgewählt werden. Die standardmäßig ausgewählten Komponenten sind dabei so zu übernehmen, wie in Abbildung 8.6 zu sehen.

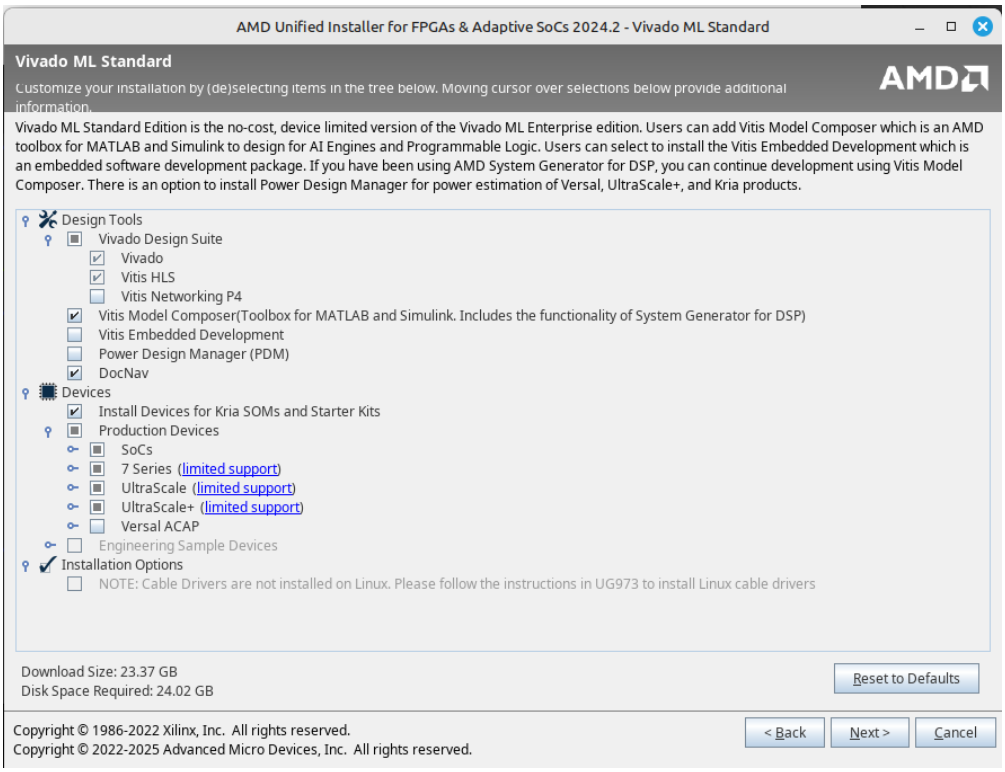


Abbildung 8.6: Auswahl der zu installierenden Tools

Im Anschluss daran müssen die Lizenzbestimmungen der einzelnen Pakete akzeptiert werden. Abschließend kann nun der Download gestartet werden. Nachdem die Installationsdateien heruntergeladen wurden,

lässt sich der heruntergeladene Offline-Installer starten. An dieser Stelle sei erwähnt, dass die Installation nicht als root-Benutzer erfolgen sollte. Zum Starten des Installers muss zunächst in der Konsole in das Verzeichnis navigiert werden, in dem sich die heruntergeladenen Vivado-Dateien befinden. Hier kann der Installer nun mittels des folgenden Befehls gestartet werden:

```
$ ./xsetup
```

Die Installation erfolgt in ähnlichen Schritten wie der Download des Offline-Installers. Zunächst erscheint wieder ein Willkommen-Bildschirm mit verschiedenen Hinweisen. Danach können nun die zu installierenden Komponenten ausgewählt werden. Im Unterschied zum Download sind nun nur noch die heruntergeladenen Komponenten auswählbar wobei auch hier die Standardauswahl zu übernehmen ist. Diese ist identisch zu der in Abbildung 8.6 dargestellten Auswahl. Auch hier müssen die Lizenzbestimmungen der Tools wieder akzeptiert werden.

Danach muss der Installationspfad für Vivado ausgewählt werden. Der unter „Select the installation directory“ eingetragene Pfad muss bereits existieren und das gewählte Verzeichnis durch den aktuellen Benutzer schreibbar sein, außerdem sollte dieser keine Leerzeichen enthalten. Wie in Abbildung 8.7 dargestellt wird der Pfad „/opt/Xilinx“ angelegt und ausgewählt. In dieses Verzeichnis werden Vivado sowie die ausgewählten Komponenten installiert. Anschließend kann die Installation gestartet werden.

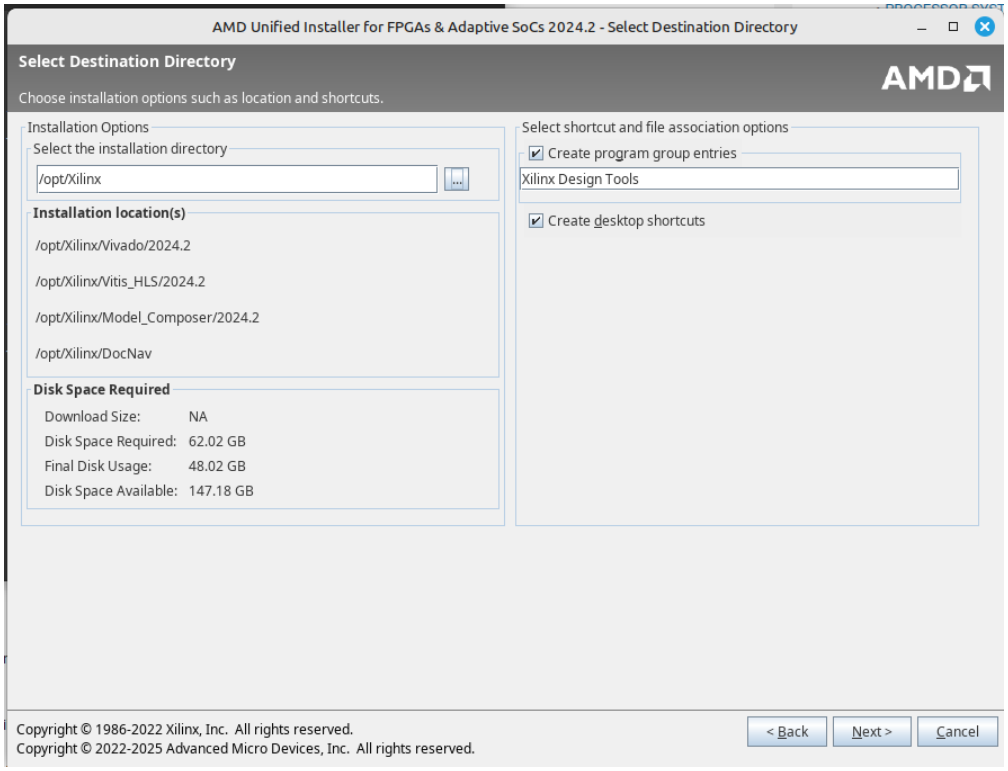


Abbildung 8.7: Auswahl der zu installierenden Tools

8.1.4 Vorbereitungen zur Installation des FINN-Compilers

geschrieben von Simon Sajnog

Dieser Abschnitt beschreibt nach [FIN24] die notwendigen Vorkehrungen, damit im nächsten Schritt die Installation des FINN-Compilers fehlerfrei abläuft. Es wird empfohlen, diesen Schritt nicht parallel zur Installation der AMD Vivado Design Suite auszuführen, sondern bis zu deren erfolgreichen Abschluss zu warten.

Anlegen von notwendigen Umgebungsvariablen

geschrieben von Marvin Hoyer

Bevor der FINN-Compiler installiert werden kann sind auch hier einige notwendige Einstellungen vorzunehmen. Der FINN-Compiler benötigt einen Verweis auf die installierte AMD Vivado Design Suite. Dazu müssen die Umgebungsvariablen `FINN_XILINX_PATH` und `FINN_XILINX_VERSION` angelegt werden. Diese verweisen auf das Installationsverzeichnis von Vivado sowie die davon installierte Version. Um diese Umgebungsvariablen permanent anlegen zu können, muss die Datei „.bashrc“ geöffnet werden:

```
$ ./bashrc
```

In diese Datei müssen nun die folgenden Zeilen hinzugefügt werden:

```
export FINN_XILINX_PATH=/home/mint/Programs/Xilinx
export FINN_XILINX_VERSION=2024.2
```

Um die neuen Umgebungsvariablen auch in die Konsole zu übernehmen, muss einmalig der Befehl

```
$ exec $BASH
```

ausgeführt werden. Zur Überprüfung, ob die Aktualisierung erfolgreich war, kann mittels folgendem Befehl der Wert einer Umgebungsvariable angezeigt werden, z.B.:

```
$ printenv FINN_XILINX_VERSION
```

Zusätzlich kann über eine Umgebungsvariable auf gleiche Weise das zu verwendende Board eingestellt werden. Standardmäßig ist das PYNQ-Z1-Board bei FINN eingestellt. Mit dem folgendem Eintrag wird hier das eingestellte Board auf das in diesem Projekt verwendete Board PYNQ-Z2 umgestellt:

```
export PYNQ_BOARD=Pynq-Z2
```

Installation von Docker

geschrieben von Marvin Hoyer

Es wird empfohlen, die auf der Installationsseite von Docker unter [Docb] beschriebenen Pre-Installationsschritte auszuführen, z.B. zur Deinstallation alter Versionen von Docker. Da es sich bei dem verwendeten Betriebssystem jedoch um ein vollständig neu aufgesetztes System ohne vorherige Installationen handelt, sind diese Schritte in diesem Projekt nicht notwendig und werden nicht weiter behandelt.

Docker kann über die Konsole mittels „apt“ heruntergeladen und installiert werden. Dazu muss zunächst das „Docker apt repository“ hinzugefügt werden, wofür die Ausführung der folgenden Befehle in der hier angegebenen Reihenfolge notwendig ist:

```
$ sudo apt-get update

$ sudo apt-get install ca-certificates curl

$ sudo install -m 0755 -d /etc/apt/keyrings

$ sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/
    keyrings/docker.asc

$ sudo chmod a+r /etc/apt/keyrings/docker.asc

$ echo \
```

```
"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
    https://download.docker.com/linux/ubuntu \
$(. /etc/os-release && echo "$UBUNTU_CODENAME") stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Hierbei ist anzumerken, dass im obigen Befehl `$UBUNTU_CODENAME` speziell für Ubuntu-basierte Distributionen genutzt werden muss. Für andere Linux-Distributionen muss dort stattdessen `$VERSION_CODENAME` eingesetzt werden [siehe Docb].

Im Anschluss an die Ausführung dieses Befehls muss apt erneut aktualisiert werden:

```
$ sudo apt-get update
```

Danach kann Docker installiert werden. Dazu muss der folgende Befehl in der Kommandozeile ausgeführt werden:

```
$ sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin
    docker-compose-plugin
```

Zur Verifizierung, dass die Installation erfolgreich verlief, kann nun der Hello-World Dockercontainer gestartet werden, der einen Hello-World-Text auf der Konsole ausgibt. Wenn dieser ohne Fehlermeldungen startet, ist die Installation von Docker abgeschlossen. Gestartet werden kann der Container mittels:

```
$ sudo docker run hello-world
```

Docker-Rechte konfigurieren

geschrieben von Simon Sajnog

Der FINN-Compiler macht es erforderlich, dass Docker ohne root-Benutzer ausgeführt werden kann. Dazu muss zunächst eine neue Benutzergruppe angelegt werden [siehe Docc]:

```
$ sudo groupadd docker
```

Dieser Gruppe muss nun der aktuelle Benutzer hinzugefügt werden. Sollte die Gruppe bereits existieren, muss keine neue Gruppe angelegt werden, sondern ebenfalls nur der aktuelle Benutzer dieser Gruppe hinzugefügt werden. Dies wird mit folgendem Befehl erreicht:

```
$ sudo usermod -aG docker $USER
```

Als nächsten Schritt empfiehlt es sich, den Rechner einmalig vollständig neu zu starten. Anschließend kann erneut der Hello-World-Container von Docker ausgeführt werden, um die Rechteänderung zu testen. Startet dieser Container ohne „sudo“ im Befehl, wurden die Rechte korrekt übernommen:

```
$ docker run hello-world
```

8.1.5 Installation des FINN-Compilers

geschrieben von Marvin Hoyer

FINN kann durch Klonen des FINN-Git-Repositories auf das Gerät heruntergeladen werden [siehe FIN24]. Dazu muss in der Konsole zunächst an den Ort navigiert werden, an dem sich das Repository nach dem Klonen befinden soll. Ansonsten ist auch ein Verschieben im Anschluss des Klonens möglich. Klonen lässt sich das Repository mit folgendem Befehl:

```
$ git clone https://github.com/Xilinx/finn/
```

Nun wird das Repository heruntergeladen. Im Verzeichnis des Repositories muss nach Abschluss des Downloads der Installations-Quicktest ausgeführt werden:

```
$ ./run-docker.sh quicktest
```

Die Ausführung dieses Befehls startet eine Reihe von Downloads von für FINN notwendigen Bibliotheken. Dies kann je nach Internetverbindung einige Zeit dauern. Sobald die Installation abgeschlossen ist, lässt sich mittels folgendem Befehl der Jupyter Notebook-Server starten:

```
$ bash ./run-docker.sh notebook
```

Dieser Befehl startet den Server von Jupyter Notebook. Die dazu aufzurufende URL wird nach kurzer Zeit in der Konsole ausgegeben und hat folgendes Format:

```
http://127.0.0.1:8888/?token=[...]
```

Der im Link enthaltene Token ist nach jedem Start des Jupyter Servers ein anderer.

8.2 Inspizierung des vorhandenen NNs

geschrieben von Simon Sajnog

Da aufgrund von Problemen beim Training des neuronalen Netzes dieses nicht erstellt werden kann, wird für diesen Schritt nun das neuronale Netz der vorherigen Gruppe aus dem Wintersemester 2022/23 eingesetzt [siehe Git]. Hierbei handelt es sich um das `DefaultModel.onnx`-NN. Die verwendete ONNX-Datei ist im GitLab-Repository unter „library/finn-framework“ zu finden. Um nachvollziehen zu können, wie dieses Modell aufgebaut ist, wurde neben dem Projektbericht [siehe Bur23] auch Netron verwendet. Nach [Net] ist Netron ist eine Anwendung zur Visualisierung von neuronalen Netzen in verschiedenen Formaten, wie z.B. in diesem Fall ONNX. Damit können die unterschiedlichen Layer des Netzes dargestellt und genauer analysiert werden. Dargestellt werden bspw. die Größen der Eingabe- und Ausgabedaten sowie die Gewichte oder die von den einzelnen Layern verwendeten Aktivierungsfunktionen. Bei der Inspizierung stellte sich heraus, dass das Netz bereits quantisiert ist und somit für die folgenden Schritte verwendet werden kann. Das Model hat den in Abbildung 8.8 dargestellten Aufbau.

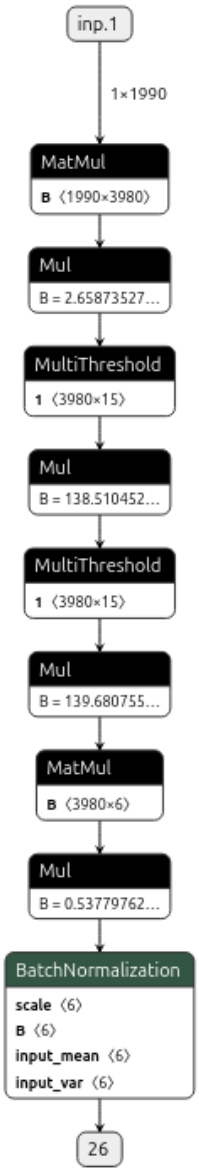


Abbildung 8.8: Das DefaultModel.onnx-Model, dargestellt in Netron.

8.3 Erstellen des IP-Blocks aus dem NN

geschrieben von Thomas Klein

Wie bereits in Abschnitt 2.4.6 aufgeführt bietet das FINN-Framework verschiedene Möglichkeiten um ein neuronales Netz auf ein FPGA-Board zu laden. In diesem Projekt wird die Möglichkeit genutzt, mittels FINN einen IP-Block zur Verwendung in Vivado zu generieren. Damit ist es möglich, das neuronale Netz als eigenständige Komponente in das Gesamtprojekt zu laden und mit den notwendigen anderen Komponenten entsprechend zu verbinden. Für die Generierung eines IP-Blocks mittels FINN wurde hier das „3-build-accelerator-with-finn.ipynb“ Jupyter-Notebook von FINN herangezogen, dieses ist unter [FINa] oder im Installationsverzeichnis von FINN unter „notebooks/end2end_example/cybersecurity“ zu finden. Es enthält beispielhaften Programmcode für verschiedene interessante mögliche Operationen auf einem neuronalen Netz im ONNX-Format. Die hier angewendete und leicht angepasste Version befindet sich im GitLab-Repository unter „library/finn-framework“.

8.3.1 Erstellen eines Estimate Reports

geschrieben von Thomas Klein

Bevor aus dem vorliegenden neuronalen Netz ein IP-Block erstellt wird, kann optional ein sogenannter Estimate Report generiert werden. Dieser erlaubt es bereits ohne die Durchführung einer Synthese erste Eigenschaften des neuronalen Netzes auf dem FPGA-Board analysieren zu können. Dazu zählen u.a. die Anzahl der Taktzyklen eines jeden Layers des NNs sowie der ungefähre Ressourcenverbrauch auf dem FPGA. Daraus lässt sich erkennen, ob das verwendete NN für das FPGA geeignet ist und wie viele Taktzyklen das neuronale Netz bis zur Ausgabe benötigt. Hierbei handelt es sich jedoch um eine erste grobe Schätzung, da durch die Durchführung des Syntheseprozesses die bisherige Schaltung weiter optimiert und damit die Anzahl an Taktzyklen oder die notwendigen Ressourcen des FPGAs reduziert werden kann. Der Estimate Report generiert viele verschiedene Dateien, in denen diese Abschätzungen aufgeführt sind. Für die Erstellung des Estimate Reports kann der folgende Programmcode aus dem Notebook [FINa] verwendet werden. Hier müssen im ersten Schritt notwendige Bibliotheken eingebunden werden.

```
import finn.builder.build_dataflow as build
import finn.builder.build_dataflow_config as build_cfg
import os
import shutil
```

Anschließend müssen die zu analysierende ONNX-Datei sowie das Ausgabeverzeichnis für den Report angegeben werden. `model_dir` definiert das Verzeichnis, wo die ONNX-Datei gespeichert ist, ausgehend vom Root-Verzeichnis von FINN. `model_file` gibt den Dateinamen der zu analysierenden ONNX-Datei an und `estimate_report_output_dir` definiert das Ausgabeverzeichnis, in welches die Ausgabedateien des Reports geschrieben werden. Das Ausgabeverzeichnis ist dabei ausgehend von dem aktuellen Verzeichnis, in dem sich das ausführende Notebook befindet.

```
model_dir = os.environ['FINN_ROOT'] + "/notebooks/models"
model_file = model_dir + "/DefaultModel.onnx"

estimate_report_output_dir = "default_model_estimate_report"
```

Zur Sicherheit können mit folgendem Code die Ergebnisse einer vorherigen Ausführung ggf. automatisch gelöscht werden, bevor eine neue Ausführung gestartet wird.

```
if os.path.exists(estimate_report_output_dir):
    shutil.rmtree(estimate_report_output_dir)
    print("Vorherige Ergebnisse gelöscht!")
```

Danach kann ein `DataflowBuildConfig`-Objekt erstellt werden. Dieses Objekt besitzt verschiedene Attribute, die bei der Ausführung für bestimmte Einstellungen genutzt werden. Im Folgenden ist die in diesem Projekt verwendete Konfiguration dargestellt.

```
cfg_estimates = build.DataflowBuildConfig(
    output_dir = estimate_report_output_dir,
    mvau_wwidth_max = 80,
    target_fps = 1000000,
    synth_clk_period_ns = 10.0,
    fpga_part = "xc7z020clg400-1",
    steps = build_cfg.estimate_only_dataflow_steps,
    generate_outputs=[
        build_cfg.DataflowOutputType.ESTIMATE_REPORTS,
    ]
)
```

Dabei geben `output_dir` das Ausgabeverzeichnis und `fpga_part` das zu verwendende Board an. „xc7z020clg400-1“ steht hierbei sowohl für das PYNQ-Board Z1 als auch Z2. Die genaue Bezeichnung für ein Board ist unter [Xil] zu finden. Um die HLS-Synthese zu überspringen und damit die Erstellung des Estimate Reports zu beschleunigen, muss für `steps` der Wert `build_cfg.estimate_only_dataflow_steps` angegeben werden. Damit nur die Ergebnisse des Estimate Reports ausgegeben und geschrieben werden, muss dies zusätzlich unter `generate_outputs` angegeben werden, in dem dort nur `build_cfg.DataflowOutputType.ESTIMATE_REPORTS` als gewünschte Ausgabe eingetragen wird. Sobald ein `DataflowBuildConfig`-Objekt erstellt wurde, kann dieses mittels des folgenden Befehls ausgeführt werden. Dies startet den Prozess des Estimate Reports und speichert die Ergebnisse ab.

```
%%time
build.build_dataflow_cfg(model_file, cfg_estimates)
```

Anschließend können die unterschiedlichen generierten Ergebnisse angesehen und analysiert werden. Sowohl die Abschätzung der verbrauchenden Ressourcen des FPGA-Boards als auch die Performance anhand notwendiger Taktflanken finden sich im Ausgabeverzeichnis im Unterverzeichnis „report“. Dort befinden sich verschiedene Dateien im JSON-Format, die mit dem folgenden Befehl ausgelesen werden können, wobei `<Dateiname>` durch den Namen der zu lesenden Datei zu ersetzen ist.

```
read_json_dict(estimate_report_output_dir + "/report/<Dateiname>")
```

Die Datei „estimate_network_performance.json“ beinhaltet die Performance und Latency des Models auf dem FPGA und beinhaltet die folgenden Ergebnisse für das verwendete NN.

```
{
  "critical_path_cycles": 140,
  "max_cycles": 120,
  "max_cycles_node_name": "MVAU_hls_0",
  "estimated_throughput_fps": 833333.3333333334,
  "estimated_latency_ns": 1400.0
}
```

In der Datei „estimate_layer_cycles.json“ stehen die Anzahl an notwendigen Taktzyklen für jedes Layer. Da das langsamste Layer die allgemeine Geschwindigkeit des neuronalen Netzes angibt liegen hier nach 64 Taktzyklen die Ergebnisse an.

```
{'MVAU_hls_0': 60, 'MVAU_hls_1': 64, 'MVAU_hls_2': 64, 'MVAU_hls_3': 64}
```

Zuletzt können noch die verbrauchten Ressourcen analysiert werden, indem die Datei „estimate_layer_resources.json“ ausgelesen wird. Zu den Ressourcen zählen neben der Anzahl verwendeter Look-Up-Tables auch z.B. Blockspeicher, URAMs oder Digital Signal Processing-Blöcke (DSPs). Für die einzelnen Layer des Models ergeben sich die folgenden notwendigen Ressourcen.

```
{
  'MVAU_hls_0': {'BRAM_18K': 36,
  'BRAM_efficiency': 0.11574074074074074,
  'LUT': 6741,
  'URAM': 0,
  'URAM_efficiency': 1,
  'DSP': 0},
 'MVAU_hls_1': {'BRAM_18K': 4,
  'BRAM_efficiency': 0.1111111111111111,
  'LUT': 1149,
  'URAM': 0,
  'URAM_efficiency': 1,
  'DSP': 0},
```

```
 'MVAU_hls_2': {'BRAM_18K': 4,
  'BRAM_efficiency': 0.1111111111111111,
  'LUT': 1148,
  'URAM': 0,
  'URAM_efficiency': 1,
  'DSP': 0},
 'MVAU_hls_3': {'BRAM_18K': 1,
  'BRAM_efficiency': 0.006944444444444444,
  'LUT': 316,
  'URAM': 0,
  'URAM_efficiency': 1,
  'DSP': 0},
 'total': {'BRAM_18K': 45.0, 'LUT': 9354.0, 'URAM': 0.0, 'DSP': 0.0}}
```

8.3.2 Transformation des Models

geschrieben von Marvin Hoyer

Nach der Analyse des Models sind vor dem Generieren des IP-Blocks zunächst weitere Verarbeitungsschritte notwendig. Dafür wurde nicht das von FINN vorhandene Jupyter-Notebook verwendet bzw. erweitert, sondern auf Basis des dort vorhandenen Programmcodes ein eigenes Notebook angefertigt. Dieses Notebook, „create_ip_poc.ipynb“, ist ebenfalls im GitLab-Repository unter „library/finn-framework“ zu finden und im Anhang unter B.3.1 aufgeführt.

In diesem Notebook müssen ebenfalls zunächst notwendige Bibliotheken importiert werden. Zusätzlich umfasst dies für die Transformationen des Models nun auch die Bibliotheken „Onnx“ und „Torch“.

```
import onnx
import torch
import os
```

Danach kann wie zuvor die zu transformierende ONNX-Datei und das gewünschte Ausgabeverzeichnis definiert werden.

```
from qonnx.core.modelwrapper import ModelWrapper
from qonnx.core.datatype import DataType
from finn.transformation.qonnx.convert_qonnx_to_finn import ConvertQONNXtoFINN


# Das Verzeichnis der ONNX-Datei
model_dir = "."
# Das relative Verzeichnis für die Ausgabegabedatein des Notebooks
builder_dir = model_dir + "/builder_output"
# Der Dateiname der ONNX-Datei
ready_model_filename = model_dir + "/DefaultModel.onnx"
```

Das Model muss für die Ausführung verschiedener Transformationen zunächst in ein von FINN definiertes `ModelWrapper`-Objekt geladen werden. Um die Struktur des Models vor Beginn der Tranformationen zu sehen, kann das Model zuvor noch einmal mittels Netron visualisiert werden.

```
from finn.util.visualization import showInNetron, showSrc

# ModelWrapper wird zum Laden der ONNX-Datei in FINN benoetigt
# Zudem werden hierdurch verschiedene Hilsfunktionen fuer das Modell
# zur Verfuegung gestellt
```

```
model = ModelWrapper(ready_model_filename)

# Visualisierung des neuronalen Netzes
showInNetron(ready_model_filename)
```

Damit das Model in einen IP-Block umgewandelt werden kann, muss dieses zunächst transformiert werden. Für diesen Schritt und die noch folgenden Schritte wird nach Abschluss der Transformationen bzw. Operationen eine Sicherheitskopie des jeweiligen Models abgespeichert. Diese Kopie wird dann im nächsten Schritt wieder geladen. Dies erlaubt, bei Anpassungen in einem einzelnen Schritt auch nur diesen und die folgenden Schritte erneut ausführen zu müssen, ohne alle vorherigen Schritte ebenfalls durchlaufen zu müssen. Hier wird bspw. das Model nach der Transformation als „model_tidy.onnx“ abgespeichert.

```
from qonnx.transformation.general import GiveUniqueNodeNames,
    GiveReadableTensorNames, RemoveStaticGraphInputs
from qonnx.transformation.infer_datatypes import InferDataTypes
from qonnx.transformation.infer_shapes import InferShapes
from qonnx.transformation.fold_constants import FoldConstants

model = model.transform(InferShapes())
model = model.transform(FoldConstants())
model = model.transform(GiveUniqueNodeNames())
model = model.transform(GiveReadableTensorNames())
model = model.transform(InferDataTypes())
model = model.transform(RemoveStaticGraphInputs())

# Speichern des transformierten Modells als Zwischenausgabe
model.save(builder_dir + "/model_tidy.onnx")
```

Das transformierte Model kann nun erneut geladen und in Netron angezeigt werden, um die Ergebnisse der Transformationen analysieren zu können.

```
# Visualisierung des transfomierten Modells
model = ModelWrapper(builder_dir + "/model_tidy.onnx")
showInNetron(builder_dir + "/model_tidy.onnx")
```

Dieses Model muss im nächsten Schritt nun auf das Hardware-Layer des verwendeten Boards abgebildet werden. Das daraus resultierende Model wird erneut separat abgespeichert, dieses Mal als „parent.onnx“ und kann ebenfalls wieder mit Netron visualisiert werden.

```
from finn.transformation.fpgadataflow.create_dataflow_partition import
    CreateDataflowPartition

model = ModelWrapper(builder_dir + "/model_tidy.onnx")
parent_model = model.transform(CreateDataflowPartition())
parent_model.save(builder_dir + "/parent.onnx")

# Visualisierung des Modells mit Hardware-Layern
model = ModelWrapper(builder_dir + "/parent.onnx")
showInNetron(builder_dir + "/parent.onnx")
```

8.3.3 Generieren des IP-Blocks

geschrieben von Thomas Klein

Nach den verschiedenen Transformationen ist es nun möglich, das Model in einen IP-Block umzuwandeln. Hierfür wird ebenfalls das „create_ip_poc.ipynb“-Notebook verwendet. Für die Umwandlung in einen IP-Block müssen zunächst verschiedene Werte definiert werden, wie z.B. das zu verwendende Board oder die zu verwendende Taktfrequenz.

```
# FPGA-Board festlegen
pynq_board = "Pynq-Z2"
# FPGA-Board-Informationen auswaehlen
fpga_part = pynq_part_map[pynq_board]
# Takt-Frequenz festlegen
target_clk_ns = 10
```

Es müssen verschiedene Schritte ausgeführt werden, um das Model in einen IP-Block umwandeln zu können. Der erste auszuführende Schritt ist die Vorbereitung des Models zur Implementierung in einem IP-Block mittels der Transformation `PrepareIP`.

```
from finn.transformation.fpgadataflow.prepare_ip import PrepareIP

# Vorbereitung des Modells zur Implementierung in einem IP-Block
model = ModelWrapper(builder_dir + "/parent.onnx")
model = model.transform(PrepareIP(fpga_part, target_clk_ns))
model.save(builder_dir + "/model_prepared.onnx")
```

Danach muss ein QONNX, ein quantisiertes ONNX-Model, in ein FINN-Model umgewandelt werden. Dazu wird die Transformation `ConvertQONNXtoFINN` angewendet.

```
from finn.transformation.qonnx.convert_qonnx_to_finn import ConvertQONNXtoFINN

# Umwandlung des Modells in ein fuer FINN und die weiteren Schritte
# notwendiges Format
model_file = builder_dir + "/model_prepared.onnx"
model = ModelWrapper(model_file)
model = model.transform(ConvertQONNXtoFINN())
model.save(builder_dir + "/model_file_ready.onnx")
```

Dann kann ähnlich zum Prozess beschrieben im offiziellen FINN-Notebook der IP-Block aus dem FINN-Model generiert werden. Anders als dort ist hier für `generate_outputs` lediglich `build_cfg.DataflowOutputType.STITCHED_IP` angegeben, da sowohl die anschließende „RTLSIM_PERFORMANCE“, als auch die „OOC_SYNTH“ keine Auswirkungen auf den generierten IP-Block haben. Diese können ähnlich dem Estimate Report zur Analyse der Eigenschaften des Blocks ausgeführt werden, dies ist hier jedoch nicht notwendig.

Allerdings kommt es bei der Ausführung der Generierung des IP-Blocks zu einem Fehler in Schritt 13 „step_set_fifo_depths“. Hierbei kann die Datei „Vfinn_design_wrapper“ nicht gefunden werden. Dieser Fehler kann umgangen werden, wenn für den Parameter `auto_fifo_depths` der Wert `False` und für `stitched_ip_gen_dcp` der Wert `True` angegeben wird.

```
import finn.builder.build_dataflow as build
import finn.builder.build_dataflow_config as build_cfg
import os
import shutil

# Laden des fuer die IP-Block umgewandelten Modells
model_file = builder_dir + "/model_file_ready.onnx"
model = ModelWrapper(model_file)
rtl_output = model_dir + "/rtl_output"
```

```
# Loeschen von vorherigen Ausgaben des Build-Prozesses
if os.path.exists(rtl_output):
    shutil.rmtree(rtl_output)
    print("Previous run results deleted!")

# Erstellen der Konfiguration zur Erzeugung des IP-Blocks
cfg_stitched_ip = build.DataflowBuildConfig(
    output_dir = rtl_output,
    mvau_wwidth_max = 80,
    target_fps = 1000000,
    synth_clk_period_ns = target_clk_ns,
    fpga_part = fpga_part,
    generate_outputs=[
        build_cfg.DataflowOutputType.STITCHED_IP,
    ],
    auto_fifo_depths = False,
    stitched_ip_gen_dcp = True
)
```

Nach Abschluss dieses Prozesses befindet sich der IP-Block im angegebenen Ausgabeverzeichnis unter „stitched-ip“. Dieses Verzeichnis beinhaltet den aus dem gegebenen neuronalen Netz generierten IP-Block als Vivado-Projekt. Dieses kann nun ohne weitere Änderungen in andere Projekte eingefügt werden. Die Struktur dieses Blocks kann im Blockdiagramm in Abbildung 8.9 eingesehen werden.

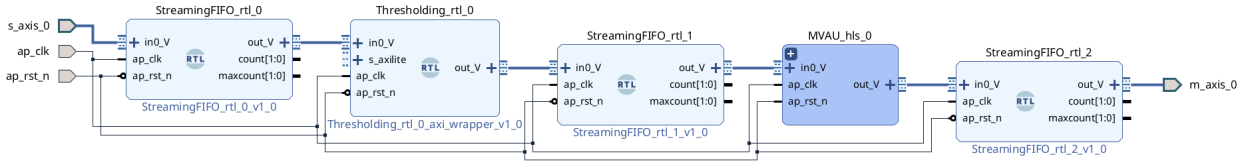


Abbildung 8.9: Blockdiagramm des von FINN generierten IP-Blocks des neuronalen Netzes

8.4 Einbindung des IP-Blocks in das Gesamtprojekt

geschrieben von Marvin Hoyer

Nachdem der IP-Block mittels FINN erstellt wurde, kann dieser in das Gesamtprojekt eingefügt werden. Dazu wird nun das neuronale Netz als IP-Block in die Gesamtschaltung integriert. Da die Bitbreite von Ein- sowie Ausgabesignal des IP-Blocks des neuronalen Netzes nicht mit denen des AXI Direct Memory Access Blocks, an welchen der IP-Block des neuronalen Netzes angeschlossen werden sollte, übereinstimmt, werden für die Verbindung zwei AXI4-Stream Data Width Converter verwendet. Auf diese Weise ist es möglich, die Bitbreite der Ein- und Ausgabesignale anzupassen und so den Block einzubinden. Ebenso ist auch das Validieren des Blockdiagramms ohne Fehler möglich. Grund für die Notwendigkeit dieses Workarounds ist, dass das verwendete neuronale Netz nicht in das geplante Gesamtsystem passt, da es sich beim Model wie bereits in Abschnitt 8.2 erwähnt um das Modell einer anderen Gruppe handelt, die bereits in der Vergangenheit an einem ähnlichen Projekt gearbeitet hat. Das gesamte Blockdiagramm nach Einbindung des neuronalen Netzes und der AXI4-Konverter ist im Anhang in Abbildung C.2 dargestellt.

Abschließend kann die Synthese der Gesamtschaltung gestartet werden. Nach Abschluss der Synthese kann dann die gesamte Schaltung auf das FPGA-Board geladen und ausgeführt werden. Allerdings traten bei der Synthese der Gesamtschaltung weitere Fehler auf und aufgrund von Zeitmangel am Ende des Projektes war es nicht mehr möglich, diese vollständig zu beheben.

Kapitel 9

Evaluation

9.1 Erfassung von Audio

geschrieben von Mattes Bielefeld

Den Kern der Audioerfassung bilden zwei Module, die kontinuierliche Audioaufzeichnung und das übermitteln der Audiodaten an einen Speicherort, auf den weitere Blöcke der Audioverarbeitung zugriff haben. Für die zwei Kernmodule wurden Komponententests ausgearbeitet und in einem Integrationstest wurde die Funktionalität der Kombination dieser beiden überprüft.

9.1.1 Testen der kontinuierlichen Audioerfassung

Um die kontinuierliche Audioerfassung zu testen, wurde die in Blöcke unterteilte Audioaufzeichnung, wie sie für die Audioerfassung für die Weiterverarbeitung geplant ist, für einen zeitlich begrenzten Zeitraum durchgeführt. Zusätzlich wurde eine nicht in Blöcke unterteilte Audioaufzeichnung für denselben Zeitraum durchgeführt als Kontrollprobe. In beiden Fällen wurde Sprache aufgezeichnet. Beide Aufzeichnungen wurden je in eine Wave-Datei übertragen. Abschließend wurde durch ein Vergleich der Audiowiedergabe der beiden Dateien überprüft, inwiefern die Audioerfassung durch die Stückelung der Methode für eine kontinuierliche Aufnahme beeinflusst wurde. Bei der Kontrollprobe handelte es sich um eine völlig normale Audioaufzeichnung. Bei der gestückelten Aufzeichnung wurde festgestellt, dass diese zwar durch die Stückelung beeinträchtigt war, der Sinn in der Aufzeichnung jedoch noch immer klar zu erkennen war. Dieser Test kann unter Anhang B.2.4 nachvollzogen werden.

9.1.2 Testen der Audiodatenübertragung

Die Übertragung von Daten, die für die Audiodaten geplant war, wurde getestet, indem ein kleiner Testdatensatz erstellt wurde. Dieser wurde anschließend mittels des MMIO-Moduls an eine Speicherstelle übertragen. Zuvor wurden die Daten ausgegeben. Anschließend wurde ebenfalls über das MMIO-Modul von derselben Speicherstelle gelesen und die Daten ausgegeben. Die ausgegebenen Daten konnten anschließend verglichen werden, woraus hervorging, dass die Daten korrekt übertragen werden können. Dieser Test kann unter Anhang B.2.5 nachvollzogen werden.

9.1.3 Integrationstest für die Audioerfassung

Abschließend wurde eine mögliche Implementation einer kontinuierlichen Audioaufzeichnung für die Sprachkommandoerkennung angefertigt, in der die kontinuierliche Audioerfassung und die Audiodatenübertragung kombiniert werden. Hier wurde ebenfalls über eine Ausgabe der Audiodaten sichergestellt, dass diese korrekt übertragen werden.

9.2 Verarbeitung von Audio

9.2.1 MFFC Implementation via Python

geschrieben von David Schulz

In der Abbildung 9.1 wird die Python MFCC Implementation, auf die sich geeinigt wurde, mit der Implementation von der Python-Bibliothek Librosa verglichen.

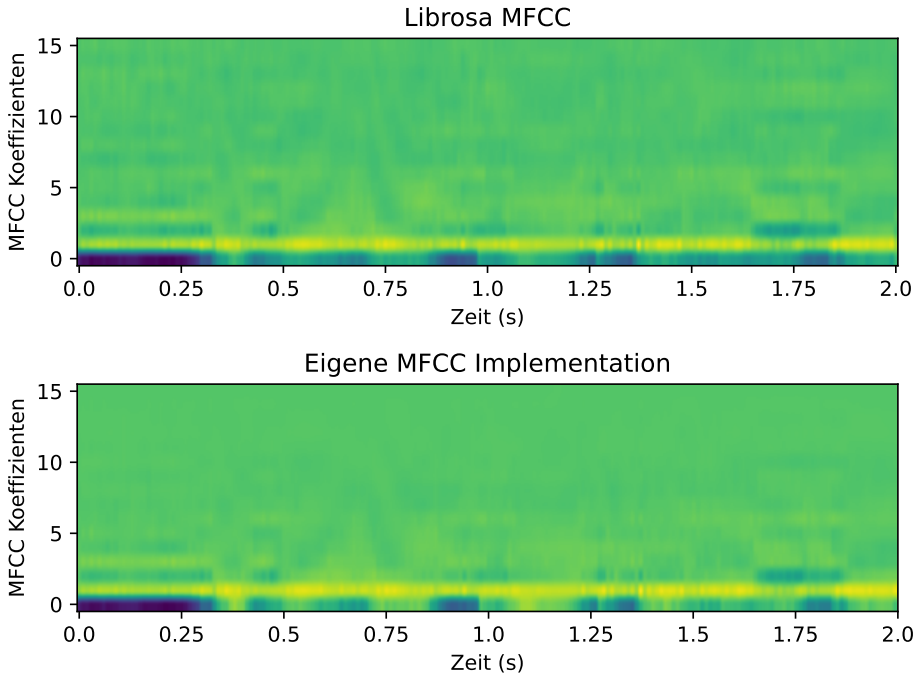


Abbildung 9.1: Vergleich der eigenen Implementation in Python mit Implementation des Packages Librosa.

In der Abbildung 9.1 ist zu erkennen, dass bei den niedrigen MFCC Koeffizienten die beiden Implementation sehr nahe beieinander liegen und größere Unterschiede erst bei den hohen MFCC Koeffizienten auftreten, wo die Librosa Implementation deutlich mehr Details aufweist. Dies ist darauf zurückzuführen, dass die Librosa-Bibliothek implizit Änderungen durchführt, welche in der eigenen Implementation nicht zu tragen kommen. Da die hohen MFCC Koeffizienten Informationen über höhere Frequenzen liefern und die menschliche Stimme im Frequenzbereich von 80 Hz bis 255 Hz liegt, sollte dieser Unterschied bei der Erkennung von Wörtern keine relevante Bedeutung haben [SS24].

9.2.2 Zeitverhalten

geschrieben von David Schulz

Bei der Audioverarbeitung wurde untersucht, inwiefern es sinnvoll ist, die Berechnung der MFCCs auf dem FPGA-Board durchzuführen im Sinne der Performanz.

Die Bestimmung des Zeitverhaltens ermöglicht es herauszufinden, ob es einen Vorteil bietet die Berechnung auf dem FPGA-Board durchzuführen und den damit verbundenen Mehraufwand in Kauf zu nehmen. Denn bei der Implementation auf dem Mikrocontroller kann man auf bereits fertige Implementation zurückgegriffen werden, welche bereits umfangreich getestet und mit Optimierungsverfahren ausgestattet sind, die auf dem FPGA-Board unter Umständen eine komplexere Implementierung bedeuten würden.

Das Zeitverhalten zur Berechnung des MFCCs wurde auf den FFT Teil beschränkt, da dieser den Hauptteil der Berechnung darstellt, wenn anstelle des DCTs das FFT verwendet wird. Um für die Berechnung des FFTs das Zeitverhalten zu bestimmen wurden zuerst 1000 zufällige Eingabedaten generiert, die

jeweils im ersten Fall 1024 Datenpunkte aufwiesen und im zweiten Fall 16 384. Es wurden zwei Fälle betrachtet, da die Vermutung nahe liegt, dass die Implementation auf dem FPGA-Board vor allem durch Parallelität profitieren kann und daher eine höhere Anzahl von Datenpunkte zu einem größeren Vorteil des FFT-Blocks auf dem FPGA-Board führt. Die Begrenzung auf 16 384 Datenpunkte ist auf das verwendete FPGA-Board zurückzuführen. Das PYNQ-Z2 lässt es nicht zu mehr Datenpunkte zu verwenden, da die Anzahl der Datenpunkte eine 2er Potenz sein muss und die nächst höhere 2er Potenz von 32 768 mehr BRAM benötigt als das FPGA-Board zur Verfügung hat. Die Verwendung von 1000 zufällige Eingabedaten hängt damit zusammen, dass dadurch versucht wurde Optimierungen zu vermeiden bzw. den Einfluss dieser auf die Ergebnisse.

Für die Bestimmung des Zeitverhaltens wurde dann eine Funktion `timeit` geschrieben, die es ermöglicht die jeweilige FFT Implementation mithilfe der Eingabedaten zeitlich zu messen. Darüber hinaus berechnet die `timeit`-Funktion den Mittelwert und die Standardabweichung der Messung. Während die Bestimmung der Ausführungszeit der FFT Numpy Python Implementation trivial war, ist die exakte Bestimmung der Ausführungszeit für den FFT-Block nicht möglich, da der Transfer vom Mikrocontroller zum FPGA-Block nötig ist, da dieser Teil nicht optional ist, kann er auch nicht vollständig ausgeschlossen werden. Daher wurden für den FFT-Block 4 verschiedene Szenarien gemessen. Beim ersten Szenario wurde die Zeit des FFT-Blocks und des Datentransfers betrachtet, siehe Quellcode 9.1.

```
def fpga_fft_setup(y: np.ndarray):
    np.copyto(input_buffer, y)

def fpga_fft(y: np.ndarray):
    fft_data_send_channel.transfer(input_buffer)
    fft_data_receive_channel.transfer(output_buffer)
    fft_data_send_channel.wait()
    fft_data_receive_channel.wait()
```

Listing 9.1: Quellcode für die Setup-Funktion und die FFT-Funktion für den FFT-Block auf dem FPGA-Board, dabei ist der Datentransfer Teil der FFT-Funktion.

Beim zweiten Szenario wurde die Zeit des FFT-Blocks ohne den Beginn des Sendetransfers betrachtet, siehe Quellcode 9.2.

```
def fpga_fft_setup(y: np.ndarray):
    np.copyto(input_buffer, y)
    fft_data_send_channel.transfer(input_buffer)

def fpga_fft_without_transfer(y: np.ndarray):
    fft_data_receive_channel.transfer(output_buffer)
    fft_data_send_channel.wait()
    fft_data_receive_channel.wait()
```

Listing 9.2: Quellcode für die Setup-Funktion und die FFT-Funktion für den FFT-Block auf dem FPGA-Board, dabei wurde der Beginn des Sendetransfers ausgenommen.

Beim dritten Szenario wurde die Zeit des FFT-Blocks ohne den Beginn des Sendetransfers und ohne den Beginn des Empfangstransfers betrachtet, siehe Quellcode 9.3.

```
def fpga_fft_setup_2(y: np.ndarray):
    np.copyto(input_buffer, y)
    fft_data_send_channel.transfer(input_buffer)
    fft_data_receive_channel.transfer(output_buffer)

def fpga_fft_without_transfer_2(y: np.ndarray):
    fft_data_send_channel.wait()
    fft_data_receive_channel.wait()
```

Listing 9.3: Quellcode für die Setup-Funktion und die FFT-Funktion für den FFT-Block auf dem FPGA-Board, dabei wurde auf das Ende des Sendetransfers und Empfangtransfers gewartet.

Beim vierten Szenario wurde die Zeit des FFT-Blocks betrachtet, wenn man nur auf das Ende des Empfangstransfers wartet, siehe Quellcode 9.4. Dabei ist anzumerken, dass diese Messung nicht die

exakte Zeit des FFT-Blocks zeigt, da die Berechnung des FFT-Blocks bereits anfangen kann, bevor auf das Ergebnis gewartet wird.

```
def fpga_fft_setup_3(y: np.ndarray):
    np.copyto(input_buffer, y)
    fft_data_send_channel.transfer(input_buffer)
    fft_data_receive_channel.transfer(output_buffer)
    fft_data_send_channel.wait()

def fpga_fft_without_transfer_3(y: np.ndarray):
    fft_data_receive_channel.wait()
```

Listing 9.4: Quellcode für die Setup-Funktion und die FFT-Funktion für den FFT-Block auf dem FPGA-Board, dabei wurde nur auf das Ende des Empfangstransfers gewartet. Bei dieser Messung erhält man nicht die exakte Zeit, die der FFT-Block gebraucht hat, da der FFT-Block bereits mit der Berechnung starten kann, bevor auf das Ergebnis gewartet wird.

Beschreibung	Mittelwert$\pm$Standardabweichung der Zeit für 1000 Durchläufe mit zufälligen Daten
In Python via Numpy	$(391{,}29 \pm 28{,}11)\,\mu s$
FFT-Block mit Datentransfer	$(791{,}57 \pm 56{,}00)\,\mu s$
FFT-Block ohne Beginn des Sende-Transfers, aber mit Warten auf Beendigung des Transfers	$(498{,}37 \pm 80{,}40)\,\mu s$
FFT-Block ohne Sende- und Empfang-Transfer, aber mit Warten auf Beendigung des Transfers	$(391{,}57 \pm 72{,}68)\,\mu s$
FFT-Block ohne Datentransfer (nicht exakte Zeit des FFT-Blocks)	$(292{,}68 \pm 128{,}52)\,\mu s$

Tabelle 9.1: Zeitverhalten für die FFT Implementation in Numpy und vom IP-Block auf dem FPGA-Board für 1024 Datenpunkte.

Beschreibung	Mittelwert$\pm$Standardabweichung der Zeit für 1000 Durchläufe mit zufälligen Daten
In Python via Numpy	$(9{,}14 \pm 0{,}85)\,ms$
FFT-Block mit Datentransfer	$(1{,}36 \pm 0{,}08)\,ms$
FFT-Block ohne Beginn des Sende-Transfers, aber mit Warten auf Beendigung des Transfers	$(0{,}99 \pm 0{,}02)\,ms$
FFT-Block ohne Sende- und Empfang-Transfer, aber mit Warten auf Beendigung des Transfers	$(0{,}87 \pm 0{,}11)\,ms$
FFT-Block ohne Datentransfer (nicht exakte Zeit des FFT-Blocks)	$(0{,}76 \pm 0{,}04)\,ms$

Tabelle 9.2: Zeitverhalten für die FFT Implementation in Numpy und vom IP-Block auf dem FPGA-Board für 16 384 Datenpunkte.

In den Tabellen 9.1 und 9.2 sind die Ergebnisse für das Zeitverhalten der FFT Implementation in Python mit Numpy und des FFT-IP-Blocks aufgelistet. Dabei ist zu erkennen, dass im Fall mit 1024 Datenpunkten, siehe Tabelle 9.1, selbst im besten Fall der FFT-IP-Block nur knapp 100 µs schneller ist, aber dafür eine deutlich höhere Standardabweichung von 128,52 µs im Vergleich zu 28,11 µs von

der Numpy Python Implementation aufweist. Die Zeit 292,68 µs für den FFT-IP-Block bei dem dieser knapp 100 µs schneller war als die Numpy Python Implementation ist auch nicht die exakte Zeit, die der FFT-Block braucht, da der FFT-Block bereits vor dem Start der Zeiterfassung anfangen kann zu berechnen. Bei den anderen Ergebnissen ist die Numpy Python Implementation schneller und hat eine geringere Standardabweichung.

Bei den Ergebnissen aus Tabelle 9.2 mit 16 384 Datenpunkten ist zu erkennen, dass selbst wenn man den Datentransfer vom Mikrocontroller zum FPGA-Board dazu nimmt, also im Worst-Case für den FFT-Block dieser deutlich schneller ist als die Numpy Python Implementation und zusätzlich auch eine geringere Standardabweichung hat. In allen anderen Fällen ist der FFT-Block auch deutlich schneller als die Numpy Python Implementation.

Einerseits lässt sich die Vermutung bestätigen, dass bei hoher Anzahl von Datenpunkten, hier 16 384, der FFT-Block schneller ist als die Numpy Python Implementation, allerdings bei niedrigen Datenpunkten, hier 1024, sollte der zeitliche Unterschied zwischen dem FFT-Block und der Numpy Python Implementation vernachlässigbar sein für die meisten Anwendungszwecke. Ob es allerdings ein Vorteil ist die Implementation auf dem FPGA-Board mittels des FFT-Blocks durchzuführen, ist nach diesen Ergebnissen fraglich, denn um eine Spracherkennung durchzuführen oder sogar Stimmen zu untersuchen reicht eine Abtastrate von 16 kHz was für ein Fenster von 20 ms 320 Datenpunkte ergibt [SLS24]. Für den FFT-Block wäre die Anzahl der Datenpunkte 512, da die Anzahl der Datenpunkte eine 2er Potenz sein muss. Da bei geringerer Anzahl von Datenpunkten davon auszugehen ist, dass der FFT-Block keinen relevanten zeitlichen Vorteil bietet, kann Performanz kein Grund für die Verwendung des FFT-Blocks sein. D. h. es sollte damit hauptsächlich davon abhängen, ob eine Pipeline im FPGA-Board aufgebaut werden und ob sich der immens höhere Arbeitsaufwand lohnt, da sich aus Performanzgründen keinen Grund gibt bei kleiner Anzahl von Datenpunkten diese Berechnungen auf dem FPGA-Board auszuführen. Der Mehraufwand besteht vor allem darin, alle Teile zur Berechnung der MFCCs in IP-Blöcken umzusetzen, da es zum Zeitpunkt des Projekts keine fertigen frei verfügbaren IP-Blöcke gab, welche verwendbar wären.

9.2.3 MFCC Implementation via VHDL für das FPGA

geschrieben von Moses Dimmel

Die Implementation via VHDL war aufgrund von Problemen mit der Implementierung bzw. Einbindung des Fixed-Point Datentyps nicht erfolgreich und wurde abgebrochen. Eine Verwendung des Datentyps eignete sich lediglich für die Ausführung des Programms in der Simulation. Auf dem Board selbst hätte dies zu viele Rechenressourcen beansprucht und wäre nicht verwendbar gewesen.

9.2.4 MFCC Implementation via HLS für das FPGA

geschrieben von Moses Dimmel

Der HLS-C++-Code für die MFCC-Berechnung konnte erfolgreich synthetisiert werden. Für diverse Tests und Einbindung in das Gesamtprojekt reichte die Zeit leider nicht. Durch die Verwendung des externen IP-Blocks von Vivado zur Berechnung der FFT und durch die Berechnung der DCT mithilfe der FFT, sowie durch die Vorberechnung der Hamming-Koeffizienten und der Mel-Filterbank konnten die benötigten Rechenressourcen auf dem FPGA immens vermindert werden.

9.2.5 Ausblick der MFCC Implementation via HLS für das FPGA

geschrieben von Moses Dimmel

Mithilfe von Pragma-Statements wie `#pragma HLS UNROLL` an Schleifen oder `#pragma HLS PIPELINE`, welches dem Compiler angibt, dass Pipelining verwendet werden soll kann der bestehende Code noch weiter optimiert werden. Zudem können mit `#pragma HLS BIND_STORAGE ...` die Arrays der vorberechneten

Werte der Hamming-Koeffizienten und der Mel-Filterbank im Block-RAM gespeichert werden, statt in Lookup-Tables. Dies ist essentiell da das FPGA-Board nur eine begrenzte Anzahl von Lookup-Tables besitzt und die vorberechneten Werte ausserhalb des BRAMs diese bei weitem übersteigen würden.

9.3 Training Neuronales Netz

geschrieben von Friederike Korte

Im Rahmen dieses Projektes haben wir zwei verschiedene Ansätze zum Trainieren eines Neuronalen Netzes implementiert. Der erste Ansatz, angelehnt an das Vorgängerprojekt und eigene Überlegungen und Recherche ist direkt im `ki`-Projekt implementiert. Dieser Ansatz konnte leider kein erfolgreiches Modell liefern. Der alternative zweite Ansatz basiert auf einem Tutorial von PyTorch und ist in einem Jupyter-Notebook implementiert (liegt ebenfalls im `ki`-Projekt). Hier wurden im Vergleich zum ersten Ansatz zwar Fortschritte erzielt, ein vollständiges Modell konnte jedoch auch hier nicht trainiert werden.

Die beiden Ansätze sind im Folgenden noch etwas genauer beschrieben:

Vorverarbeitung Daten

Bevor mit dem Training begonnen werden kann, werden die Audiodaten eingelesen, verändert (auf genau 2 Sekunden Länge gebracht und Hintergrundgeräusche hinzugefügt) und dann als Tensor-Objekte gespeichert.

Ansatz 1

In diesem Ansatz wird überwachtes Lernen genutzt. Als Neuronale Netze kommen dabei das Modell aus dem Vorgängerprojekt (`PreviousModel`) und ein eigens definiertes Modell (`Model2RNN`) zum Einsatz. Je nach Verfügbarkeit wird hier mit der CPU oder GPU trainiert, wobei das Trainieren mit einer GPU prinzipiell deutlich schneller ist.

Die zuvor erstellten Tensor-Objekte wurden in jedem Batch auf die gleiche Länge gebracht (*gepadded*), bevor sie in das Neuronale Netz zum Trainieren gegeben wurden. Zur Optimierung wurde die CrossEntropyLoss-Funktion mit dem Adam-Optimizer verwendet. Das Training umfasste zehn Epochen mit fortlaufender Evaluierung.

Trotz intensiver Analyse zur Problemblösung scheiterte das Training aufgrund von Inkompatbilitäten in der Netzwerkarchitektur (Matrix-Dimensionierungsfehler). Gründe hierfür können eine fehlerhafte Architektur oder eine inkosistente Datenverarbeitung sein.

Ansatz 2

Kurzfristig haben wir noch einen zweiten Ansatz implementiert, der sich an dem Tutorial *Speech Command Classification with torchaudio* orientiert. Im Unterschied zum ersten Ansatz werden die Tensor-Objekte hier auf eine einheitliche Abtastrate gesampled und auf die Basis float32 transformiert (statt int16). Grund hierfür ist eine bessere Kompatibilität mit Neuronalen Netzen, die oftmals einen normalisierten Float-Wert erwarten. Zur Optimierung wurde hier ebenfalls die CrossEntropyLoss-Funktion mit dem Adam-Optimizer verwendet.

Mit diesem Ansatz war das Trainieren auf einer CPU zu Beginn nur sehr eingeschränkt möglich. Der Prozess ist nach wenigen Minuten aufgrund eines Speicherplatz-Problemes abgebrochen. Nach einer Reduzierung der `batch_size` von 256 auf 16 und dem Trainieren auf einer AMD-GPU konnte dieses Problem gelöst werden.

Nach einer Laufzeit von 1 1/2 bis 2 Stunden zeigte sich jedoch, dass das Modell nicht weiterkam. Dies könnte auf verschiedene Faktoren zurückzuführen sein, wie z.B. unzureichende Datenqualität, suboptimale

Hyperparameter oder die Modellarchitektur.

Da leider kein Modell erfolgreich trainiert werden konnte, wird im weiteren Verlauf auf ein bereits bestehendes Modell des Vorgängerprojektes zurückgegriffen. Hier ist es gelungen, ein Neuronales Netz zu Trainieren, welches beim Klassifizieren von Audiodateien eine Erkennungsrate von ca. 90 Prozent erreicht. Beim Klassifizieren von Audiosignalen, die direkt über das an das FPGA-Board angeschlossene Mikrofon kommen, werden jedoch nur 5 Prozent Erkennungsrate erreicht. Bei Verwendung des Nachfolgemodells des FPGA-Boards mit einer verbesserten Audioschnittstelle besteht die Möglichkeit, dass hier eine bessere Erkennungsrate erzielt wird.

9.4 Neuronales Netz auf dem FPGA

geschrieben von Simon Sajnog

Aus der Durchführung des Versuchs, ein neuronales Netz zu transformieren und in Form eines IP-Blocks auf ein FPGA zu bringen, lässt sich erkennen, dass der Erfolg dieses Versuchs von vielen verschiedenen Faktoren abhängig ist. Zunächst ist es von Vorteil, einen leistungsstarken Rechner zu verwenden, da sowohl FINN bei der Erstellung des IP-Blocks als auch Vivado bei der Synthese des Gesamtprojekts nicht nur einen hohen Ressourcenverbrauch aufweisen, sondern beide Systeme auch eine lange Laufzeit bis zu einem Ergebnis benötigen.

Die Verwendung eines fremden neuronalen Netzes, in diesem Fall das Modell einer vorherigen Gruppe, ist kein geeigneter Ansatz für die Durchführung des Projekts. Die Eingabe- und Ausgabegrößen des Netzes, die Struktur der erwarteten Eingabedaten und wie die anschließende durch das Modell erfolgte Klassifizierung ausgegeben werden soll, muss in Form einer Schnittstellendefinition mit den anderen Komponenten und damit mit anderen Projektteams abgestimmt werden. Da hier allerdings kein eigenes neuronales Netz zur Verfügung stand, wurde ein fremdes Modell verwendet, um den Prozess genauer untersuchen zu können und eventuell dennoch erfolgreiche Ergebnisse zu erzielen. Von diesem Model waren die Schittstellenspezifikationen jedoch nur bedingt bekannt und sie stimmten nicht vollständig mit den Spezifikationen anderer eingesetzter Komponenten überein.

Bei der Nutzung von FINN ist aufgefallen, dass es möglich ist, ein älteres unbekanntes neuronales Netz in einen IP-Block umzuwandeln und diesen anschließend mit Vivado in das Gesamtprojekt einzufügen. Allerdings kommt es dabei in verschiedenen Schritten zu unterschiedlichen Fehlern, sowohl bei FINN, als auch bei Vivado. Die Fehlermeldungen sind dabei nicht aussagekräftig genug, um die genaue Ursache der Fehler bestimmen zu können und aufgrund von Zeitmangel konnte bis zuletzt keine erfolgreiche Synthese des Gesamtsystems durchgeführt werden.

Zusammenfassend lässt sich somit feststellen, dass FINN zwar zur Umsetzung eines neuronalen Netzes auf ein FPGA geeignet ist, dafür allerdings auch die Struktur und der Aufbau des Netzes von Beginn an definiert und bekannt sein muss. Dies ist notwendig, damit alle Komponenten geeignet miteinander kommunizieren können. Andernfalls ist der Versuch, eine funktionierende Schaltung erzeugen zu können, sowohl mit FINN als auch mit Vivado mit einem erhöhten Aufwand verbunden, welcher zu einem nicht zufriedenstellenden Ergebnis führt.

Kapitel 10

Ergebnisse

geschrieben von Jan Bredereke

Wir untersuchen am Beispiel einer Sprachkommandoerkennung, wie man ein neuronales Netz in einer stark leistungsbeschränkten Umgebung am besten nutzen kann, indem man ein feldprogrammierbares Gate-Array (FPGA) einsetzt. Das im vorliegenden Bericht beschriebene Projekt baut auf Projekten in früheren Semestern auf, und es wird fortgesetzt werden. Dieses Kapitel fasst zusammen, was im aktuellen Semester geschehen ist.

Zusammenfassend wurden die nötigen Verarbeitungsschritte an neue Audio-Hardware angepasst sowie detaillierter identifiziert und festgelegt. Aus einem aufgetretenen Schnittstellenproblem heraus ergab sich eine neue Einsicht zu einer Systemarchitektur für eine effiziente Verarbeitung auf einem FPGA. Bei den einzelnen Teilaufgaben wurden ansonsten bedauerlicherweise keine wesentlichen Fortschritte gegenüber dem Vorgängerprojekt erzielt.

Im einzelnen:

Systemarchitektur: Zunächst wurde die als Endziel geplante Systemarchitektur detaillierter ausgearbeitet, die eine Verarbeitung komplett auf einem FPGA ermöglichen soll, siehe Abbildung 3.1 auf Seite 17. (Vergleiche das Vorgängerprojekt von Burggräf u.a. [Bur23, Abb. 3 auf S. 10] und das Vorvorgängerprojekt von Hagen u.a. [Hag22, Abb. 3.1 auf S. 23].) Nun ist außer der Motoransteuerung insbesondere auch die zweimalige zeitliche Fensterung explizit:

- Die erste zeitliche Fensterung erzeugt Blöcke von Audio-Amplituden über die Zeit, mit Blöcken in einer Größenordnung von 20 ms. Hierauf wird die schnelle Fourier-Transformation FFT angewendet.
- Nach einer Mel-Filterung erzeugt die zweite zeitliche Fensterung Blöcke von Mel-Frequenz-Cepstrum-Koeffizienten MFCC über die Zeit, mit Blöcken in einer Größenordung von 2 s. Hierauf soll ein neuronales Netz ein jeweils in seiner Gesamtheit vorliegendes Sprachkommando erkennen.

Die zeitliche Fensterung wird weiter unten gleich noch einmal relevant.

Audiodatenverarbeitung: Der Anfang der Audiodatenverarbeitungskette wurde komplett neu entwickelt, da wie geplant nun ein PYNQ-Z2 Board anstelle des Vorgängermodells PYNQ-Z1 eingesetzt wurde. Dies ermöglicht Audioaufnahmen in höherer Qualität, weil nun der im PYNQ-Z2 Board verbaute 24-Bit Analog-Digital-Wandler ADAU1761 verwendet wird.

Zu Beginn der jetzigen Arbeit war unklar, wie und wo genau die Audiodaten aus dem Analog-Digital-Wandler zur Verfügung stehen. In Kapitel 2.3.3 findet sich nun ein guter Überblick über den Aufbau des Analog-Digital-Wandlers ADAU1761, einschließlich vieler Quellverweise. Ob und ggf. auf welche Weise der Analog-Digital-Wandler direkt an das FPGA angebunden ist, wurde dagegen noch nicht herausgefunden. Dies wird nötig sein, um auf dem FPGA eine durchgehende Verarbeitungskette zu realisieren.

Trotzdem wurde ein erster Zugriff auf die Audioaufnahme realisiert. Dies geschah mit Hilfe des mit dem PYNQ-Board mitgelieferten Beispielcodes BaseOverlay. Das BaseOverlay enthält zum einen die nötige FPGA-Schaltung und zum anderen die nötige Python-Bibliothek, um auf die diverse Hardware des PYNQ-Boards von der Hochsprache Python aus zugreifen zu können. Durch Verwenden des BaseOverlays wurde ohne tiefere Einarbeitung die Initialisierung des Analog-Digital-Wandlers realisiert.

Die für eine eigene Verwendung des Analog-Digital-Wandlers nötigen spezifischen Komponenten des BaseOverlays wurden noch nicht identifiziert. Stattdessen wurde ein Weg ausgearbeitet, um das BaseOverlay um eigene Funktionalität zu erweitern und dann neu zu übersetzen.

Transformation der rohen Audiodaten in Mel-Frequenz-Cepstrum-Koeffizienten: Hier wurden die nötigen Berechnungen zunächst nicht für das FPGA, sondern in der Hochsprache Python angegangen. Der gleichartige Ansatz des Vorgängerprojekts von Burggräf u.a. [Bur23, Kap. 4.5 auf S. 19] auf Basis einer vorgefertigten Python-Bibliothek wurde dabei nicht wieder aufgegriffen. Stattdessen wurden die einzelnen Verarbeitungsschritte getrennt implementiert, um es zu ermöglichen, diese Verarbeitungsschritte später ebenso separat in IP-Blöcke für das FPGA zu übertragen. Trotzdem wurde zusätzlich eine Lösung mit der Python-Bibliothek Librosa implementiert, um Vergleichswerte für die eigene Implementation zu bekommen. Die Evaluation in Kap. 9.2.1 legt nahe, dass die resultierenden MFCC-Koeffizienten wie gewünscht eine Ähnlichkeit haben.

Für die entsprechende Verarbeitung im FPGA wurde die Kette der dafür nötigen Verarbeitungsschritte weiter ausgearbeitet, siehe Abb. 6.1 auf Seite 32. Die Schritte Hamming-Fensterung, FFT und Mel-Filter-Anwenden werden in den Abschnitten 2.1.1, 2.1.2 und 2.1.5 genauer erläutert. Für die Schritte Betrag-Nehmen, Logarithmieren und DCT fehlt noch das Ausarbeiten einer Begründung ihres Platzes in der Verarbeitungskette. Das Vorvorgängerprojekt von Hagen u.a. [Hag22, Kap. 2.2.1 auf Seite 19–20 und Kap. 5.3 auf Seite 30 oben] verweist für diese drei Schritte zumindest kurz auf die Literatur. (Siehe auch Abschnitt 2.1.4 im aktuellen Bericht zu einer Beschreibung der DCT an sich.)

Für die Schritte Hamming-Fensterung, Betrag-Nehmen und DCT wurde eine Implementierung in VHDL versucht, siehe Kap. 6.2, die aber nicht gelang. Für die FFT wurde ein nicht näher genannter IP-Block für diese Aufgabe dem Overlay für das FPGA hinzugefügt, aber noch nicht weiter verwendet oder getestet.

In der Folge wurde eine Implementierung der gesamten Verarbeitungskette (Hamming-Fensterung, FFT, Betrag-Nehmen, Mel-Filter-Anwenden, Logarithmieren und DCT) in der Sprache C++ versucht, um diesen Code mit dem Werkzeug Xilinx Vitis HLS automatisiert in VHDL übersetzen zu lassen (siehe Kap. 6.3). Die FFT wurde dabei wie zuvor mit dem fertigen IP-Block realisiert. Dieser Ansatz führte zu einem erfolgreichen Synthetisieren des Codes. Für Tests und ein Einbinden in das Gesamtprojekt fehlte die Zeit. Bisher ist also nur die syntaktische Korrektheit bestätigt.

Beim Betrag-Nehmen von komplexen Zahlen erzeugte das nötige Wurzelziehen das Hindernis, dass diese Operation in VHDL sehr viele FPGA-Ressourcen benötigt. Daher wurde sie näherungsweise durch eine einfache Summe ersetzt. Hier bleibt zu untersuchen, was das für die Qualität des Ergebnisses bedeutet. In der C++-Version wurde die richtige Wurzel-Operation verwendet. Es bleibt zu untersuchen, was dies für die FPGA-Ressourcen bedeutet.

Ein Vergleich des Zeitverhaltens des IP-Blocks für die FFT und der Python-Numpy-Implementierung für die FFT deutet an, dass die Implementierung im FPGA schneller ist (siehe Kap. 9.2.2).

Architektur des neuronalen Netzes: Schon das Vorvorgängerprojekt realisierte ein funktionierendes neuronales Netz, das eine Erkennungsrate von 90 % lieferte (siehe [Hag22, Kap. 7.2]). Danach brachte das Vorgängerprojekt in die Architektur einige Verbesserungen ein, die die Erkennungsrate (vor der Quantisierung des Netzes) auf 95 % erhöhten (siehe [Bur23, Kap. 7.2.2]).

Dem aktuellen Projekt gelang das Einarbeiten in dieses neuronale Netz nicht. Stattdessen wurde eine ganz neue Architektur für das neuronale Netz entworfen.

Kernidee für die neue Architektur war, dass die gestellte Aufgabe eine Analyse einer Zeitreihe erfordert. Hierfür sind laut vorherrschender Ansicht rekurrente neuronale Netze (RNNs), speziell Long-Short-Term-

Memory-Netze (LSTM-Netze), gut geeignet. Entsprechend wurde eine Architektur mit einem LSTM-Teil in der Mitte entworfen.

Diese Kernidee übersieht, dass das neuronale Netz auf dem FPGA seine Daten nicht als langen, kontinuierlichen Datenstrom bekommt, sondern gemäß der Gesamt-Systemarchitektur (s.o.) zeitlich gefenstert jeweils in kleinen, festen 2 s-Blöcken. Entsprechend müssten diese Blöcke für das neuronale Netz zunächst zurück in jeweils einen kurzen sequenziellen Datenstrom konvertiert werden. Damit wäre der Vorteil eines LSTM-Netzes, automatisch einen Überblick über den Gesamtdatenstrom zu haben, verloren.

Weiterhin hatte bereits das Vorgängerprojekt herausgefunden, dass das FINN-Framework, mit dem das neuronale Netz quantisiert und auf das FPGA gebracht werden soll, wahrscheinlich ausschließlich Feed-Forward-Netze unterstützt, also keine LSTM-Netze (siehe [Bur23, Kap. 2.2]).

Trotzdem führt das beschriebene Schnittstellenproblem zu der neuen Einsicht, dass es günstiger sein könnte, die zweite zeitliche Fensterung in der Gesamt-Architektur entfallen zu lassen und ein rekurrentes neuronales Netz, z.B. ein LSTM-Netz, auf einen kontinierlichen Strom von MFCC-Koeffizienten anzuwenden.

Dies kann die Effizienz steigern, indem eine doppelte Bearbeitung jedes MFCC-Koeffizienten vermieden wird. Bei dem gefensterten Ansatz ergibt sich das Problem, dass ein Sprachkommando quer über eine Grenze zwischen den 2 s-Fenstern zu liegen kommen kann. Die Lösung dafür ist, das Fenster immer nur jeweils 1 s weiterzuschieben, um das Sprachkommando wenigstens einmal voll im Fenster zu haben. Der Preis für diese Lösung ist, dass jeder einzelne MFCC-Koeffizient dadurch zweimal vom neuronalen Netz verarbeitet werden muss. Die Verarbeitung eines kontinuierlichen Stroms durch ein neuronales Netz mit „Gedächtnis“, wie es RNNs sind, vermeidet diese doppelte Bearbeitung. Ob ein LSTM-Netz ansonsten effizienter oder weniger effizient als die bisher verwendeten gefalteten neuronalen Netze (CNNs) ist, bleibt allerdings zu untersuchen. Voraussetzung für diesen Ansatz ist, dass das FINN-Framework doch LSTM-Netze verarbeiten kann, oder entsprechend erweitert oder ersetzt werden kann.

Training des neuronalen Netzes: Das Vorvorgängerprojekt erzeugte einen Trainingsdatensatz mit Sprachkommandos (siehe [Hag22, Kap. 7.1]). Dabei wurden die Audioschnipsel auf eine einheitliche Länge gebracht, indem sie hinten mit Nullen aufgefüllt wurden. Das Vorgängerprojekt verbesserte dies, indem das eigentliche Sprachkommando auf eine zufällige Position im festen Zeitrahmen gelegt wurde, um keine Position bei ihrer Relevanz fälschlich zu bevorzugen (siehe [Bur23, Kap. 5.2.4]).

Das aktuelle Projekt berücksichtigte diese vorhandene Vorverarbeitung nicht, sondern setzte sie in gleichartiger Weise noch einmal neu um.

Das Training mit dem selbstentworfenen neuronalen Netz war nicht erfolgreich. Bevor ein grundlegendes Training geschafft war, wurden bereits einige Optimierungsansätze integriert. Nachdem das System aus neuronalem Netz und Trainingsumgebung nicht ausgeführt werden konnte, war eine Beurteilung des Nutzens nicht möglich.

Als Ersatz wurde kurzfristig versucht, ein anderes Modell für Spracherkennung aus einem Tutorial zu trainieren. Auch dies war nicht erfolgreich.

Quantisieren und Übersetzen des neuronalen Netzes mit dem FINN-Compiler für das FPGA Das Vorgängerprojekt erarbeitete bereits eine ausführliche Anleitung, wie man mit dem FINN-Compiler ein neuronales Netz quantisiert, übersetzt und auf das FPGA bringt (siehe [Bur23, Kap. 6]).

Das aktuelle Projekt berücksichtigte diese vorhandene Anleitung nicht, sondern erarbeitete eine ähnliche, ähnlich ausführliche Anleitung erneut (siehe Kap. 8.1 und Kap. 8.3). Dabei wurde die nun aktuelle Version des Werkzeugs Vivado verwendet. Das Einfügen des erzeugten IP-Blocks in das Gesamtprojekt war nicht erfolgreich (siehe Kap. 8.4).

Kapitel 11

Ausblick

geschrieben von Jan Bredereke

Dieses Kapitel fasst den aktuellen Stand der Ideen zusammen, wie die laufende Serie von Mini-Projekten fortgesetzt werden kann.

11.1 Systemarchitektur

Kapitel 10 oben verweist auf die nun detaillierter ausgearbeitete Systemarchitektur für eine Verarbeitung komplett auf einem FPGA. Es sollte zusammengetragen werden, für welche Komponenten inzwischen bekannt ist, dass sie nicht zeitkritisch sind. Für diese Teile sollte untersucht werden, ob sie sinnvoll auf die CPU des Boards ausgelagert werden können, um den doch beträchtlichen Implementierungsaufwand für eine FPGA-Lösung zu sparen. Die nachfolgenden Abschnitte nennen hierzu bereits teilweise Erkenntnisse.

11.2 Sprachkommandoerkennung

Eine neue Idee, um die Effizienz der Sprachkommandoerkennung zu steigern, ist, gezielt einen für diese spezielle Aufgabe besonders geeigneten Klassifikator für einen Frequenzkoeffizientenstrom zu suchen. Bisher wurden nur gefaltete neuronale Netze (CNNs) verwendet. In Kapitel 10 wurde bereits dikutiert, dass Long-Short-Term-Memory-Netze (LSTM-Netze) grundsätzlich besser geeignet sein könnten, um eine Zeitreihenanalyse wie die Sprachkommandoerkennung effizient durchzuführen. Voraussetzung dafür dürfte sein, dass sich auch diese Netze quantisieren lassen. Insbesondere Werkzeugunterstützung dürfte wichtig sein. Dies kann recherchiert, prototypisch ausprobiert und bei Erfolg tiefer weiterverfolgt werden.

Auch der Blick auf ganz andere Klassifikatoren könnte Fortschritte bringen, z.B. auf die sich derzeit sehr stark entwickelnden Transformer. Sie erscheinen ebenfalls für eine Zeitreihenanalyse sehr geeignet. In einem ersten Schritt wäre zu klären, welche Rechenoperationen dort zu Grunde liegen, und ob sie sich, insbesondere mit Quantisierung, grundsätzlich effizient auf ein FPGA bringen lassen.

Im übrigen sollten die in der aktuellen Arbeit direkt offengebliebenen Aufgabenstellungen weiter verfolgt werden:

Der Analog-Digital-Wandler auf dem PYNQ-Z2-Board sollte so tief verstanden werden, dass er gezielt in eigene FPGA-Projekte eingebunden werden kann.

Für die Transformation der rohen Audiodaten in Mel-Frequenz-Cepstrum-Koeffizienten sollten die nötigen Schritte noch klarer beschrieben und ggf. ausgewählt werden (vergleiche Kap 10). Anschließend sollten diese Schritte in VHDL umgesetzt oder mit fertigen IP-Blöcken realisiert werden. Eine Umsetzung auf abstrakterer Ebene, z.B. mit C++ in der Vitis-Entwicklungsumgebung ist zwar ebenfalls denkbar. Allerdings dürfte es dann schwieriger werden, zu einer auch effizienten Implementierung zu kommen, weil mehr Details durch Abstraktion unsichtbar würden. Entsprechend dürfte es aufwändiger werden,

ineffiziente Teile des generierten Codes zu identifizieren und zu verbessern.

11.3 Gestenerkennung

Außer der Sprachkommandoerkennung für das autonome Fahrzeug wurde bereits eine Steuerung durch Gesten untersucht und recht weit umgesetzt. Altnickel u.a. [Alt22, Kap.7.2] beschreiben ausführlich, was hier noch zu tun ist. Einige Idee von dort sind:

- Einbinden der Kachelung (Tiling) per Hardware in das Gesamtsystem, die in der Bachelorarbeit von Müller [Mül21] realisiert wurde.
- Integration der Gestenerkennung in das Gesamtsystem.
- Verwenden eines neuronalen Netzes speziell für Single-Class-Detection, also eines, das nur ja/nein-Antworten für die Personenerkennung gibt.
- Implementieren von wenig zeitkritischen neuronalen Netzen, z.B. zur Gestenerkennung nach erfolgter Personenerkennung, auf der CPU des Systems. Dafür machen die Autoren bereits eine Reihe von Vorschlägen für mögliche Software-Frameworks. Hierfür müsste die aktuelle Systemarchitektur entsprechend angepasst werden, vergleiche im aktuellen Bericht Kap. 11.1 oben.
- Verwenden bestimmter, konkret genannter Layer in einem neuronalen Netz, um es noch genauer an seine Aufgabe anzupassen.
- Diverse Verbesserungen beim Training.
- Analyse, warum der Ende-zu-Ende-Systemtest nicht ganz so gute Erkennungsergebnisse lieferte wie das Validierungsskipt.

Im idealen Endausbau wäre das Fahrzeug sowohl durch Gesten als auch durch Sprachkommandos zu steuern.

Darüber hinaus kann die Idee aus Kap. 11.2 oben, andere Klassifikatoren zu verwenden, auch auf die Bilderkennung angewandt werden. Insbesondere können wir die Beobachtung übertragen, dass Sprachsignale in einer zeitlichen Reihenfolge eintreffen und eine zeitliche Abhängigkeit haben.

Bisher wurden bei der Bilderkennung alle Bilder separat behandelt. Aber auch im zeitlichen Strom von Bilddaten gibt es starke zeitliche Abhängigkeiten. Meist sieht ein Bild recht ähnlich wie das vorherige Bild aus. Daher wäre es nicht nötig, dieses Bild von Grund auf neu zu analysieren, um die zu erkennende Person und ggf. ihre Geste zu finden. Ist eine Person einmal gefunden, sollte sie im nächsten Bild zunächst an der selben Stelle und dann in der unmittelbaren Umgebung gesucht werden. Schon bisher wurde die Bildanalyse in eine erste, grobe Stufe der Personenerkennung und in eine zweite, feine Stufe der Erkennung der Geste dieser Person aufgeteilt. Dabei musste aus Ressourcengründen für die Personenerkennung das Bild in Kacheln („Tiles“) aufgeteilt werden. Es liegt nun nahe, dass im Folgebild zunächst Kacheln analysiert werden, die dicht bei der Position der letzten erkannten Person liegen. Sobald eine Person gefunden wurde, kann die Suche abgebrochen werden. Dies dürfte den durchschnitlichen Ressourcenverbrauch für die Personensuche ganz massiv reduzieren.

Darüber hinaus kann recherchiert werden, ob es nicht auch für die Bilderkennung bereits fertige Klassifikatoren gibt, die speziell für Bildströme, also Video, optimiert sind.

11.4 Fahrsteuerung

Wie schon Altnickel u.a. [Alt22, Kap.7.2.3] bemerkten, wurde die einst entwickelte Fahrsteuerung, die an Hand der erkannten Personenposition und der erkannten Gesten das Fahrzeug steuert, nie richtig getestet und funktioniert vermutlich auch nicht richtig.

Diese Teilfunktion könnte unabhängig von allen Kommandoerkennungsfragen separat überarbeitet und zum zuverlässigen Funktionieren gebracht werden. Es wäre lediglich eine Schnittstelle mit abstrakten Kommandosignalen zu definieren, die den Gestenkommandos, oder auch Sprachkommandos, entsprechen. Diese Kommandos könnten für Testzwecke ebensogut mit einer einfachen handelsüblichen Fernsteuerung gegeben werden.

Bekannt ist auch ([Alt22, Kap.7.2.6]), dass der „Search Mode“, in dem neu nach einer Person in der Umgebung gesucht wird, derzeit eher zu simplistisch agiert. Das könnte verbessert werden.

Die Fahrsteuerung sollte vorzugsweise in Software entwickelt werden, da sie wenig zeitkritisch ist. Hierfür müsste die aktuelle Systemarchitektur entsprechend angepasst werden, vergleiche Kap. 11.1 oben.

11.5 Elektronik und Mechanik der Fahrzeuge

An der Elektronik und Mechanik der Fahrzeuge sind die folgenden Dinge zu tun.

11.5.1 Integration eines PYNQ-Z2-Boards

Ein neues PYNQ-Z2-Board muss jeweils anstelle des PYNQ-Z1-Boards mechanisch und elektrisch in beide vorhandenen Fahrzeuge integriert werden.

11.5.2 Integration eines Mikrofons

Mit dem Abschied vom PYNQ-Z1-Board entfällt das darauf eingebaute einfache Mikrofon. Entsprechend muss ein gutes Mikrofon ausgewählt, mechanisch integriert und angeschlossen werden, für jedes der beiden Fahrzeuge.

Als Erweiterung kann darüber nachgedacht werden, ein Richtmikrofon auszuwählen oder aber mit zwei oder mehr Mikrofonen plus passender Verarbeitung eine Richtcharakteristik zu erreichen.

Es ist zu bedenken, dass Fahrgeräusche das Hörvermögen des Fahrzeugs beeinträchtigen werden. Einerseits könnte hier eine mechanische Isolierung gegen Körperschall entwickelt werden. Andererseits könnte, neben einer Richtcharakteristik für das Mikrofon, eine Fahrsteuerung mit Lauschpausen zum Einsatz kommen.

11.5.3 Verbesserung der Motorsteuerung

Altnickel u.a. [Alt22, Kap.7.2.6] merken an, dass die aktuellen Motoren recht ungenau arbeiten. Das Fahrzeug fährt zum Beispiel nicht einfach genau geradeaus. Weiterhin können die Motoren nicht sehr langsam fahren, was zu verwischten Kamerabildern führt. Auch ein Gimbal für die Kamera wird angedacht. Hier dürfte einiges Potential für Verbesserungen sein.

Kapitel 12

Quellen

[Agg18] Charu C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*. Springer, 2018. ISBN: 978-3-319-94463-0. DOI: 10.1007/978-3-319-94463-0. URL: https://link.springer.com/book/10.1007/978-3-319-94463-0.

[Alt21] Philipp Altnickel u. a. *Gesten- und Objekterkennung durch schwache FPGAs in autonomen Fahrzeugen mittels neuronaler Netze*. Techn. Ber. Hochschule Bremen, 1. Sep. 2021. 153 S. URL: https://homepages.on.hs-bremen.de/~jbredereke/de/forschung/veroeffentlichungen/gestenerkennung-fpga-neuronale-netze-projekt-21.html (besucht am 30. 03. 2022).

[Alt22] Philipp Altnickel u. a. *Personenerkennung durch schwache FPGAs in autonomen Fahrzeugen mittels Neuronaler Netze*. Techn. Ber. Hochschule Bremen, 1. März 2022. 57 S. URL: https://homepages.on.hs-bremen.de/~jbredereke/de/forschung/veroeffentlichungen/personenerkennung-fpga-neuronale-netze-projekt-2122.html (besucht am 17. 03. 2025).

[AMDa] AMD. *AUP PYNQ-Z2*. URL: https://www.amd.com/de/corporate/university-program/aup-boards/pynq-z2.html (besucht am 12. 02. 2025).

[AMDb] AMD. *Downloads*. URL: https://www.xilinx.com/support/download.html (besucht am 16. 01. 2025).

[AMD22a] AMD. *AXI Interconnect v2.1 LogiCORE IP Product Guide*. Englisch. 17. Mai 2022. URL: https://docs.amd.com/r/en-US/pg059-axi-interconnect/AXI-Interconnect-v2.1-LogiCORE-IP-Product-Guide (besucht am 01. 02. 2025).

[AMD22b] AMD. *PYNQ Online Documentation and Guide*. Englisch. 11. Okt. 2022. URL: https://pynq.readthedocs.io/en/latest/index.html (besucht am 26. 01. 2025).

[AMD22c] AMD. *PYNQ Overlay*. Englisch. 11. Okt. 2022. URL: https://pynq.readthedocs.io/en/latest/pynq_overlays.html (besucht am 20. 02. 2025).

[AMD22d] AMD. *SmartConnect v1.0 LogiCORE IP Product Guide*. Englisch. 19. Okt. 2022. URL: https://docs.amd.com/r/en-US/pg247-smartconnect/SmartConnect-v1.0-LogiCORE-IP-Product-Guide (besucht am 31. 01. 2025).

[AMD23] AMD. *AXI Basics Series*. Englisch. 21. Feb. 2023. URL: https://adaptivesupport.amd.com/s/topic/0TO2E000000YNxCWAW/axi-basics-series?language=en_US&tabset-50c42=2&tabset-b221e=2 (besucht am 26. 01. 2025).

[AMD24a] AMD. *AMD Technical Information Portal — docs.amd.com*. Nov. 2024. URL: https://docs.amd.com/r/en-US/pg109-xfft/Introduction (besucht am 07. 02. 2025).

[AMD24b] AMD. *AMD Technical Information Portal — docs.amd.com*. Nov. 2024. URL: https://docs.amd.com/r/en-US/pg109-xfft/Features (besucht am 07. 02. 2025).

[AMD24c] AMD. *AMD Technical Information Portal — docs.amd.com*. Nov. 2024. URL: https://docs.amd.com/r/en-US/pg109-xfft/IP-Facts (besucht am 07. 02. 2025).

[AMD24d] AMD. *AMD Technical Information Portal — docs.amd.com*. Nov. 2024. URL: https://docs.amd.com/r/en-US/pg109-xfft/ (besucht am 07. 02. 2025).

[AMD24e] AMD. *AXI DMA LogiCORE IP Product Guide (PG021)*. Englisch. 20. Juni 2024. URL: https://docs.amd.com/r/en-US/pg021_axi_dma/Introduction (besucht am 31. 01. 2025).

[AMD24f] AMD. *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing (UG973)*. 2024. URL: https://docs.amd.com/r/en-US/ug973-vivado-release-notes-install-license/Cable-Installation-Requirements (besucht am 11. 01. 2025).

[arm] arm. *Advanced Microcontroller Bus Architecture (AMBA) Specifications*. Englisch. URL: https://developer.arm.com/Architectures/AMBA (besucht am 26. 01. 2025).

[arm20] arm. *AMBA AXI and ACE Protocol Specification*. Englisch. 31. März 2020. URL: https://developer.arm.com/documentation/ihi0022/h/?lang=en (besucht am 26. 01. 2025).

[arm21] arm. *AMBA AXI-Stream Protocol Specification*. Englisch. 9. Apr. 2021. URL: https://developer.arm.com/documentation/ihi0051/latest/ (besucht am 26. 01. 2025).

[Aut] IEEE P1076 WG Authors. *Github Fixed Float Types*. URL: https://github.com/ghdl/ghdl/blob/master/libraries/ieee2008/fixed_float_types.vhdl (besucht am 13. 02. 2025).

[BOU] ANNA BOUNCHALEUN. „AN ELEMENTARY INTRODUCTION TO FAST FOURIER TRANSFORM ALGORITHMS“. In: (). URL: https://math.uchicago.edu/~may/REU2019/REUPapers/Bounchaleun.pdf.

[Bre23] Jan Bredereke. „Enabling Neural Network Edge Computing on a Small Robot Vehicle“. Englisch. In: *Intelligent Distributed Computing XV* (Bremen, Germany, 14.–15. Sep. 2022). Hrsg. von Lars Braubach, Kai Jander und Costin Bădică. Bd. 1089. Studies in Computational Intelligence. Springer, 2023, S. 33–40. ISBN: 978-3-031-29103-6. DOI: 10.1007/978-3-031-29104-3_4. URL: https://homepages.on.hs-bremen.de/~jbredereke/de/forschung/veroeffentlichungen/bredereke-enabling-nn-edge-computing-on-robot-2022.html (besucht am 27. 04. 2023).

[Bur23] Jarno Burggräf u. a. *Neuronale Netze auf einem FPGA zur Sprachkommandoerkennung*. Techn. Ber. Hochschule Bremen, 8. März 2023. 55 S. URL: https://homepages.on.hs-bremen.de/~jbredereke/de/forschung/veroeffentlichungen/neuronale-netze-fpga-sprachkommandoerkennung-wp_embeds-2023.html (besucht am 17. 03. 2025).

[Cre] Stack Overflow Creators. *ffmpeg - Free unaffiliated eBook created from Stack Overflow Creators*. URL: https://riptutorial.com/Download/ffmpeg-de.pdf (besucht am 30. 01. 2025).

[Dev] Analog Devices. *ADAU1761*. URL: https://www.analog.com/en/products/adau1761.html (besucht am 12. 02. 2025).

[Dev22] Advanced Micro Devices. *PYNQ Introduction*. 2022. URL: https://pynq.readthedocs.io/en/latest/ (besucht am 12. 02. 2025).

[Dev24] Analog Devices. *Data Sheet ADAU1761*. Englisch. 2024. URL: https://www.analog.com/media/en/technical-documentation/data-sheets/ADAU1761.pdf (besucht am 12. 02. 2025).

[Doca] Docker. *Docker Engine*. URL: https://docs.docker.com/engine/ (besucht am 17. 02. 2025).

[Docb] Docker. *Install Docker Engine on Ubuntu*. URL: https://docs.docker.com/engine/install/ubuntu/ (besucht am 11. 01. 2025).

[Docc] Docker. *Linux post-installation steps for Docker Engine*. URL: https://docs.docker.com/engine/install/linux-postinstall/#manage-docker-as-a-non-root-user (besucht am 11. 01. 2025).

[Ele18] Mouser Electronics. *PYNQ-Z2 Reference Manual v1.0*. Englisch. 17. Mai 2018. URL: https://www.mouser.com/datasheet/2/744/pynqz2_user_manual_v1_0-1525725.pdf?srsltid=AfmBOooD7IDYehdUtbUeAXXvqWXmLeVMBb_e4y4RZIp18nJioR7d3LwD (besucht am 12. 02. 2025).

[FINa] FINN. *Building the Streaming Dataflow Accelerator*. URL: https://github.com/Xilinx/finn/blob/main/notebooks/end2end_example/cybersecurity/3-build-accelerator-with-finn.ipynb (besucht am 11. 02. 2025).

[FINb] FINN. *End-to-End Flow*. URL: https://finn.readthedocs.io/en/latest/end_to_end_flow.html (besucht am 30. 01. 2025).

[FINc] FINN. *FINN*. URL: https://xilinx.github.io/finn/ (besucht am 19. 01. 2025).

[FIN24] FINN. *Getting Started*. 2024. URL: https://finn.readthedocs.io/en/latest/getting_started.html (besucht am 11. 01. 2025).

[GBC16] Ian Goodfellow, Yoshua Bengio und Aaron Courville. *Deep Learning*. MIT Press, 2016. ISBN: 978-0-262-03561-3. URL: https://www.deeplearningbook.org/.

[Gér19] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media, 2019. ISBN: 978-1-492-03264-9. URL: https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/.

[Git] GitLab. *Embeds Projekt*. URL: https://gitlab.com/embedsprojekt_wise22/ki/-/blob/394a9190059111edb86ebf983435bebd783f94e1/old/ki-main/finnModel/DefaultModel.onnx (besucht am 12. 02. 2025).

[GR23] Steffen Goebbels und Stefan Ritter. „Diskrete Fourier-Transformation". In: *Mathematik verstehen und anwenden: Differenzialgleichungen, Fourier- und Vektoranalysis, Laplace-Transformation und Stochastik*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2023, S. 383–439. ISBN: 978-3-662-68369-9. DOI: 10.1007/978-3-662-68369-9_14. URL: https://doi.org/10.1007/978-3-662-68369-9_14.

[Hag22] Fynn Hagen u. a. *Neural Network on an FPGA for Speech Command Recognition on an Autonomous Vehicle*. Englisch. Techn. Ber. Hochschule Bremen, 8. März 2022. 49 S. URL: https://homepages.on.hs-bremen.de/~jbredereke/de/forschung/veroeffentlichungen/speech-recognition-fpga-neural-network-wp_embeds-2022.html (besucht am 17. 03. 2025).

[Her24] Hector Gerardo Muñoz Hernandez u. a. „A Novel System Simulation Framework for HBM2 FPGA Platforms". In: *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Springer Nature Switzerland AG, 2024, S. 72–84. DOI: 10.1007/978-3-031-78380-7_7. URL: https://link.springer.com/chapter/10.1007/978-3-031-78380-7_7.

[Hut20] Colin von Huth u. a. *Bericht zum Projekt ‚Neuronale Netze auf strahlungstoleranten FPGAs für die Raumfahrt'*. Techn. Ber. Hochschule Bremen, 14. Feb. 2020. 88 S. URL: https://homepages.on.hs-bremen.de/~jbredereke/de/forschung/veroeffentlichungen/neuronale-netze-fpgas-projekt-1920.html (besucht am 30. 03. 2022).

[IBM24] IBM. *Was ist Quantisierung?* 29. Juli 2024. URL: https://www.ibm.com/de-de/think/topics/quantization (besucht am 12. 02. 2025).

[Ins] National Instruments. *Understanding FFTs and Windowing*. URL: https://download.ni.com/evaluation/pxi/Understanding%5C%20FFTs%5C%20and%5C%20Windowing.pdf (besucht am 13. 02. 2025).

[La19] Frank La La. *Wie lernen neuronale Netze?* 2019. URL: https://learn.microsoft.com/de-de/archive/msdn-magazine/2019/april/artificially-intelligent-how-do-neural-networks-learn (besucht am 12. 02. 2025).

[Lina] Linux Mint. *Linux Mint 22.1*. URL: https://linuxmint.com/ (besucht am 30. 01. 2025).

[Linb] Linux Mint. *Linux Mint 22.1 "Xia"*. URL: https://linuxmint.com/download.php (besucht am 16. 01. 2025).

[Linc] Linux Mint Installation Guide. *Linux Mint Installation Guide*. URL: https://linuxmint-installation-guide.readthedocs.io/en/latest/ (besucht am 30. 01. 2025).

[Log] Beth Logan. *Mel Frequency Cepstral Coefficients for Music Modelling*. URL: https://ismir2000.ismir.net/papers/logan_paper.pdf (besucht am 13. 02. 2025).

[Mem] balkris (Member). *AMD Forum*. URL: https://adaptivesupport.amd.com/s/question/0D52E00006hpbL3SAI/cannot-find-fixedpkg-in-ieee-or-ieeeproposed?language=en_US (besucht am 13. 02. 2025).

[Mer23] Alfred Mertins. „Diskrete Blocktransformationen". In: *Signaltheorie: Grundlagen der Signalbeschreibung, Filterbänke, Wavelets, Zeit-Frequenz-Analyse, Parameter- und Signalschätzung*. Wiesbaden: Springer Fachmedien Wiesbaden, 2023, S. 183–208. ISBN: 978-3-658-41529-7. DOI: 10.1007/978-3-658-41529-7_5. URL: https://doi.org/10.1007/978-3-658-41529-7_5.

[Mica] Microsoft. *BitLocker-Wiederherstellungsschlüssel*. URL: https://account.microsoft.com/devices/recoverykey (besucht am 16. 01. 2025).

[Micb] Microsoft. *Find your BitLocker recovery key*. URL: https://support.microsoft.com/en-us/windows/find-your-bitlocker-recovery-key-6b71ad27-0b89-ea08-f143-056f5ab347d6 (besucht am 16. 01. 2025).

[Mis23] Pradeepta Mishra. *PyTorch Recipes: A Problem-Solution Approach to Build, Train and Deploy Neural Network Models*. Second Edition. Apress, 2023. ISBN: 978-1-4842-8925-9. DOI: 10.1007/978-1-4842-8925-9. URL: https://link.springer.com/book/10.1007/978-1-4842-8925-9.

[Mül21] Felix Müller u. a. *Applying Binarized Neural Networks on FPGAs to an Autonomous Driving Problem*. Englisch. Techn. Ber. Hochschule Bremen, 31. März 2021. 50 S. URL: https://homepages.on.hs-bremen.de/~jbredereke/de/forschung/veroeffentlichungen/bnns-on-fpgas-driving-projekt-2021.html (besucht am 30. 03. 2022).

[Mül21] Felix Müller. „Dynamisches Tiling auf schwachen FPGAs zur Objekterkennung mithilfe kleiner neuronaler Netze". Bachelorthesis. Hochschule Bremen, 23. Juni 2021. URL: https://homepages.on.hs-bremen.de/~jbredereke/de/forschung/veroeffentlichungen/mueller-bsc-thesis-2021.html (besucht am 17. 03. 2025).

[MV24] A. Lakshmi Muddana und Sandhya Vinayakam. *Python for Data Science*. Springer Nature Switzerland AG, 2024. ISBN: 978-3-031-52473-8. DOI: 10.1007/978-3-031-52473-8. URL: https://link.springer.com/chapter/10.1007/978-3-031-52473-8_1.

[Net] Netron. *Netron*. URL: https://github.com/lutzroeder/netron (besucht am 11. 02. 2025).

[ONN] ONNX. *Open Neural Network Exchange*. URL: https://onnx.ai/ (besucht am 19. 01. 2025).

[PyTa] PyTorch. *Exporting a Model from PyTorch to ONNX and Running it using ONNX Runtime*. URL: https://pytorch.org/tutorials/advanced/super_resolution_with_onnxruntime.html (besucht am 11. 02. 2025).

[PyTb] PyTorch. *Install PyTorch*. URL: https://pytorch.org/ (besucht am 11. 02. 2025).

[PyTc] PyTorch. *Speech Command Classification with torchaudio*. URL: https://pytorch.org/tutorials/intermediate/speech_command_classification_with_torchaudio_tutorial.html (besucht am 10. 02. 2025).

[ROCa] ROCm. *Quick start installation guide*. URL: https://rocm.docs.amd.com/projects/install-on-linux/en/latest/install/quick-start.html (besucht am 11. 02. 2025).

[ROCb] ROCm. *System requirements (Linux)*. URL: https://rocm.docs.amd.com/projects/install-on-linux/en/latest/reference/system-requirements.html (besucht am 12. 02. 2025).

[Ruf] Rufus. *Rufus*. URL: https://rufus.ie/de/ (besucht am 16. 01. 2025).

[Sch] Florian Schiel. *Einführung in die Signalverarbeitung Phonetik und Sprachverarbeitung, 2. Fachsemester, Block Sprachtechnologie I*. URL: https://www.phonetik.uni-muenchen.de/studium/skripten/P4.1_EinfuehrungSignalverarbeitung/EinfSignalverarbeitung_5.pdf (besucht am 13. 02. 2025).

[Sid23] Siddiqi. *pydub Tutorial: Audio Manipulation in Python*. 2023. URL: https://coderslegacy.com/pydub-tutorial-audio-manipulation-in-python/ (besucht am 30. 01. 2025).

[SLS24] Manjit Singh Sidhu, Nur Atiqah Abdul Latib und Kirandeep Kaur Sidhu. „MFCC in audio signal processing for voice disorder: a review“. In: *Multimedia Tools and Applications* (Apr. 2024). ISSN: 1573-7721. DOI: 10.1007/s11042-024-19253-1. URL: https://doi.org/10.1007/s11042-024-19253-1.

[SS24] Dario Salice und Jennifer Salice. „Voice“. In: *Foundations and Opportunities of Biometrics: An Introduction to Technology, Applications, and Responsibilities*. Berkeley, CA: Apress, 2024, S. 21–36. ISBN: 979-8-8688-0509-7. DOI: 10.1007/979-8-8688-0509-7_7. URL: https://doi.org/10.1007/979-8-8688-0509-7_7.

[Tea] Jupyter Team. *Jupyter Notebook*. Englisch. URL: https://docs.jupyter.org/en/latest/ (besucht am 20. 02. 2025).

[Xil] Xilinx. *finn-base*. URL: https://github.com/Xilinx/finn-base/blob/dev/src/finn/util/basic.py#L41 (besucht am 11. 02. 2025).

Anhang A

Struktur der Repositories

geschrieben von Jesse Gollub

Im GitLab der Hochschule wurden die folgenden Repositories erstellt:

- Protokolle
- library
- ki
- pl-definition
- howto

A.1 Protokolle

In diesem Repository wurden die Protokolle der Präsenzveranstaltungen und die Präsentationen, die aus Recherchen entstanden sind gesammelt.

A.2 library

- Enthält Code für die FFT-Analyse, Audioaufnahme und die Erstellung eines IP-Blocks mit dem FINN-Framework.
- MFCC: Python-Implementierung zur Berechnung von MFCC und FPGA-Implementierungen für Teile dieser Berechnung.
- FINN Framework: Verarbeitet quantisierte neuronale Netze und erstellt IP-Blöcke. sowie das DefaultModel.onnx und Zwischenergebnisse bei der Erstellung eines IP-Blocks.
- Audioaufnahme: Zeitliche Fensterung zur kontinuierlichen Aufnahme von Audiodaten mit Python

Dieses Repository enthält den Code für die FFT-Analyse, die Audioaufnahme, sowie den Code zur Erstellung eines IP-Blocks mit dem FINN-Framework

A.3 ki

Dieses Repository beinhaltet Code und Modelle für das Trainieren von neuronalen Netzen mit Pytorch. Setzt auf externe Abhängigkeiten wie ffmpeg und PyDub für Audioverarbeitung.

A.4 pl-definition

Dieses Repository enthält das Haupt-Vivado-Projekt mit einem MainDesign, das AXI-Blöcke verbindet. Außerdem gibt es mehrere IP-Blöcke im `ip_repo`-Verzeichnis, welche ein AXI-Block und ein neuronales Netzwerk aus dem FINN Framework enthält.

A.5 howto

In diesem Repository liegen Hilfedateien zur Nutzung der verwendeten Soft- und Hardware.

Anhang B

Quellcode

B.1 MFCC

B.1.1 Python

fft_visualized.py

```python
# Author: David Schulz

import pyqtgraph as pg
from PySide6 import QtCore, QtWidgets
import numpy as np
import pyaudio


CHUNK = 1024
FORMAT = pyaudio.paFloat32
CHANNELS = 1
RATE = 44100
RECORD_SECONDS = CHUNK / RATE

p = pyaudio.PyAudio()

stream = p.open(
    format=FORMAT,
    channels=CHANNELS,
    rate=RATE,
    input=True,
    frames_per_buffer=CHUNK
)


class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()

        self.mainbox = QtWidgets.QWidget()
        self.setCentralWidget(self.mainbox)

        self.layout = QtWidgets.QVBoxLayout(self.mainbox)
        self.canvas = pg.GraphicsLayoutWidget()

        self.plot_graph = pg.PlotWidget()
        self.plot_graph.setBackground("w")
        pen = pg.mkPen(color=(0, 0, 255))
        self.plot_graph.setYRange(-.25, .25, padding=0.1)
        self.time = np.arange(0, RECORD_SECONDS, 1/RATE)
        data = stream.read(CHUNK, exception_on_overflow=False)
```

```
        self.data = np.frombuffer(data, dtype=np.float32)
        # Get a line reference
        self.line = self.plot_graph.plot(
            self.time,
            self.data,
            name="Audio Data",
            pen=pen,
            symbol="o",
            symbolSize=4,
            symbolBrush="b",
        )

        self.plot_graph2 = pg.PlotWidget()
        self.plot_graph2.setBackground("w")
        self.plot_graph2.setXRange(0, 12000, padding=0.1)
        self.plot_graph2.setYRange(0, 10, padding=0.1)

        freq = np.fft.fftshift(np.fft.fftfreq(
            len(self.time), 1/RATE))[CHUNK//2:]
        data_fft = np.fft.fftshift(np.fft.fft(self.data))[CHUNK//2:]
        self.line2 = self.plot_graph2.plot(
            freq,
            np.abs(data_fft),
            name="Audio Data",
            pen=pen,
            symbol="o",
            symbolSize=4,
            symbolBrush="b",
        )

        self.layout.addWidget(self.plot_graph)
        self.layout.addWidget(self.plot_graph2)

        # Add a timer to simulate new data measurements
        self.timer = QtCore.QTimer()
        self.timer.setInterval(30)
        self.timer.timeout.connect(self.update_plot)
        self.timer.start()

    def update_plot(self):
        data = stream.read(CHUNK, exception_on_overflow=False)
        data_new = np.frombuffer(data, dtype=np.float32)
        data_fft = np.fft.fftshift(np.fft.fft(data_new))[CHUNK//2:]
        freq = np.fft.fftshift(np.fft.fftfreq(
            len(self.time), 1/RATE))[CHUNK//2:]
        self.data = data_new
        self.line.setData(self.time, self.data)
        self.line2.setData(freq, np.abs(data_fft))


app = QtWidgets.QApplication([])
main = MainWindow()
main.show()
app.exec()

stream.stop_stream()
stream.close()
p.terminate()
```

Listing B.1: Quellcode für die Visualisierung der FFT auf Audiodaten.

calc_mfcc.py

```
from typing import Union
```

```
2
3  import matplotlib.pyplot as plt
4  import numpy as np
5  from librosa import load, ex, feature
6  from scipy.fft import dct
7
8  """
9  This file is intended for testing calculating mfcc as such it is to be understood as a test zone.
10 For the actual final implementation see mfcc.py.
11 This file shows other ways to produce the filters which could be of interest when implementing in
       fpga.
12
13 AUTHOR: TOBIAS GUIMARAES ZIMMER
14 """
15
16
17 def segment_signal(signal: np.array, sample_rate: int, window_duration=20e-3, time_step=10e-3) ->
       np.array:
18     window_sample_amount = int(window_duration * sample_rate)
19     window_sample_hop_amount = int(time_step * sample_rate)
20
21     n_frames = int(len(signal) / window_sample_hop_amount)
22     # pad for last window
23     signal = np.pad(signal, window_sample_amount // 2, mode='reflect')
24
25     frames = np.zeros((n_frames, window_sample_amount))
26     # generate matrix of hamming function values
27     window_matrix = generate_hamming_window_matrix(window_sample_amount)
28
29     for n in range(n_frames):
30         frames[n] = signal[n * window_sample_hop_amount:n * window_sample_hop_amount +
       window_sample_amount]
31         frames[n] = window_matrix * frames[n]
32
33     return frames
34
35
36 def generate_hamming_window_matrix(window_length: int) -> np.array:
37     if window_length == 0:
38         return np.zeros(0)
39     returning_array = np.arange(0, window_length, 1)
40     # von Matlabs implementation:
41     return 0.54 - 0.46 * np.cos(2.0 * np.pi * returning_array / (window_length - 1))
42
43
44 def calc_nearest_pow_2(n: int) -> int:
45     return int(2 ** np.ceil(np.log2(n)))
46     # brute force variant:
47     # k = 1
48     # while n > 2 ** k:
49     #     k = k + 1
50     # return 2 ** k
51
52
53 def freq_to_mel(frequency: float) -> float:
54     # formula is: m = 2595 * log10(1 f/700)
55     return 2595.0 * np.log10(1.0 + frequency / 700.0)
56
57
58 def mel_to_freq(mel: Union[float, np.array]) -> float:
59     #  formula is:  f = 700 (10^(m/2595)-1)
60     return 700.0 * (10.0 ** (mel / 2595.0) - 1.0)
61
62
63 def get_filter_points(f_min: float, f_max: float,
64                       mel_filter_num: int, nfft: int, sample_rate: float) -> (np.array, np.array):
```

```
    # freq bounds to mel bounds
    min_mel = freq_to_mel(f_min)
    max_mel = freq_to_mel(f_max)

    # space them out
    # +2 since these points are the start, middle and stop points of the triangles
    mels = np.linspace(min_mel, max_mel, num=mel_filter_num + 2)
    # return to freq domain
    freqs = mel_to_freq(mels)

    # return frequencies and points converted from freq to index
    return np.floor((nfft + 1) / sample_rate * freqs).astype(int), freqs


def get_bartlett_window_matrix(left_bound: int, right_bound: int, filter: np.array) -> np.array:
    # doesn't always return 1 in middle
    # formula taken from matlab implementation
    L = right_bound - left_bound
    N = L - 1
    n = np.linspace(0, N, L)
    bartlett = np.where(np.less_equal(n, (N / 2)), 2 * n / N, 2 - 2 * n / N)
    filter[left_bound:right_bound] = bartlett
    return filter


def get_triang_window_matrix(left_bound: int, right_bound: int, filter: np.array) -> np.array:
    # broken i dunno why
    # formula taken from matlab implementation
    L = right_bound - left_bound
    N = L + 1
    n = np.linspace(1, N, L)
    if L % 2 != 0:
        triang = np.where(np.less_equal(n, (N / 2)), 2 * n / N, 2 - 2 * n / N)
    else:
        N = L
        triang = np.where(np.less_equal(n, (N / 2)), (2 * n - 1) / N, 2 - (2 * n - 1) / N)
    filter[left_bound:right_bound] = triang
    return filter


def get_filters(filter_points, nfft):
    # actual filter are - 2 and nfft/2 since only half the fft values are of relevance
    filters = np.zeros((len(filter_points) - 2, int(nfft / 2 + 1)))

    filters2 = np.zeros((len(filter_points) - 2, int(nfft / 2 + 1)))
    for n in range(len(filter_points) - 2):
        filters2[n] = get_bartlett_window_matrix(filter_points[n], filter_points[n + 2], filters[n
    ])

    filters3 = np.zeros((len(filter_points) - 2, int(nfft / 2 + 1)))
    for n in range(len(filter_points) - 2):
        filters3[n] = get_triang_window_matrix(filter_points[n], filter_points[n + 2], filters[n])

    for n in range(len(filter_points) - 2):
        filters[n, filter_points[n]: filter_points[n + 1]] = np.linspace(0, 1, filter_points[n + 1]
     - filter_points[n])
        filters[n, filter_points[n + 1]: filter_points[n + 2]] = np.linspace(1, 0, filter_points[n
    + 2] - filter_points[
            n + 1])

    return filters


y, sample_rate = load(ex('libri1'), sr=48000, duration=2)

windowed_matrix = segment_signal(y, sample_rate)
```

```
128
129 min_freq = 0
130 max_freq = sample_rate / 2
131 # has to be 2^n so dct via fft blocks becomes viable
132 mel_filter_num = 16
133
134 n_fft = calc_nearest_pow_2(len(windowed_matrix[0]))
135 fft_frames = np.copy(windowed_matrix)
136
137 # pad to 2^n if not (with zeros)
138 if n_fft != len(windowed_matrix[0]):
139     odd: int = 1 if n_fft - len(windowed_matrix[0]) % 2 != 0 else 0
140
141     fft_frames = np.pad(fft_frames,
142                         ((0, 0),
143                          ((n_fft - len(windowed_matrix[0])) // 2,
144                           (n_fft - len(windowed_matrix[0])) // 2 + odd)),
145                         mode='constant', constant_values=0)
146
147 fft_frames = np.fft.fft(fft_frames)
148
149 # get power spectrum
150 power_spectrum = np.abs(fft_frames) ** 2
151
152 # filter points (start and stop) and corresponding frequencies
153 filter_points, frequency = get_filter_points(min_freq, max_freq, mel_filter_num, n_fft, sample_rate
        )
154 filters = get_filters(filter_points, n_fft)
155
156 # enorm = 2.0 / (frequency[2:mel_filter_num+2] - frequency[:mel_filter_num])
157 # filters *= enorm[:, np.newaxis]
158
159 # for n in range(filters.shape[0]):
160 #     plt.plot(filters[n])
161 # plt.show()
162
163
164 # take right side of spektrum
165 right = power_spectrum[:, :len(power_spectrum[0]) // 2 + 1]
166 # right = power_spectrum[:, power_spectrum.shape[0]]
167
168 # faltung mit filtern
169 filtered_frames = np.dot(filters, right.T)
170
171 filtered_frames = 10.0 * np.log10(filtered_frames)
172
173 filtered_frames = filtered_frames.T
174 filtered_frames_copy = np.flip(filtered_frames, 1)
175
176 prepared_filtered_frames = np.append(filtered_frames, filtered_frames_copy, 1)
177
178 second_fft = np.fft.fft(prepared_filtered_frames).real
179 second_fft = second_fft[:, :int(len(second_fft[0]) / 2)].T
180 second_fft_dct = dct(filtered_frames).T
181
182 # they do some normalization here which cannot be disabled
183 lib_mfcc = feature.mfcc(y=y, sr=sample_rate, n_mfcc=16, dct_type=2, win_length=960, hop_length=480,
        n_fft=1024,
184                         window='hamm')
185 # go second way to disable it
186 lib_spec = feature.melspectrogram(y=y, sr=sample_rate, win_length=960, hop_length=480, n_fft=1024,
187                                   window='hamm', norm=None, center=True)
188 lib_spec = 10.0 * np.log10(lib_spec)
189 lib_mfcc = feature.mfcc(S=lib_spec, n_mfcc=16)
190
191 error = np.sum(np.abs(second_fft_dct - second_fft)) / (len(second_fft) * len(second_fft[0]))
```

```
192
193 plt.figure(1)
194 plt.subplot(311)
195 plt.imshow(lib_mfcc, aspect='auto', origin='lower')
196 plt.subplot(312)
197 plt.imshow(second_fft_dct, aspect='auto', origin='lower')
198 plt.subplot(313)
199 plt.imshow(second_fft, aspect='auto', origin='lower')
200 plt.show()
201 print(error)
```

Listing B.2: Quellcode in dem getestet wurde wie sich das MFCC berechnen lässt.

mfcc.py

```
1  from typing import Union
2
3  import numpy as np
4
5  """
6  This file is the library to be used on the board.
7  Just calling mfcc should be enough.
8
9  AUTHOR: Tobias Guimaraes Zimmer, David Schulz
10 """
11
12
13 def segment_signal(signal: np.array, sample_rate: int, window_duration=20e-3, time_step=10e-3) ->
       np.array:
14     window_sample_amount = int(window_duration * sample_rate)
15     window_sample_hop_amount = int(time_step * sample_rate)
16
17     n_frames = int(len(signal) / window_sample_hop_amount)
18     # pad for last window
19     signal = np.pad(signal, window_sample_amount // 2, mode='reflect')
20
21     frames = np.zeros((n_frames, window_sample_amount))
22     # generate matrix of hamming function values
23     window_matrix = generate_hamming_window_matrix(window_sample_amount)
24
25     for n in range(n_frames):
26         frames[n] = signal[n * window_sample_hop_amount:n * window_sample_hop_amount +
       window_sample_amount]
27         frames[n] = window_matrix * frames[n]
28
29     return frames
30
31
32 def generate_hamming_window_matrix(window_length: int) -> np.array:
33     if window_length == 0:
34         return np.zeros(0)
35     returning_array = np.arange(0, window_length, 1)
36     # taken from Matlabs implementation:
37     return 0.54 - 0.46 * np.cos(2.0 * np.pi * returning_array / (window_length - 1))
38
39
40 def calc_nearest_pow_2(n: int) -> int:
41     return int(2 ** np.ceil(np.log2(n)))
42
43
44 def freq_to_mel(frequency: float) -> float:
45     return 2595.0 * np.log10(1.0 + frequency / 700.0)
46
47
```

```
48 def mel_to_freq(mel: Union[float, np.array]) -> float:
49     return 700.0 * (10.0 ** (mel / 2595.0) - 1.0)
50
51
52 def get_filter_points(f_min: float, f_max: float,
53                       mel_filter_num: int, nfft: int, sample_rate: float) -> (np.array, np.array):
54     # freq bounds to mel bounds
55     min_mel = freq_to_mel(f_min)
56     max_mel = freq_to_mel(f_max)
57
58     # space them out
59     # +2 since these points are the start, middle and stop points of the triangles
60     mels = np.linspace(min_mel, max_mel, num=mel_filter_num + 2)
61     # return to freq domain
62     freqs = mel_to_freq(mels)
63
64     # return frequencies and points converted from freq to index
65     return np.floor((nfft + 1) / sample_rate * freqs).astype(int), freqs
66
67
68 def get_filters(filter_points, nfft):
69     # actual filter are - 2 and nfft/2 since only half the fft values are of relevance
70     filters = np.zeros((len(filter_points) - 2, int(nfft / 2 + 1)))
71
72     for n in range(len(filter_points) - 2):
73         filters[n, filter_points[n]: filter_points[n + 1]] = np.linspace(0, 1, filter_points[n + 1]
       - filter_points[n])
74         filters[n, filter_points[n + 1]: filter_points[n + 2]] = np.linspace(1, 0, filter_points[n
     + 2] - filter_points[
75             n + 1])
76
77     return filters
78
79
80 def mfcc(raw_audio: np.array, sample_rate: int) -> np.array:
81     windowed_matrix = segment_signal(raw_audio, sample_rate)
82     min_freq = 0
83     max_freq = sample_rate / 2
84     # has to be 2^n so dct via fft blocks becomes viable
85     mel_filter_num = 16
86
87     n_fft = calc_nearest_pow_2(len(windowed_matrix[0]))
88     fft_frames = np.copy(windowed_matrix)
89     # pad to 2^n if not (with zeros)
90     if n_fft != len(windowed_matrix[0]):
91         odd: int = 1 if n_fft - len(windowed_matrix[0]) % 2 != 0 else 0
92
93         fft_frames = np.pad(fft_frames,
94                             ((0, 0),
95                              ((n_fft - len(windowed_matrix[0])) // 2,
96                               (n_fft - len(windowed_matrix[0])) // 2 + odd)),
97                             mode='constant', constant_values=0)
98     fft_frames = np.fft.fft(fft_frames)
99     # get power spectrum
100    power_spectrum = np.abs(fft_frames) ** 2
101    # filter points (start and stop) and corresponding frequencies
102    filter_points, frequency = get_filter_points(min_freq, max_freq, mel_filter_num, n_fft,
       sample_rate)
103    filters = get_filters(filter_points, n_fft)
104    # take right side of spectrum
105    right = power_spectrum[:, :len(power_spectrum[0]) // 2 + 1]
106    # convolution with filters
107    filtered_frames = np.dot(filters, right.T)
108    # log
109    filtered_frames = 10.0 * np.log10(filtered_frames)
110
```

```
    filtered_frames = filtered_frames.T
    filtered_frames_copy = np.flip(filtered_frames, 1)

    prepared_filtered_frames = np.append(filtered_frames, filtered_frames_copy, 1)

    second_fft = np.fft.fft(prepared_filtered_frames).real
    return second_fft[:, :int(len(second_fft[0]) / 2)].T
```

Listing B.3: Quellcode für die geschriebene Library zur Berechnung der MFCCs.

comparison.py

```
# Author: David Schulz
# Comparisson between the librosa implementation of mfcc and the custom one

import numpy as np
from librosa import load, ex, feature
from mfcc import mfcc
from matplotlib import pyplot as plt



y, sample_rate = load(ex('libri1'), sr=48000, duration=2)

lib_mfcc = feature.mfcc(
    y=y,
    sr=sample_rate,
    n_mfcc=16,
    dct_type=2,
    win_length=960,
    hop_length=480,
    n_fft=1024,
    window='hamm'
)

lib_spec = feature.melspectrogram(
    y=y,
    sr=sample_rate,
    win_length=960,
    hop_length=480,
    n_fft=1024,
    window='hamm',
    norm=None,
    center=True
)
lib_spec = 10.0 * np.log10(lib_spec)
lib_mfcc = feature.mfcc(S=lib_spec, n_mfcc=16)


mfccs = mfcc(y, sample_rate=sample_rate)



plt.figure(1)
plt.subplot(211)
plt.imshow(lib_mfcc, aspect='auto', origin='lower')
plt.title('Librosa MFCC')
plt.xlabel('Time (s)')
plt.xticks(np.arange(0, 225, 25), np.linspace(0, 2, 9))
plt.ylabel('MFCC Koeffizienten')
plt.tight_layout()
plt.subplot(212)
plt.imshow(mfccs, aspect='auto', origin='lower')
plt.title('Eigene MFCC Implementation')
```

```
53 plt.xlabel('Time (s)')
54 plt.xticks(np.arange(0, 225, 25), np.linspace(0, 2, 9))
55 plt.ylabel('MFCC Koeffizienten')
56 plt.tight_layout()
57 plt.show()
```

Listing B.4: Quellcode für den Vergleich der eigenen Implementation mit der Implementation der Librosa-Bibliothek

B.1.2 Jupyter Notebooks

MFCC_tests.ipynb

MFCC_tests

February 11, 2025

1 Test Implementation von MFCC

Author: David Schulz

[27]:
```
import decimal

import numpy
import math
import logging
from scipy.fftpack import dct
from librosa import load, ex, feature
```

[3]:
```
def calculate_nfft(samplerate, winlen):
    window_length_samples = winlen * samplerate
    nfft = 1
    while nfft < window_length_samples:
        nfft *= 2
    return nfft
```

[5]:
```
def mfcc(signal,samplerate=16000,winlen=0.025,winstep=0.01,numcep=13,
         nfilt=26,nfft=None,lowfreq=0,highfreq=None,preemph=0.
 ↪97,ceplifter=22,appendEnergy=True,
         winfunc=lambda x:numpy.ones((x,))):
    nfft = nfft or calculate_nfft(samplerate, winlen)
    feat,energy =␣
 ↪fbank(signal,samplerate,winlen,winstep,nfilt,nfft,lowfreq,highfreq,preemph,winfunc)
    feat = numpy.log(feat)
    feat = dct(feat, type=2, axis=1, norm='ortho')[:,:numcep]
    feat = lifter(feat,ceplifter)
    if appendEnergy: feat[:,0] = numpy.log(energy) # replace first cepstral␣
 ↪coefficient with log of frame energy
    return feat
```

[30]:
```
def fbank(signal,samplerate=16000,winlen=0.025,winstep=0.01,
          nfilt=26,nfft=512,lowfreq=0,highfreq=None,preemph=0.97,
          winfunc=lambda x:numpy.ones((x,))):
    highfreq= highfreq or samplerate/2
    signal = preemphasis(signal,preemph)
```

1

```
    frames = framesig(signal, winlen*samplerate, winstep*samplerate, winfunc)
    pspec = powspec(frames,nfft)
    energy = numpy.sum(pspec,1) # this stores the total energy in each frame
    energy = numpy.where(energy == 0,numpy.finfo(float).eps,energy) # if energy␣
↪is zero, we get problems with log

    fb = get_filterbanks(nfilt,nfft,samplerate,lowfreq,highfreq)
    feat = numpy.dot(pspec,fb.T) # compute the filterbank energies
    feat = numpy.where(feat == 0,numpy.finfo(float).eps,feat) # if feat is␣
↪zero, we get problems with log

    return feat,energy
```

[7]:
```
def logfbank(signal,samplerate=16000,winlen=0.025,winstep=0.01,
             nfilt=26,nfft=512,lowfreq=0,highfreq=None,preemph=0.97,
             winfunc=lambda x:numpy.ones((x,))):
    feat,energy =␣
↪fbank(signal,samplerate,winlen,winstep,nfilt,nfft,lowfreq,highfreq,preemph,winfunc)
    return numpy.log(feat)
```

[8]:
```
def ssc(signal,samplerate=16000,winlen=0.025,winstep=0.01,
        nfilt=26,nfft=512,lowfreq=0,highfreq=None,preemph=0.97,
        winfunc=lambda x:numpy.ones((x,))):
    highfreq= highfreq or samplerate/2
    signal = sigproc.preemphasis(signal,preemph)
    frames = sigproc.framesig(signal, winlen*samplerate, winstep*samplerate,␣
↪winfunc)
    pspec = sigproc.powspec(frames,nfft)
    pspec = numpy.where(pspec == 0,numpy.finfo(float).eps,pspec) # if things␣
↪are all zeros we get problems

    fb = get_filterbanks(nfilt,nfft,samplerate,lowfreq,highfreq)
    feat = numpy.dot(pspec,fb.T) # compute the filterbank energies
    R = numpy.tile(numpy.linspace(1,samplerate/2,numpy.size(pspec,1)),(numpy.
↪size(pspec,0),1))

    return numpy.dot(pspec*R,fb.T) / feat
```

[11]:
```
def hz2mel(hz):
    return 2595 * numpy.log10(1+hz/700.)
```

[10]:
```
def mel2hz(mel):
    return 700*(10**(mel/2595.0)-1)
```

[12]:
```
def get_filterbanks(nfilt=20,nfft=512,samplerate=16000,lowfreq=0,highfreq=None):
    highfreq= highfreq or samplerate/2
    assert highfreq <= samplerate/2, "highfreq is greater than samplerate/2"
```

```
    # compute points evenly spaced in mels
    lowmel = hz2mel(lowfreq)
    highmel = hz2mel(highfreq)
    melpoints = numpy.linspace(lowmel,highmel,nfilt+2)
    # our points are in Hz, but we use fft bins, so we have to convert
    #  from Hz to fft bin number
    bin = numpy.floor((nfft+1)*mel2hz(melpoints)/samplerate)

    fbank = numpy.zeros([nfilt,nfft//2+1])
    for j in range(0,nfilt):
        for i in range(int(bin[j]), int(bin[j+1])):
            fbank[j,i] = (i - bin[j]) / (bin[j+1]-bin[j])
        for i in range(int(bin[j+1]), int(bin[j+2])):
            fbank[j,i] = (bin[j+2]-i) / (bin[j+2]-bin[j+1])
    return fbank
```

[13]:
```
def lifter(cepstra, L=22):
    if L > 0:
        nframes,ncoeff = numpy.shape(cepstra)
        n = numpy.arange(ncoeff)
        lift = 1 + (L/2.)*numpy.sin(numpy.pi*n/L)
        return lift*cepstra
    else:
        # values of L <= 0, do nothing
        return cepstra
```

[14]:
```
def delta(feat, N):
    if N < 1:
        raise ValueError('N must be an integer >= 1')
    NUMFRAMES = len(feat)
    denominator = 2 * sum([i**2 for i in range(1, N+1)])
    delta_feat = numpy.empty_like(feat)
    padded = numpy.pad(feat, ((N, N), (0, 0)), mode='edge')   # padded version␣
 ↪of feat
    for t in range(NUMFRAMES):
        delta_feat[t] = numpy.dot(numpy.arange(-N, N+1), padded[t : t+2*N+1]) /␣
 ↪denominator   # [t : t+2*N+1] == [(N+t)-N : (N+t)+N+1]
    return delta_feat
```

[16]:
```
def round_half_up(number):
    return int(decimal.Decimal(number).quantize(decimal.Decimal('1'),␣
 ↪rounding=decimal.ROUND_HALF_UP))
```

[17]:
```
def rolling_window(a, window, step=1):
    # http://ellisvalentiner.com/post/2017-03-21-np-strides-trick
    shape = a.shape[:-1] + (a.shape[-1] - window + 1, window)
```

```
    strides = a.strides + (a.strides[-1],)
    return numpy.lib.stride_tricks.as_strided(a, shape=shape, strides=strides)[:
↪:step]
```

[18]:
```
def framesig(sig, frame_len, frame_step, winfunc=lambda x: numpy.ones((x,)),␣
↪stride_trick=True):
    """Frame a signal into overlapping frames.

    :param sig: the audio signal to frame.
    :param frame_len: length of each frame measured in samples.
    :param frame_step: number of samples after the start of the previous frame␣
↪that the next frame should begin.
    :param winfunc: the analysis window to apply to each frame. By default no␣
↪window is applied.
    :param stride_trick: use stride trick to compute the rolling window and␣
↪window multiplication faster
    :returns: an array of frames. Size is NUMFRAMES by frame_len.
    """
    slen = len(sig)
    frame_len = int(round_half_up(frame_len))
    frame_step = int(round_half_up(frame_step))
    if slen <= frame_len:
        numframes = 1
    else:
        numframes = 1 + int(math.ceil((1.0 * slen - frame_len) / frame_step))

    padlen = int((numframes - 1) * frame_step + frame_len)

    zeros = numpy.zeros((padlen - slen,))
    padsignal = numpy.concatenate((sig, zeros))
    if stride_trick:
        win = winfunc(frame_len)
        frames = rolling_window(padsignal, window=frame_len, step=frame_step)
    else:
        indices = numpy.tile(numpy.arange(0, frame_len), (numframes, 1)) +␣
↪numpy.tile(
            numpy.arange(0, numframes * frame_step, frame_step), (frame_len,␣
↪1)).T
        indices = numpy.array(indices, dtype=numpy.int32)
        frames = padsignal[indices]
        win = numpy.tile(winfunc(frame_len), (numframes, 1))

    return frames * win
```

[19]:
```

```

```
def deframesig(frames, siglen, frame_len, frame_step, winfunc=lambda x: numpy.
↪ones((x,))):
    """Does overlap-add procedure to undo the action of framesig.

    :param frames: the array of frames.
    :param siglen: the length of the desired signal, use 0 if unknown. Output␣
↪will be truncated to siglen samples.
    :param frame_len: length of each frame measured in samples.
    :param frame_step: number of samples after the start of the previous frame␣
↪that the next frame should begin.
    :param winfunc: the analysis window to apply to each frame. By default no␣
↪window is applied.
    :returns: a 1-D signal.
    """
    frame_len = round_half_up(frame_len)
    frame_step = round_half_up(frame_step)
    numframes = numpy.shape(frames)[0]
    assert numpy.shape(frames)[1] == frame_len, '"frames" matrix is wrong size,␣
↪2nd dim is not equal to frame_len'

    indices = numpy.tile(numpy.arange(0, frame_len), (numframes, 1)) + numpy.
↪tile(
        numpy.arange(0, numframes * frame_step, frame_step), (frame_len, 1)).T
    indices = numpy.array(indices, dtype=numpy.int32)
    padlen = (numframes - 1) * frame_step + frame_len

    if siglen <= 0: siglen = padlen

    rec_signal = numpy.zeros((padlen,))
    window_correction = numpy.zeros((padlen,))
    win = winfunc(frame_len)

    for i in range(0, numframes):
        window_correction[indices[i, :]] = window_correction[
                                               indices[i, :]] + win + 1e-15  #␣
↪add a little bit so it is never zero
        rec_signal[indices[i, :]] = rec_signal[indices[i, :]] + frames[i, :]

    rec_signal = rec_signal / window_correction
    return rec_signal[0:siglen]
```

[20]:
```
def magspec(frames, NFFT):
    """Compute the magnitude spectrum of each frame in frames. If frames is an␣
↪NxD matrix, output will be Nx(NFFT/2+1).

    :param frames: the array of frames. Each row is a frame.
```

```
    :param NFFT: the FFT length to use. If NFFT > frame_len, the frames are␣
↪zero-padded.
    :returns: If frames is an NxD matrix, output will be Nx(NFFT/2+1). Each row␣
↪will be the magnitude spectrum of the corresponding frame.
    """
    if numpy.shape(frames)[1] > NFFT:
        logging.warn(
            'frame length (%d) is greater than FFT size (%d), frame will be␣
↪truncated. Increase NFFT to avoid.',
            numpy.shape(frames)[1], NFFT)
    complex_spec = numpy.fft.rfft(frames, NFFT)
    return numpy.absolute(complex_spec)
```

[21]:
```
def powspec(frames, NFFT):
    """Compute the power spectrum of each frame in frames. If frames is an NxD␣
↪matrix, output will be Nx(NFFT/2+1).

    :param frames: the array of frames. Each row is a frame.
    :param NFFT: the FFT length to use. If NFFT > frame_len, the frames are␣
↪zero-padded.
    :returns: If frames is an NxD matrix, output will be Nx(NFFT/2+1). Each row␣
↪will be the power spectrum of the corresponding frame.
    """
    return 1.0 / NFFT * numpy.square(magspec(frames, NFFT))
```

[22]:
```
def logpowspec(frames, NFFT, norm=1):
    """Compute the log power spectrum of each frame in frames. If frames is an␣
↪NxD matrix, output will be Nx(NFFT/2+1).

    :param frames: the array of frames. Each row is a frame.
    :param NFFT: the FFT length to use. If NFFT > frame_len, the frames are␣
↪zero-padded.
    :param norm: If norm=1, the log power spectrum is normalised so that the␣
↪max value (across all frames) is 0.
    :returns: If frames is an NxD matrix, output will be Nx(NFFT/2+1). Each row␣
↪will be the log power spectrum of the corresponding frame.
    """
    ps = powspec(frames, NFFT);
    ps[ps <= 1e-30] = 1e-30
    lps = 10 * numpy.log10(ps)
    if norm:
        return lps - numpy.max(lps)
    else:
        return lps
```

[23]:
```
def preemphasis(signal, coeff=0.95):
    """perform preemphasis on the input signal.

    :param signal: The signal to filter.
    :param coeff: The preemphasis coefficient. 0 is no filter, default is 0.95.
    :returns: the filtered signal.
    """
    return numpy.append(signal[0], signal[1:] - coeff * signal[:-1])
```

[25]:
```
y, sr = load(ex('libri1'))
```

[28]:
```
feature.mfcc(y=y, sr=sr, n_mfcc=40)
```

[28]:
```
array([[-565.9195   , -564.28815  , -562.8431   , …, -437.97177  ,
        -426.48358  , -434.66782  ],
       [  10.304618 ,   12.508709 ,   14.130636 , …,   89.98184  ,
          88.43012  ,   90.120255 ],
       [   9.748487 ,   11.672071 ,   12.139029 , …,  -10.730869 ,
         -10.773369 ,   -5.2245703],
       …,
       [  -1.5704863,   -2.0978153,   -3.0435863, …,   -2.585959 ,
         -4.4007945,   -4.668362 ],
       [  -1.9920207,   -2.682422 ,   -3.1783807, …,   -4.045431 ,
          1.487895 ,    3.6362567],
       [  -2.1463804,   -2.8828554,   -2.8759928, …,    1.6753986,
          3.1740243,    2.8180344]], dtype=float32)
```

[34]:
```
mfcc(y, samplerate=sr, nfilt=40)
```

[34]:
```
array([[-13.75522287, -16.69152564,  -2.98704684, …,  28.00897888,
          8.3748943 ,  -0.12729957],
       [-13.85139797, -16.48606701,  -4.88373073, …,  22.32051789,
          9.41018658,   6.71527343],
       [-13.9578787 , -16.69240289,  -7.96790991, …,  21.63484217,
          2.43201493,  12.01686139],
       …,
       [ -8.23216429,  -1.17245832, -35.19614476, …,  13.39814182,
         -0.18192372,  -3.80421863],
       [ -8.15667423,   1.13421399, -33.88410326, …,  21.16195423,
          2.57219832,  -4.70251236],
       [ -8.54108162,  -0.15602263, -28.69141386, …,   7.52136996,
         -1.00294898,  -8.72056597]])
```

[ ]:
```

```

MFCC_Librosa_tests.ipynb

MFCC_Librosa_tests

February 11, 2025

1 Testing the Library Librosa

Author: David Schulz

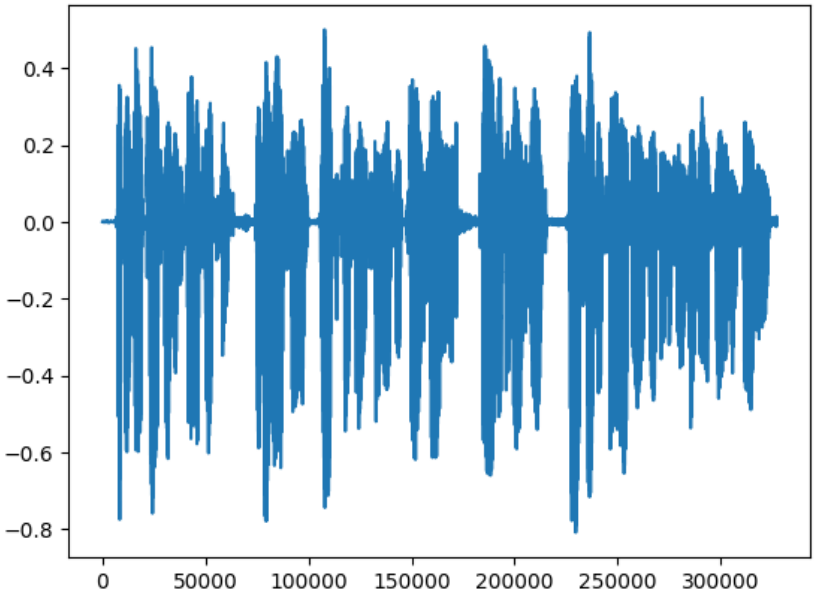
[14]:
```
from librosa import load, feature, ex, display, power_to_db
from matplotlib import pyplot as plt
import numpy as np
```

[26]:
```
y, sr = load(ex('libri1'))
```

[26]:
```
array([0.00037424, 0.00036333, 0.00024905, …, 0.00101743, 0.0023917 ,
       0.00200722], dtype=float32)
```

[27]:
```
plt.plot(y)
```

[27]:
```
[<matplotlib.lines.Line2D at 0x115771160>]
```



[22]:
```
S = feature.melspectrogram(y=y, sr=sr, n_mels=128,fmax=8000)
feature.mfcc(S=power_to_db(S))
```

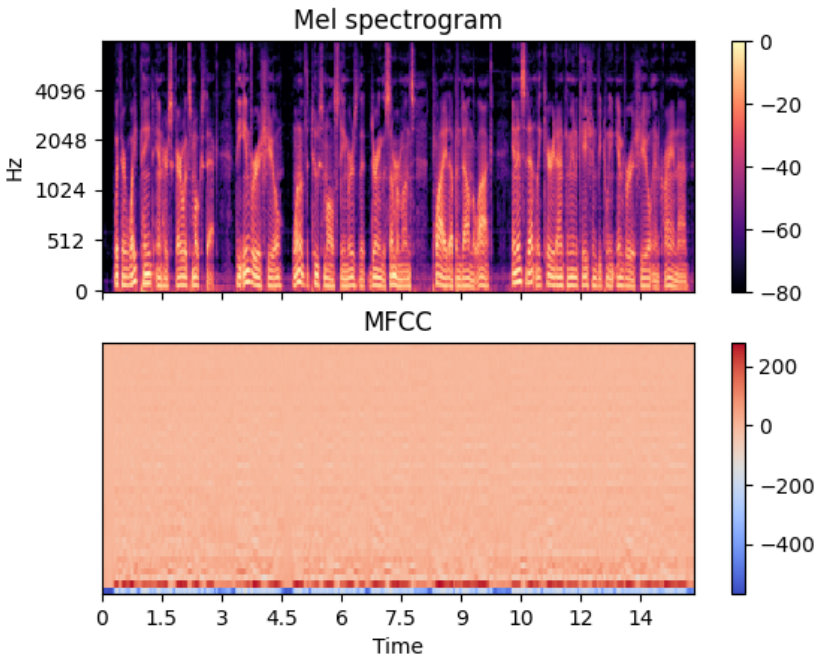
[22]:
```
array([[-5.5997437e+02, -5.5844916e+02, -5.5695001e+02, …,
        -4.2392285e+02, -4.1196042e+02, -4.2045792e+02],
       [ 1.1018242e+01,  1.3046116e+01,  1.4626852e+01, …,
         8.1172806e+01,  7.6971542e+01,  8.0888062e+01],
       [ 1.0409790e+01,  1.2093786e+01,  1.2224331e+01, …,
        -1.3710316e+01, -8.7659798e+00, -5.3241410e+00],
       …,
       [ 2.6907277e+00,  2.0947857e+00, -2.1460218e+00, …,
        -8.4138060e+00, -8.0626354e+00, -6.1864362e+00],
       [ 2.7126925e+00,  2.3788295e+00, -1.3990482e+00, …,
        -3.2831826e+00,  1.4638859e+00, -2.8352113e+00],
       [ 2.7118654e+00,  2.6189370e+00, -3.9603093e-01, …,
         5.6303434e+00,  2.2087626e+00,  6.4752209e-01]], dtype=float32)
```

[23]:
```
mfccs = feature.mfcc(y=y, sr=sr, n_mfcc=40)
```

[25]:
```
fig, ax = plt.subplots(nrows=2, sharex=True)
img = display.specshow(power_to_db(S, ref=np.max),x_axis='time', y_axis='mel',␣
↪fmax=8000,ax=ax[0])
```

```
fig.colorbar(img, ax=[ax[0]])
ax[0].set(title='Mel spectrogram')
ax[0].label_outer()
img = display.specshow(mfccs, x_axis='time', ax=ax[1])
fig.colorbar(img, ax=[ax[1]])
ax[1].set(title='MFCC')
```

[25]: [Text(0.5, 1.0, 'MFCC')]



[ ]:

MFCC_librosa

February 11, 2025

1 Extracted the Code from the Librosa Library and simplified the code

Simplifications by David Schulz

[430]:
```
from librosa import load, ex, display
import numpy as np
import scipy
from numpy.lib.stride_tricks import as_strided
```

[431]:
```
# Copyright (c) 2013--2023, librosa development team.

MAX_MEM_BLOCK = 2**8 * 2**10
__FFTLIB = None
```

[432]:
```
class Deprecated(object):
    """A placeholder class to catch usage of deprecated variable names"""

    def __repr__(self) -> str:
        """Pretty-print display for deprecated objects"""
        return "<DEPRECATED parameter>"
```

[679]:
```
def mfcc(
    *,
    y = None,
    sr = 22050,
    n_mfcc = 20,
    dct_type = 2,
    norm = "ortho",
    lifter: float = 0,
    mel_norm = "slaney"
):

    S = power_to_db(melspectrogram(y=y, sr=sr))

    return scipy.fftpack.dct(S, axis=-2, type=dct_type, norm=norm)[..., :
 ↪n_mfcc, :]
```

[717]:
```python
def melspectrogram(
    *,
    y = None,
    sr = 22050,
    n_fft = 2048,
    hop_length = 512,
    win_length = None,
    window = "hann",
    center= True,
    pad_mode = "constant",
    power = 2.0
):
    S, n_fft = _spectrogram(
        y=y,
        n_fft=n_fft,
        hop_length=hop_length,
        power=power,
        win_length=win_length,
        window=window,
        center=center,
        pad_mode=pad_mode,
    )

    # Build a Mel filter
    mel_basis = mel(sr=sr, n_fft=n_fft)

    return np.einsum("...ft,mf->...mt", S, mel_basis, optimize=True)
```

[719]:
```python
def power_to_db(S):
    ref = 1.0,
    amin = 1e-10,
    top_db = 80.0

    magnitude = S

    ref_value = np.abs(ref)

    log_spec = 10.0 * np.log10(np.maximum(amin, magnitude))
    log_spec -= 10.0 * np.log10(np.maximum(amin, ref_value))

    return np.maximum(log_spec, log_spec.max() - top_db)
```

[721]:
```python
def _spectrogram(
    *,
    y = None,
    S = None,
    n_fft = 2048,
```

```
    hop_length = 512,
    power = 1,
    win_length = None,
    window = "hann",
    center = True,
    pad_mode = "constant",
):
    S = (
        np.abs(
            stft(
                y,
                n_fft=n_fft,
                hop_length=hop_length,
                win_length=win_length,
                center=center,
                window=window,
                pad_mode=pad_mode,
            )
        )
        ** power
    )

    return S, n_fft
```

[787]:
```
def stft(
    y,
    *,
    n_fft = 2048,
    hop_length = None,
    win_length = None,
    window = "hann",
    center = True,
    dtype = None,
    pad_mode = "constant",
    out = None,
):
    # By default, use the entire frame
    if win_length is None:
        win_length = n_fft

    # Set the default hop, if it's not already specified
    if hop_length is None:
        hop_length = int(win_length // 4)
    elif not is_positive_int(hop_length):
        raise ParameterError(f"hop_length={hop_length} must be a positive␣
 ↪integer")
```

```
    # Check audio is valid
    valid_audio(y, mono=False)

    fft_window = get_window(window, win_length, fftbins=True)

    # Pad the window out to n_fft size
    fft_window = pad_center(fft_window, size=n_fft)

    # Reshape so that the window can be broadcast
    fft_window = expand_to(fft_window, ndim=1 + y.ndim, axes=-2)

    # Pad the time series so that frames are centered
    if center:
        if pad_mode in ("wrap", "maximum", "mean", "median", "minimum"):
            # Note: padding with a user-provided function "works", but
            # use at your own risk.
            # Since we don't pass-through kwargs here, any arguments
            # to a user-provided pad function should be encapsulated
            # by using functools.partial:
            #
            # >>> my_pad_func = functools.partial(pad_func, foo=x, bar=y)
            # >>> librosa.stft(..., pad_mode=my_pad_func)

            raise ParameterError(
                f"pad_mode='{pad_mode}' is not supported by librosa.stft"
            )

        if n_fft > y.shape[-1]:
            warnings.warn(
                f"n_fft={n_fft} is too large for input signal of length={y.
↪shape[-1]}"
            )

        # Set up the padding array to be empty, and we'll fix the target␣
↪dimension later
        padding = [(0, 0) for _ in range(y.ndim)]

        # How many frames depend on left padding?
        start_k = int(np.ceil(n_fft // 2 / hop_length))

        # What's the first frame that depends on extra right-padding?
        tail_k = (y.shape[-1] + n_fft // 2 - n_fft) // hop_length + 1

        if tail_k <= start_k:
            # If tail and head overlap, then just copy-pad the signal and carry␣
↪on
            start = 0
```

```
            extra = 0
            padding[-1] = (n_fft // 2, n_fft // 2)
            y = np.pad(y, padding, mode=pad_mode)
        else:
            # If tail and head do not overlap, then we can implement padding on␣
↪each part separately
            # and avoid a full copy-pad

            # "Middle" of the signal starts here, and does not depend on head␣
↪padding
            start = start_k * hop_length - n_fft // 2
            padding[-1] = (n_fft // 2, 0)

            # +1 here is to ensure enough samples to fill the window
            # fixes bug #1567
            y_pre = np.pad(
                y[..., : (start_k - 1) * hop_length - n_fft // 2 + n_fft + 1],
                padding,
                mode=pad_mode,
            )
            y_frames_pre = frame(y_pre, frame_length=n_fft,␣
↪hop_length=hop_length)
            # Trim this down to the exact number of frames we should have
            y_frames_pre = y_frames_pre[..., :start_k]

            # How many extra frames do we have from the head?
            extra = y_frames_pre.shape[-1]

            # Determine if we have any frames that will fit inside the tail pad
            if tail_k * hop_length - n_fft // 2 + n_fft <= y.shape[-1] + n_fft /
↪/ 2:
                padding[-1] = (0, n_fft // 2)
                y_post = np.pad(
                    y[..., (tail_k) * hop_length - n_fft // 2 :], padding,␣
↪mode=pad_mode
                )
                y_frames_post = frame(
                    y_post, frame_length=n_fft, hop_length=hop_length
                )
                # How many extra frames do we have from the tail?
                extra += y_frames_post.shape[-1]
            else:
                # In this event, the first frame that touches tail padding␣
↪would run off
                # the end of the padded array
```

```
                # We'll circumvent this by allocating an empty frame buffer for␣
↪the tail
                # this keeps the subsequent logic simple
                post_shape = list(y_frames_pre.shape)
                post_shape[-1] = 0
                y_frames_post = np.empty_like(y_frames_pre, shape=post_shape)
    else:
        if n_fft > y.shape[-1]:
            raise ParameterError(
                f"n_fft={n_fft} is too large for uncentered analysis of input␣
↪signal of length={y.shape[-1]}"
            )

        # "Middle" of the signal starts at sample 0
        start = 0
        # We have no extra frames
        extra = 0

    fft = get_fftlib()

    if dtype is None:
        dtype = dtype_r2c(y.dtype)

    # Window the time series.
    y_frames = frame(y[..., start:], frame_length=n_fft, hop_length=hop_length)

    # Pre-allocate the STFT matrix
    shape = list(y_frames.shape)

    # This is our frequency dimension
    shape[-2] = 1 + n_fft // 2

    # If there's padding, there will be extra head and tail frames
    shape[-1] += extra

    if out is None:
        stft_matrix = np.zeros(shape, dtype=dtype, order="F")
    elif not (np.allclose(out.shape[:-1], shape[:-1]) and out.shape[-1] >=␣
↪shape[-1]):
        raise ParameterError(
            f"Shape mismatch for provided output array out.shape={out.shape}␣
↪and target shape={shape}"
        )
    elif not np.iscomplexobj(out):
        raise ParameterError(f"output with dtype={out.dtype} is not of complex␣
↪type")
    else:
```

```
        if np.allclose(shape, out.shape):
            stft_matrix = out
        else:
            stft_matrix = out[..., : shape[-1]]

    # Fill in the warm-up
    if center and extra > 0:
        off_start = y_frames_pre.shape[-1]
        stft_matrix[..., :off_start] = fft.rfft(fft_window * y_frames_pre,␣
↪axis=-2)

        off_end = y_frames_post.shape[-1]
        if off_end > 0:
            stft_matrix[..., -off_end:] = fft.rfft(fft_window * y_frames_post,␣
↪axis=-2)
    else:
        off_start = 0

    n_columns = int(
        MAX_MEM_BLOCK // (np.prod(y_frames.shape[:-1]) * y_frames.itemsize)
    )
    n_columns = max(n_columns, 1)

    for bl_s in range(0, y_frames.shape[-1], n_columns):
        bl_t = min(bl_s + n_columns, y_frames.shape[-1])

        stft_matrix[..., bl_s + off_start : bl_t + off_start] = fft.rfft(
            fft_window * y_frames[..., bl_s:bl_t], axis=-2
        )
    return stft_matrix
```

[789]:
```
def is_positive_int(x):
    # Check type first to catch None values.
    return isinstance(x, (int, np.integer)) and (x > 0)
```

[791]:
```
def valid_audio(y, *, mono = Deprecated()):
    if not isinstance(y, np.ndarray):
        raise ParameterError("Audio data must be of type numpy.ndarray")

    if not np.issubdtype(y.dtype, np.floating):
        raise ParameterError("Audio data must be floating-point")

    if y.ndim == 0:
        raise ParameterError(
            f"Audio data must be at least one-dimensional, given y.shape={y.
↪shape}"
        )
```

```
    if isinstance(mono, Deprecated):
        mono = False

    if mono and y.ndim != 1:
        raise ParameterError(
            f"Invalid shape for monophonic audio: ndim={y.ndim:d}, shape={y.
↪shape}"
        )

    if not np.isfinite(y).all():
        raise ParameterError("Audio buffer is not finite everywhere")

    return True
```

[793]:
```
def get_window(
    window,
    Nx,
    *,
    fftbins = True,
):
    if callable(window):
        return window(Nx)

    elif isinstance(window, (str, tuple)) or np.isscalar(window):
        # TODO: if we add custom window functions in librosa, call them here

        win: np.ndarray = scipy.signal.get_window(window, Nx, fftbins=fftbins)
        return win

    elif isinstance(window, (np.ndarray, list)):
        if len(window) == Nx:
            return np.asarray(window)

        raise ParameterError(f"Window size mismatch: {len(window):d} != {Nx:d}")
    else:
        raise ParameterError(f"Invalid window specification: {window!r}")
```

[795]:
```
def pad_center(
    data, *, size, axis = -1, **kwargs
):
    kwargs.setdefault("mode", "constant")

    n = data.shape[axis]

    lpad = int((size - n) // 2)
```

```
    lengths = [(0, 0)] * data.ndim
    lengths[axis] = (lpad, int(size - n - lpad))

    if lpad < 0:
        raise ParameterError(
            f"Target size ({size:d}) must be at least input size ({n:d})"
        )

    return np.pad(data, lengths, **kwargs)
```

[797]:
```
def expand_to(
    x, *, ndim, axes
):
    # Force axes into a tuple
    axes_tup: Tuple[int]
    try:
        axes_tup = tuple(axes)  # type: ignore
    except TypeError:
        axes_tup = tuple([axes])  # type: ignore

    if len(axes_tup) != x.ndim:
        raise ParameterError(
            f"Shape mismatch between axes={axes_tup} and input x.shape={x.
 ↪shape}"
        )

    if ndim < x.ndim:
        raise ParameterError(
            f"Cannot expand x.shape={x.shape} to fewer dimensions ndim={ndim}"
        )

    shape: List[int] = [1] * ndim
    for i, axi in enumerate(axes_tup):
        shape[axi] = x.shape[i]

    return x.reshape(shape)
```

[799]:
```
def frame(
    x,
    *,
    frame_length,
    hop_length,
    axis = -1,
    writeable = False,
    subok = False,
):
```

```
    x = np.array(x, copy=False, subok=subok)

    if x.shape[axis] < frame_length:
        raise ParameterError(
            f"Input is too short (n={x.shape[axis]:d}) for␣
↪frame_length={frame_length:d}"
        )

    if hop_length < 1:
        raise ParameterError(f"Invalid hop_length: {hop_length:d}")

    # put our new within-frame axis at the end for now
    out_strides = x.strides + tuple([x.strides[axis]])

    # Reduce the shape on the framing axis
    x_shape_trimmed = list(x.shape)
    x_shape_trimmed[axis] -= frame_length - 1

    out_shape = tuple(x_shape_trimmed) + tuple([frame_length])
    xw = as_strided(
        x, strides=out_strides, shape=out_shape, subok=subok,␣
↪writeable=writeable
    )

    if axis < 0:
        target_axis = axis - 1
    else:
        target_axis = axis + 1

    xw = np.moveaxis(xw, -1, target_axis)

    # Downsample along the target axis
    slices = [slice(None)] * xw.ndim
    slices[axis] = slice(0, None, hop_length)
    return xw[tuple(slices)]
```

[801]:
```
def dtype_r2c(d, *, default = np.complex64):
    mapping: Dict[DTypeLike, type] = {
        np.dtype(np.float32): np.complex64,
        np.dtype(np.float64): np.complex128,
        np.dtype(float): np.dtype(complex).type,
    }

    # If we're given a complex type already, return it
    dt = np.dtype(d)
    if dt.kind == "c":
        return dt
```

```
    # Otherwise, try to map the dtype.
    # If no match is found, return the default.
    return np.dtype(mapping.get(dt, default))
```

[803]:
```
def get_fftlib():
    """Get the FFT library currently used by librosa

    Returns
    -------
    fft : module
        The FFT library currently used by librosa.
        Must API-compatible with `numpy.fft`.
    """
    if __FFTLIB is None:
        # This path should never occur because importing
        # this module will call set_fftlib
        assert False  # pragma: no cover

    return __FFTLIB


def set_fftlib(lib = None):

    global __FFTLIB
    if lib is None:
        from numpy import fft

        lib = fft

    __FFTLIB = lib

set_fftlib(None)
```

[805]:
```
def mel(
    *,
    sr,
    n_fft,
    n_mels = 128,
    fmin = 0.0,
    fmax = None,
    htk = False,
    norm = "slaney",
    dtype = np.float32,
):
    if fmax is None:
        fmax = float(sr) / 2
```

```
    # Initialize the weights
    n_mels = int(n_mels)
    weights = np.zeros((n_mels, int(1 + n_fft // 2)), dtype=dtype)

    # Center freqs of each FFT bin
    fftfreqs = fft_frequencies(sr=sr, n_fft=n_fft)

    # 'Center freqs' of mel bands - uniformly spaced between limits
    mel_f = mel_frequencies(n_mels + 2, fmin=fmin, fmax=fmax, htk=htk)

    fdiff = np.diff(mel_f)
    ramps = np.subtract.outer(mel_f, fftfreqs)

    for i in range(n_mels):
        # lower and upper slopes for all bins
        lower = -ramps[i] / fdiff[i]
        upper = ramps[i + 2] / fdiff[i + 1]

        # .. then intersect them with each other and zero
        weights[i] = np.maximum(0, np.minimum(lower, upper))

    if isinstance(norm, str):
        if norm == "slaney":
            # Slaney-style mel is scaled to be approx constant energy per␣
↪channel
            enorm = 2.0 / (mel_f[2 : n_mels + 2] - mel_f[:n_mels])
            weights *= enorm[:, np.newaxis]
        else:
            raise ParameterError(f"Unsupported norm={norm}")
    else:
        weights = util.normalize(weights, norm=norm, axis=-1)

    # Only check weights if f_mel[0] is positive
    if not np.all((mel_f[:-2] == 0) | (weights.max(axis=1) > 0)):
        # This means we have an empty channel somewhere
        warnings.warn(
            "Empty filters detected in mel frequency basis. "
            "Some channels will produce empty responses. "
            "Try increasing your sampling rate (and fmax) or "
            "reducing n_mels.",
            stacklevel=2,
        )

    return weights
```

[807]:
```
def fft_frequencies(*, sr = 22050, n_fft = 2048):
    return np.fft.rfftfreq(n=n_fft, d=1.0 / sr)
```

[809]:
```
def mel_frequencies(
    n_mels = 128, *, fmin = 0.0, fmax = 11025.0, htk = False
):
    # 'Center freqs' of mel bands - uniformly spaced between limits
    min_mel = hz_to_mel(fmin, htk=htk)
    max_mel = hz_to_mel(fmax, htk=htk)

    mels = np.linspace(min_mel, max_mel, n_mels)

    hz: np.ndarray = mel_to_hz(mels, htk=htk)
    return hz
```

[811]:
```
def hz_to_mel(
    frequencies, *, htk = False
):
    frequencies = np.asanyarray(frequencies)

    if htk:
        mels: np.ndarray = 2595.0 * np.log10(1.0 + frequencies / 700.0)
        return mels

    # Fill in the linear part
    f_min = 0.0
    f_sp = 200.0 / 3

    mels = (frequencies - f_min) / f_sp

    # Fill in the log-scale part

    min_log_hz = 1000.0  # beginning of log region (Hz)
    min_log_mel = (min_log_hz - f_min) / f_sp  # same (Mels)
    logstep = np.log(6.4) / 27.0  # step size for log region

    if frequencies.ndim:
        # If we have array data, vectorize
        log_t = frequencies >= min_log_hz
        mels[log_t] = min_log_mel + np.log(frequencies[log_t] / min_log_hz) /␣
↪logstep
    elif frequencies >= min_log_hz:
        # If we have scalar data, heck directly
        mels = min_log_mel + np.log(frequencies / min_log_hz) / logstep

    return mels
```

[813]:
```
def mel_to_hz(
    mels, *, htk = False
):
    mels = np.asanyarray(mels)

    if htk:
        return 700.0 * (10.0 ** (mels / 2595.0) - 1.0)

    # Fill in the linear scale
    f_min = 0.0
    f_sp = 200.0 / 3
    freqs = f_min + f_sp * mels

    # And now the nonlinear scale
    min_log_hz = 1000.0  # beginning of log region (Hz)
    min_log_mel = (min_log_hz - f_min) / f_sp  # same (Mels)
    logstep = np.log(6.4) / 27.0  # step size for log region

    if mels.ndim:
        # If we have vector data, vectorize
        log_t = mels >= min_log_mel
        freqs[log_t] = min_log_hz * np.exp(logstep * (mels[log_t] -␣
 ↪min_log_mel))
    elif mels >= min_log_mel:
        # If we have scalar data, check directly
        freqs = min_log_hz * np.exp(logstep * (mels - min_log_mel))

    return freqs
```

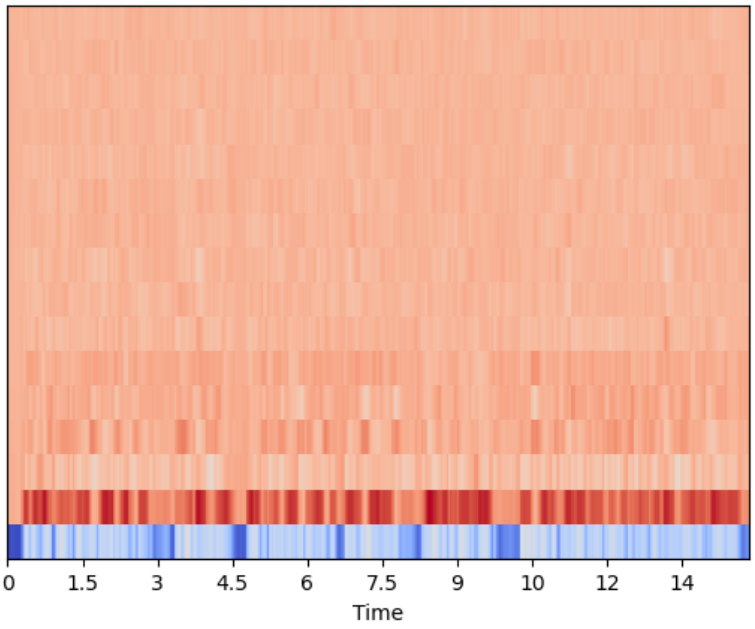
[815]:
```
y, sr = load(ex('libri1'))
```

[817]:
```
mfccs = mfcc(y=y, sr=sr, n_mfcc=16)
mfccs
```

[817]:
```
array([[-565.9194613 , -564.28805992, -562.84308008, …, -437.97176722,
        -426.48356456, -434.66781209],
       [  10.30461676,   12.50870474,   14.13063173, …,   89.98183514,
          88.43012141,   90.12025077],
       [   9.74848749,   11.67206972,   12.1390225 , …,  -10.73086516,
         -10.77335902,   -5.22456194],
       …,
       [   6.43103586,    7.01146229,    6.77968061, …,   -6.22595059,
         -7.29734196,   -2.21974808],
       [   5.43160641,    5.56400194,    4.63668666, …,    2.15356643,
          4.84169508,    5.67298347],
       [   4.42381533,    4.12633344,    2.23221949, …,   10.92779237,
         12.51458006,   10.92853341]])
```

[640]:
```
display.specshow(mfccs, x_axis='time')
```

[640]:
```
<matplotlib.collections.QuadMesh at 0x13b3c0290>
```



[ ]:

[ ]:

MFCC_eigen.ipynb

MFCC_eigen

February 13, 2025

Author: David Schulz

[93]:
```
from librosa import load, ex, feature, display
from sklearn.decomposition import PCA
import numpy as np
from matplotlib import pyplot as plt
```

[30]:
```
def segment_signal(signal, winDur = 20e-3, hopDur=10e-3, sr=16e3):
    # hop_size in ms
    winLen = int(winDur*sr)
    hopLen = int(hopDur*sr)
    signal = np.pad(signal, winLen//2, mode='reflect')

    nframes = int((len(signal) - winLen) / hopLen) + 1
    frames = np.zeros((nframes,winLen))

    window = np.hamming(winLen)
    window = window-np.min(window)
    window = window/np.max(window)

    for n in range(nframes):
        frames[n] = window*signal[n*hopLen:n*hopLen+winLen]
    return frames
```

[31]:
```
def nearestpow2(n):
    k=1
    while n>2**k:
        k = k+1
    return 2**k
```

[32]:
```
def freq_to_mel(freq):
    # converting linear scale frequency to mel-scale
    return 2595.0 * np.log10(1.0 + freq / 700.0)

def mel_to_freq(mels):
    # converting mel-scale frequency to linear scale
    return 700.0 * (10.0**(mels / 2595.0) - 1.0)
```

[33]:
```
def get_filter_points(fmin, fmax, mel_filter_num, nfft, sample_rate=16000):
    fmin_mel = freq_to_mel(fmin)
    fmax_mel = freq_to_mel(fmax)

    mels = np.linspace(fmin_mel, fmax_mel, num=mel_filter_num+2)
    freqs = mel_to_freq(mels)

    return np.floor((nfft) / sample_rate * freqs).astype(int), freqs

def get_filters(filter_points, nfft):
    filters = np.zeros((len(filter_points)-2,int(nfft/2+1)))

    for n in range(len(filter_points)-2):
        filters[n, filter_points[n] : filter_points[n + 1]] = np.linspace(0, 1,␣
↪filter_points[n + 1] - filter_points[n])
        filters[n, filter_points[n + 1] : filter_points[n + 2]] = np.
↪linspace(1, 0, filter_points[n + 2] - filter_points[n + 1])

    return filters
```

[149]:
```
def dct(dct_filter_num, filter_len):
    basis = np.empty((dct_filter_num,filter_len))
    basis[0, :] = 1.0 / np.sqrt(filter_len)

    samples = np.arange(1, 2 * filter_len, 2) * np.pi / (2.0 * filter_len)

    for i in range(1, dct_filter_num):
        basis[i, :] = np.cos(i * samples) * np.sqrt(2.0 / filter_len)

    return basis

dct_basis = dct(40,40)
```

[45]:
```
y, sr = load(ex('libri1'))
hopDur = 10e-3 #ms
winDur = 20e-3
```

[46]:
```
x_segs_hamming = segment_signal(y, winDur=winDur, hopDur= hopDur, sr=sr)
```

[133]:
```
#collapse
x_segs = x_segs_hamming.copy()
nfft = nearestpow2(x_segs_hamming.shape[1])
X = np.zeros((x_segs.shape[0],int(nfft/2)+1))
X_fft = np.fft.rfft(x_segs,nfft,axis=1) # fft block verwenden
X = np.abs(X_fft)**2
```

[142]:
```
freq_min = 0
freq_high = sr / 2
mel_filter_num = 40

filter_points, freqs = get_filter_points(freq_min, freq_high, mel_filter_num,␣
 ↪nfft=nfft, sample_rate=sr)
filters = get_filters(filter_points, nfft=nfft)
```

[143]:
```
X_filtered = np.dot(filters, X.T)
X_filtered_log = 10.0 * np.log10(X_filtered)

X = np.where(X == 0,np.finfo(float).eps, X)
```

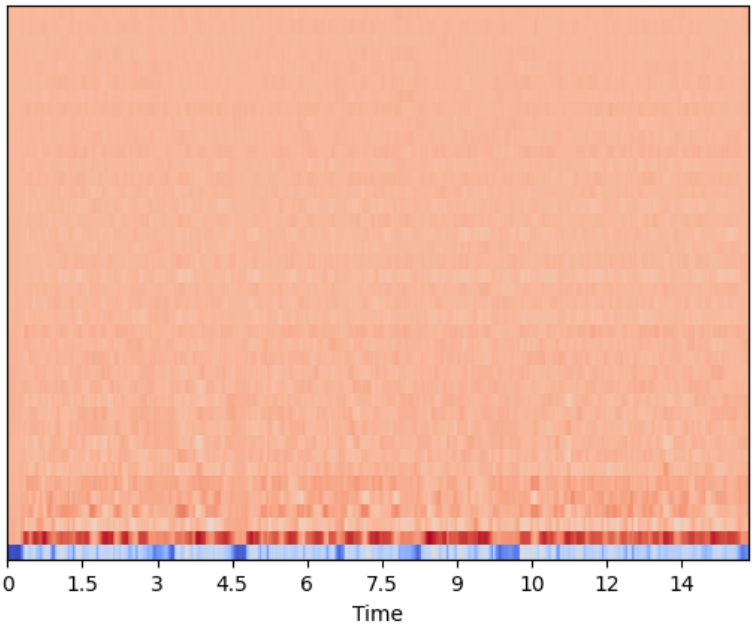
[150]:
```
dct_filter_num = X_scaled.shape[0]
dct_filters = dct(dct_filter_num, X_filtered_log.shape[0])
cepstral_coefficents = np.dot(dct_filters, X_filtered_log)
```

[145]:
```
mfccs = feature.mfcc(y=y, sr=sr, n_mfcc=40)
mfccs
```

[145]:
```
array([[-565.9195   , -564.28815  , -562.8431   , …, -437.97177  ,
        -426.48358  , -434.66782  ],
       [  10.304618 ,   12.508709 ,   14.130636 , …,   89.98184  ,
          88.43012  ,   90.120255 ],
       [   9.748487 ,   11.672071 ,   12.139029 , …,  -10.730869 ,
         -10.773369 ,   -5.2245703],
       …,
       [  -1.5704863,   -2.0978153,   -3.0435863, …,   -2.585959 ,
         -4.4007945,   -4.668362 ],
       [  -1.9920207,   -2.682422 ,   -3.1783807, …,   -4.045431 ,
          1.487895 ,    3.6362567],
       [  -2.1463804,   -2.8828554,   -2.8759928, …,    1.6753986,
          3.1740243,    2.8180344]], dtype=float32)
```

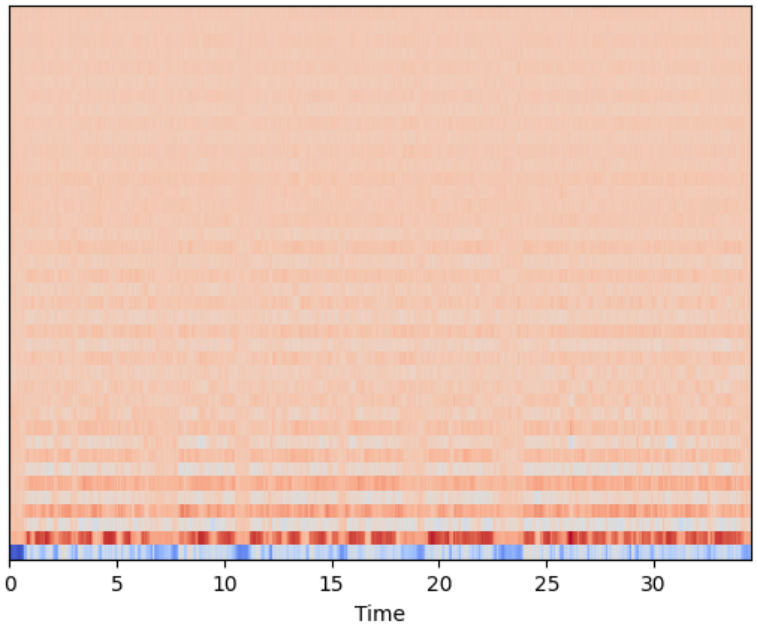
[146]:
```
display.specshow(mfccs, x_axis='time')
```

[146]:
```
<matplotlib.collections.QuadMesh at 0x10f05b200>
```



[147]:
```
display.specshow(cepstral_coefficents, x_axis='time')
```

[147]:
```
<matplotlib.collections.QuadMesh at 0x1102be4e0>
```


0
5
10
15
20
25
30
Time

[ ]:

MFCC_Custom_Implementation

February 11, 2025

1 Custom MFCC Implementation:

Author: David Schulz

[ ]:
```
import numpy as np
from librosa import load, feature, ex, display
```

[16]:
```
def extract_mfcc_feature(
    y: np.array,
    fs: float,
    n_fft: int = 512,
    frame_size: float = 0.025,
    frame_step: float = 0.01,
    n_mels: int = 40,
    n_mfcc: int = 13
):

    mag_spec_frames = np.abs(stft(y, fs, n_fft, frame_size, frame_step))

    pow_spec_frames = (mag_spec_frames**2) / mag_spec_frames.shape[1]

    mel_power_spec_frames, hz_freq = mel_filter(pow_spec_frames, 0, fs/2,␣
 ↪n_mels, fs)

    log_spec_frames = signal_power_to_db(mel_power_spec_frames)

    mfcc = discrete_cos_transformation(log_spec_frames)

    mfcc = sin_liftering(mfcc)

    mfcc = mfcc[:, 1:n_mfcc]

    return mfcc
```

[17]:
```
def framing(signal: np.array, fs: float, frame_size: float, frame_step: float):
    frame_length = np.round(frame_size * fs).astype(int)
    frame_step = np.round(frame_step * fs).astype(int)
    signal_length = signal.shape[0]
```

```
    n_frames = np.ceil(abs(signal_length - frame_length) / frame_step).
 ↪astype(int)

    pad_signal_length = int(n_frames * frame_step + frame_length)
    zeros_pad = np.zeros((1, pad_signal_length - signal_length))
    pad_signal = np.concatenate((signal.reshape((1, -1)), zeros_pad), axis=1).
 ↪reshape(-1)

    frames = np.zeros((n_frames, frame_length))
    indices = np.arange(0, frame_length)

    for i in np.arange(0, n_frames):
        offset = i * frame_step
        frames[i] = pad_signal[(indices + offset)]

    return frames


def hamming(frames: np.array):
    window_length = frames.shape[1]
    n = np.arange(0, window_length)

    h = 0.54 - 0.46 * np.cos(2 * np.pi * n / (window_length - 1))

    frames *= h

    return frames


def stft(y: np.array, fs: float, n_fft: int, frame_size: float, frame_step:␣
 ↪float):
    frames = framing(y, fs, frame_size, frame_step)

    frames = hamming(frames)

    spec_frames = np.fft.fft(frames, n=n_fft, axis=1)

    spec_frames = spec_frames[:, 0: int(n_fft / 2 + 1)]

    return spec_frames


def mel_filter(frames, f_min, f_max, n_mels, fs):

    n_fft = frames.shape[1] - 1
```

2

```
    mel_lf = 2595 * np.log10(1 + f_min / 700)
    mel_hf = 2595 * np.log10(1 + f_max / 700)

    mel_points = np.linspace(mel_lf, mel_hf, n_mels + 2)

    hz_points = 700 * (np.power(10, mel_points / 2595) - 1)

    fft_bank_bin = np.floor((n_fft + 1) * hz_points / (fs / 2))
    fft_bank_bin[-1] = n_fft

    f_bank = np.zeros((n_mels, n_fft + 1))
    for i in np.arange(1, n_mels + 1):
        left_f = int(fft_bank_bin[i - 1])
        center_f = int(fft_bank_bin[i])
        right_f = int(fft_bank_bin[i + 1])

        for k in np.arange(left_f, center_f + 1):
            f_bank[i - 1, k] = (k - left_f) / (center_f - left_f)

        for k in np.arange(center_f, right_f + 1):
            f_bank[i - 1, k] = (-k + right_f) / (-center_f + right_f)

        f_bank[i - 1] /= (hz_points[i] - hz_points[i-1])

    filtered_frames = np.dot(frames, f_bank.T)

    filtered_frames += np.finfo(float).eps

    return filtered_frames, hz_points


def signal_power_to_db(power_frames, min_amp=1e-10, top_db=80):
    log_spec = 10.0 * np.log10(np.maximum(min_amp, power_frames))
    log_spec = np.maximum(log_spec, log_spec.max() - top_db)

    return log_spec


def discrete_cos_transformation(frames: np.array):
    rows, cols = frames.shape

    N = cols
    n = np.arange(1, N + 1)

    weights = np.zeros((N, N))
    for k in np.arange(0, N):
        weights[:, k] = np.cos(np.pi * (n - 1 / 2) * k / N)
```

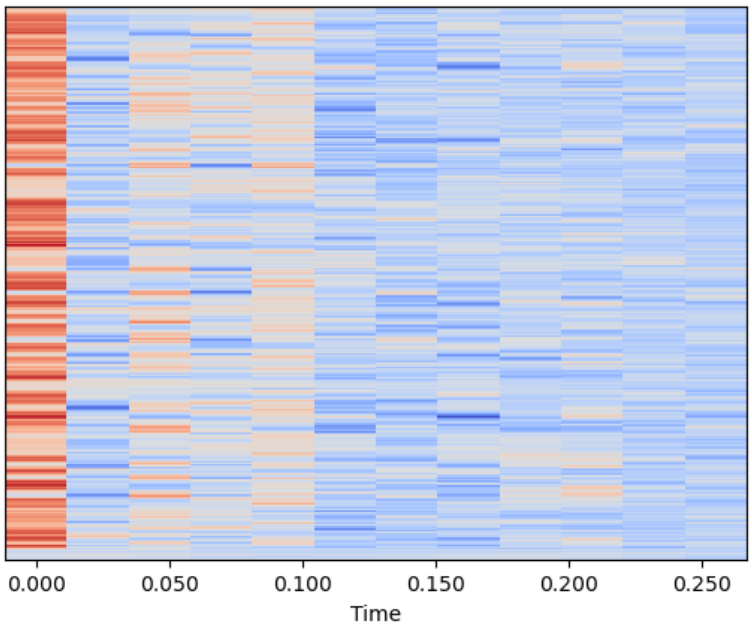
```
    dct_signal = np.sqrt(2 / N) * np.dot(frames, weights)

    return dct_signal


def sin_liftering(mfcc: np.array):
    mfcc_lift = np.zeros(mfcc.shape)

    n = np.arange(1, mfcc_lift.shape[1] + 1)
    D = 22
    w = 1 + (D / 2) * np.sin(np.pi * n / D)

    mfcc_lift = mfcc * w

    return mfcc_lift
```

[23]:
```
y, sr = load(ex('libri1'))
```

[45]:
```
mfccs = extract_mfcc_feature(y,sr)
```

[35]:
```
display.specshow(mfccs, x_axis='time')
```

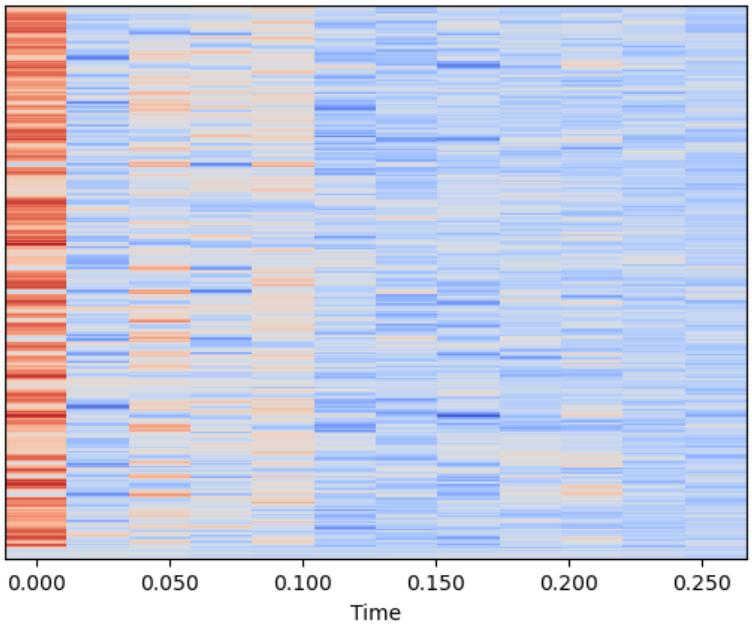
[35]:
```
<matplotlib.collections.QuadMesh at 0x138889590>
```



[47]:
```
mfccs2 = feature.mfcc(y=y, sr=sr, n_mfcc=40)
```

[38]:
```
display.specshow(mfccs, x_axis='time')
```

[38]:
```
<matplotlib.collections.QuadMesh at 0x138ffbed0>
```



[ ]:
```

```

FPGA_Numpy_Timing_Bigger_Size.ipynb

FPGA_Numpy_Timing_Bigger_Size

February 11, 2025

1 Timing Analyses of the numpy fft implementation compared to an implementation on an FPGA-Board mit $NFFT = 2^{14}$

Author: David Schulz

1.1 0. Setup

[1]:
```
import numpy as np
from time import perf_counter_ns
from pynq import Overlay, allocate
from typing import Callable, Optional
```

[2]:
```
def timeit(
    func: Callable[[np.ndarray], np.ndarray],
    values: np.ndarray,
    func_setup: Optional[Callable[[np.ndarray], None]] = lambda y: None
) -> None:
    """
    timeit is a function that times the input function with the given values

    params:
        - func: the function that is supposed to be timed
        - values: values that are used as parameters while calling the function␣
 ↪(func)

    return: None
    """

    # get the number of items inside the numpy array
    # this is the number of runs that are done
    runs: int = values.shape[0]

    # create an array with all the times for each run
    times: np.ndarray = np.zeros(runs)

    # loop over the number of runs and time the function for each loop
    for i in range(runs):
```

```
        func_setup(values[i])

        # get the start time with the performance counter in nano seconds to
        # ensure the highest precision
        start_time: int = perf_counter_ns()

        # call the function with the values for this loop
        func(values[i])

        # get the end time with the performance counter in nano seconds to
        # ensure the highest precision
        end_time: int = perf_counter_ns()

        # subtract the end time with the start time to get the time that the␣
↪function needed
        # and store this time inside the times array
        times[i] = end_time - start_time

    # calculate the mean and standard deviation in nano seconds
    mean_ns: np.float64 = np.mean(times)
    std_ns: np.float64 = np.std(times)

    # convert the mean and standard deviation to milli seconds
    mean_milli: np.float64 = mean_ns * 10**-6
    std_milli: np.float64 = std_ns * 10**-6

    # round the mean and standard deviation to 2 decimal places
    mean_milli_round: np.float64 = np.round(mean_milli, decimals=2)
    std_milli_round: np.float64 = np.round(std_milli, decimals=2)

    # print the result to the console
    # the output is supposed to be similar to the output of the built-in␣
↪package timeit
    print(f'{mean_milli_round} ms ± {std_milli_round} ms per loop (mean ± std.␣
↪dev. of {runs} runs)')
```

[3]:
```
NFFT: int = 2**14

# defined values to be used as an input to the fft functions
values = np.random.random_sample((1_000,NFFT))
```

1.2 1. Numpy Timing

[4]:
```
# defining the numpy fft function
numpy_fft: Callable[[np.ndarray], np.ndarray] = np.fft.fft
```

[5]:
```
# calling the defined timeit function to time the numpy fft function with the␣
↪predefined values
timeit(numpy_fft, values)
```

```
9.14 ms ± 0.85 ms per loop (mean ± std. dev. of 1000 runs)
```

1.3 2. FPGA FFT Timing

[6]:
```
# load the overlay of the fft block
overlay: Overlay = Overlay('FFT_test_16k.bit')

fft_data_dma = overlay.fft_data_dma
fft_config_dma = overlay.fft_config_dma

fft_data_send_channel = fft_data_dma.sendchannel
fft_data_receive_channel = fft_data_dma.recvchannel
fft_config_send_channel = fft_config_dma.sendchannel
```

[7]:
```
input_buffer = allocate(NFFT, dtype=np.csingle)
output_buffer = allocate(NFFT, dtype=np.csingle)
```

[8]:
```
def fpga_fft_setup(y: np.ndarray):
    np.copyto(input_buffer, y)
```

[9]:
```
def fpga_fft(y: np.ndarray):
    fft_data_send_channel.transfer(input_buffer)
    fft_data_receive_channel.transfer(output_buffer)
    fft_data_send_channel.wait()
    fft_data_receive_channel.wait()
```

[10]:
```
timeit(fpga_fft, values, func_setup=fpga_fft_setup)
```

```
1.36 ms ± 0.08 ms per loop (mean ± std. dev. of 1000 runs)
```

1.4 3. FPGA Timing without Transfer

[11]:
```
def fpga_fft_setup(y: np.ndarray):
    np.copyto(input_buffer, y)
    fft_data_send_channel.transfer(input_buffer)
```

[12]:
```
def fpga_fft_without_transfer(y: np.ndarray):
    fft_data_receive_channel.transfer(output_buffer)
    fft_data_send_channel.wait()
    fft_data_receive_channel.wait()
```

[13]:
```
timeit(fpga_fft_without_transfer, values, func_setup=fpga_fft_setup)
```

```
0.99 ms ± 0.02 ms per loop (mean ± std. dev. of 1000 runs)
```

2 3. FPGA Timing without Transfer 2

[14]:
```
def fpga_fft_setup_2(y: np.ndarray):
    np.copyto(input_buffer, y)
    fft_data_send_channel.transfer(input_buffer)
    fft_data_receive_channel.transfer(output_buffer)
```

[15]:
```
def fpga_fft_without_transfer_2(y: np.ndarray):
    fft_data_send_channel.wait()
    fft_data_receive_channel.wait()
```

[16]:
```
timeit(fpga_fft_without_transfer_2, values, func_setup=fpga_fft_setup_2)
```

```
0.87 ms ± 0.11 ms per loop (mean ± std. dev. of 1000 runs)
```

3 3. FPGA Timing without Transfer 3

[17]:
```
def fpga_fft_setup_3(y: np.ndarray):
    np.copyto(input_buffer, y)
    fft_data_send_channel.transfer(input_buffer)
    fft_data_receive_channel.transfer(output_buffer)
    fft_data_send_channel.wait()
```

[18]:
```
def fpga_fft_without_transfer_3(y: np.ndarray):
    fft_data_receive_channel.wait()
```

[19]:
```
timeit(fpga_fft_without_transfer_3, values, func_setup=fpga_fft_setup_3)
```

```
0.76 ms ± 0.04 ms per loop (mean ± std. dev. of 1000 runs)
```

FPGA_Numpy_Timing.ipynb

FPGA_Numpy_Timing

February 11, 2025

1 Timing Analyses of the numpy fft implementation compared to an implementation on an FPGA-Board

Author: David Schulz

1.1 0. Setup

[1]:
```
import numpy as np
from time import perf_counter_ns
from pynq import Overlay, allocate
from typing import Callable, Optional
```

[2]:
```
def timeit(
    func: Callable[[np.ndarray], np.ndarray],
    values: np.ndarray,
    func_setup: Optional[Callable[[np.ndarray], None]] = lambda y: None
) -> None:
    """
    timeit is a function that times the input function with the given values

    params:
        - func: the function that is supposed to be timed
        - values: values that are used as parameters while calling the function␣
 ↪(func)

    return: None
    """

    # get the number of items inside the numpy array
    # this is the number of runs that are done
    runs: int = values.shape[0]

    # create an array with all the times for each run
    times: np.ndarray = np.zeros(runs)

    # loop over the number of runs and time the function for each loop
    for i in range(runs):
```

```
        func_setup(values[i])

        # get the start time with the performance counter in nano seconds to
        # ensure the highest precision
        start_time: int = perf_counter_ns()

        # call the function with the values for this loop
        func(values[i])

        # get the end time with the performance counter in nano seconds to
        # ensure the highest precision
        end_time: int = perf_counter_ns()

        # subtract the end time with the start time to get the time that the␣
↪function needed
        # and store this time inside the times array
        times[i] = end_time - start_time

    # calculate the mean and standard deviation in nano seconds
    mean_ns: np.float64 = np.mean(times)
    std_ns: np.float64 = np.std(times)

    # convert the mean and standard deviation to milli seconds
    mean_milli: np.float64 = mean_ns * 10**-3
    std_milli: np.float64 = std_ns * 10**-3

    # round the mean and standard deviation to 2 decimal places
    mean_milli_round: np.float64 = np.round(mean_milli, decimals=2)
    std_milli_round: np.float64 = np.round(std_milli, decimals=2)

    # print the result to the console
    # the output is supposed to be similar to the output of the built-in␣
↪package timeit
    print(f'{mean_milli_round} µs ± {std_milli_round} µs per loop (mean ± std.␣
↪dev. of {runs} runs)')
```

[3]:
```
NFFT: int = 1024

# defined values to be used as an input to the fft functions
values = np.random.random_sample((1_000,NFFT))
```

1.2 1. Numpy Timing

[4]:
```
# defining the numpy fft function
numpy_fft: Callable[[np.ndarray], np.ndarray] = np.fft.fft
```

[5]:
```
# calling the defined timeit function to time the numpy fft function with the␣
↪predefined values
timeit(numpy_fft, values)
```

```
391.29 µs ± 28.11 µs per loop (mean ± std. dev. of 1000 runs)
```

1.3 2. FPGA FFT Timing

[6]:
```
# load the overlay of the fft block
overlay: Overlay = Overlay('FFT_test.bit')

fft_data_dma = overlay.fft_data_dma
fft_config_dma = overlay.fft_config_dma

fft_data_send_channel = fft_data_dma.sendchannel
fft_data_receive_channel = fft_data_dma.recvchannel
fft_config_send_channel = fft_config_dma.sendchannel

config_tdata_fwd: int = 1024
config_tdata_inv: int = 0

#FFT Block Config
config_input_buffer = allocate(1, dtype=np.uint16)
#output_buffer_1 = allocate(FFT_MAX_SIZE, dtype=np.csingle)
np.copyto(config_input_buffer, config_tdata_fwd)
fft_config_send_channel.transfer(config_input_buffer)
fft_config_send_channel.wait()
```

[7]:
```
input_buffer = allocate(NFFT, dtype=np.csingle)
output_buffer = allocate(NFFT, dtype=np.csingle)
```

[8]:
```
def fpga_fft_setup(y: np.ndarray):
    np.copyto(input_buffer, y)
```

[9]:
```
def fpga_fft(y: np.ndarray):
    fft_data_send_channel.transfer(input_buffer)
    fft_data_receive_channel.transfer(output_buffer)
    fft_data_send_channel.wait()
    fft_data_receive_channel.wait()
```

[10]:
```
timeit(fpga_fft, values, func_setup=fpga_fft_setup)
```

```
791.57 µs ± 56.0 µs per loop (mean ± std. dev. of 1000 runs)
```

1.4 3. FPGA Timing without Transfer

[11]:
```
def fpga_fft_setup(y: np.ndarray):
    np.copyto(input_buffer, y)
    fft_data_send_channel.transfer(input_buffer)
```

[12]:
```
def fpga_fft_without_transfer(y: np.ndarray):
    fft_data_receive_channel.transfer(output_buffer)
    fft_data_send_channel.wait()
    fft_data_receive_channel.wait()
```

[13]:
```
timeit(fpga_fft_without_transfer, values, func_setup=fpga_fft_setup)
```

```
498.37 µs ± 80.4 µs per loop (mean ± std. dev. of 1000 runs)
```

2 3. FPGA Timing without Transfer 2

[14]:
```
def fpga_fft_setup_2(y: np.ndarray):
    np.copyto(input_buffer, y)
    fft_data_send_channel.transfer(input_buffer)
    fft_data_receive_channel.transfer(output_buffer)
```

[15]:
```
def fpga_fft_without_transfer_2(y: np.ndarray):
    fft_data_send_channel.wait()
    fft_data_receive_channel.wait()
```

[16]:
```
timeit(fpga_fft_without_transfer_2, values, func_setup=fpga_fft_setup_2)
```

```
391.57 µs ± 72.68 µs per loop (mean ± std. dev. of 1000 runs)
```

3 3. FPGA Timing without Transfer 3

[17]:
```
def fpga_fft_setup_3(y: np.ndarray):
    np.copyto(input_buffer, y)
    fft_data_send_channel.transfer(input_buffer)
    fft_data_receive_channel.transfer(output_buffer)
    fft_data_send_channel.wait()
```

[18]:
```
def fpga_fft_without_transfer_3(y: np.ndarray):
    fft_data_receive_channel.wait()
```

[19]:
```
timeit(fpga_fft_without_transfer_3, values, func_setup=fpga_fft_setup_3)
```

```
292.68 µs ± 128.52 µs per loop (mean ± std. dev. of 1000 runs)
```

[ ]:
```

```

FFT_test.ipynb

FFT_test

February 11, 2025

1 FFT Block Functionality Test

Author: Tobias G. Zimmer

[1]:
```
from pynq import Overlay
from pynq.overlays.base import BaseOverlay
from pynq import allocate
import numpy as np
import matplotlib.pyplot as plt
FFT_MAX_SIZE: int = 1024
```

[2]:
```
overlay = Overlay('FFT_test.bit')
```

[3]:
```
fft_data_dma = overlay.fft_data_dma
fft_config_dma = overlay.fft_config_dma
```

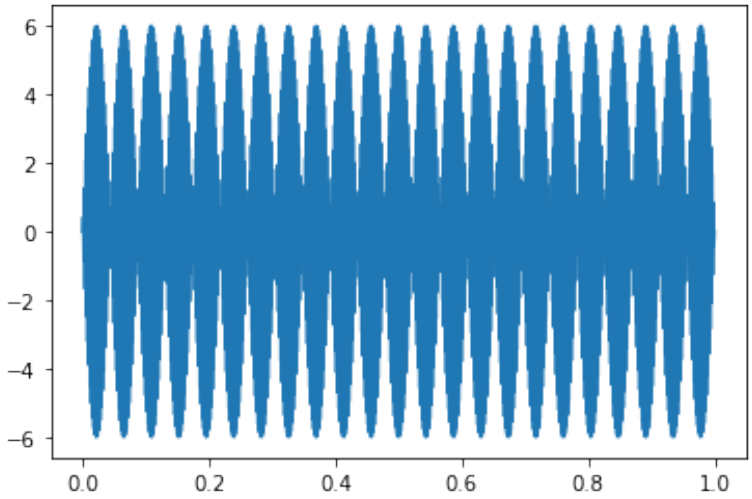
[4]:
```
fft_data_send_channel = fft_data_dma.sendchannel
fft_data_receive_channel = fft_data_dma.recvchannel
fft_config_send_channel = fft_config_dma.sendchannel
```

[5]:
```
config_tdata_fwd: int = 1024
config_tdata_inv: int = 0
```

[6]:
```
sample_points = np.linspace(0, 1,FFT_MAX_SIZE)
sample_data = 6*np.sin(2*np.pi*500*sample_points, dtype=np.csingle)
```

[7]:
```
plt.plot(sample_points, np.real(sample_data))
```

[7]:
```
[<matplotlib.lines.Line2D at 0xac451cb8>]
```

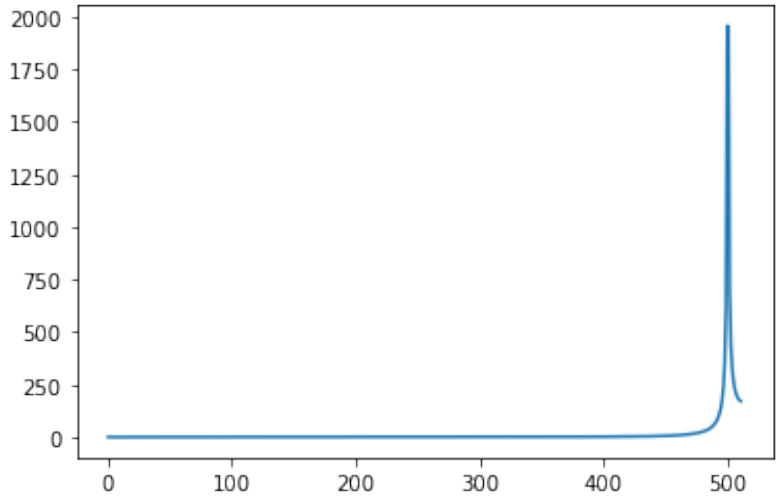


[8]:
```
%%time
#Software
software_fft = np.fft.fft(sample_data)
fft_freq = np.fft.fftfreq(FFT_MAX_SIZE, 1/FFT_MAX_SIZE)
```

```
CPU times: user 1.55 ms, sys: 0 ns, total: 1.55 ms
Wall time: 1.16 ms
```

[9]:
```
plt.plot(fft_freq[fft_freq>=0], np.abs(software_fft)[fft_freq>=0])
```

[9]:
```
[<matplotlib.lines.Line2D at 0xac3c3508>]
```



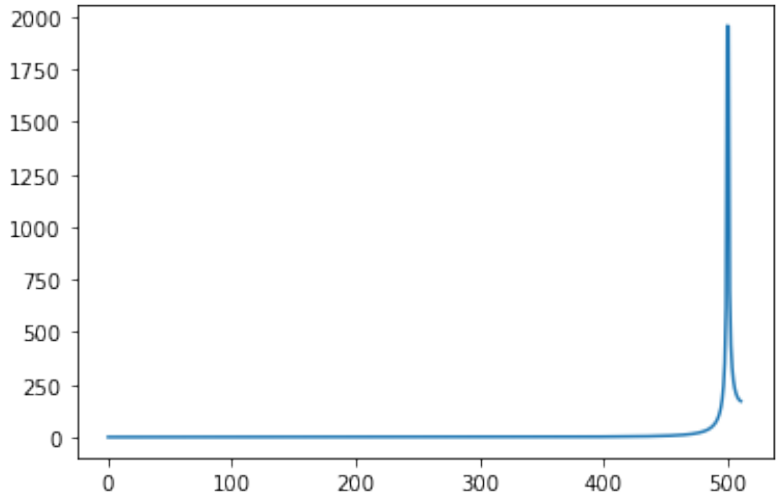
[10]:
```
#Hardware
input_buffer = allocate(FFT_MAX_SIZE, dtype=np.csingle)
output_buffer = allocate(FFT_MAX_SIZE, dtype=np.csingle)
np.copyto(input_buffer, sample_data)
```

[11]:
```
%%time
fft_data_send_channel.transfer(input_buffer)
fft_data_receive_channel.transfer(output_buffer)
fft_data_send_channel.wait()
fft_data_receive_channel.wait()
```

```
CPU times: user 1.86 ms, sys: 0 ns, total: 1.86 ms
Wall time: 2.07 ms
```

[12]:
```
plt.plot(fft_freq[fft_freq>=0], np.abs(output_buffer)[fft_freq>=0])
```

[12]:
```
[<matplotlib.lines.Line2D at 0xac3e52b0>]
```

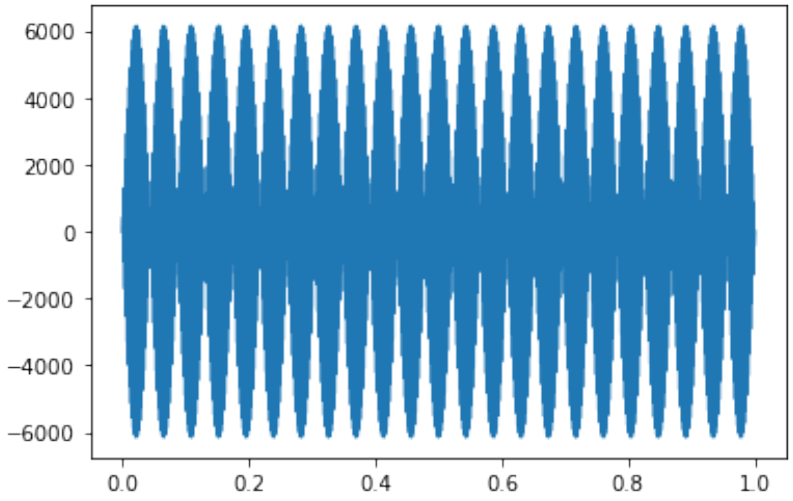


[13]:
```
#FFT Block Config
config_input_buffer = allocate(1, dtype=np.uint16)
#output_buffer_1 = allocate(FFT_MAX_SIZE, dtype=np.csingle)
np.copyto(config_input_buffer, config_tdata_inv)
fft_config_send_channel.transfer(config_input_buffer)
fft_config_send_channel.wait()
```

[14]:
```
np.copyto(input_buffer, output_buffer)
fft_data_send_channel.transfer(input_buffer)
fft_data_receive_channel.transfer(output_buffer)
fft_data_send_channel.wait()
fft_data_receive_channel.wait()
```

[15]:
```
plt.plot(sample_points, np.real(output_buffer))
```

[15]:
```
[<matplotlib.lines.Line2D at 0xabdfcb80>]
```


6000
4000
2000
0
−2000
−4000
−6000
0.0
0.2
0.4
0.6
0.8
1.0

[ ]:

B.1.3 VHDL

HammingWindow.vhd

```
-- Autor: Moses Dimmel

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.MATH_REAL.ALL;
use work.fixed_generic_pkg_mod.all;
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity HammingWindow is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           s_axis_tdata : in STD_LOGIC_VECTOR (31 downto 0);
           s_axis_tvalid : in STD_LOGIC;
           s_axis_tready : inout STD_LOGIC;
           m_axis_tdata : out STD_LOGIC_VECTOR (31 downto 0);
           m_axis_tvalid : out STD_LOGIC;
           m_axis_tready : in STD_LOGIC);
end HammingWindow;

architecture HammingWindowVerhalten of HammingWindow is

    constant WINDOW_SIZE : integer := 400;
    --attribute ram_style : string;
    --attribute ram_style of HAMMING_COEFF : signal is "block";
    --TODO: Hamming Koeffizienten in BRAM speichern
    type coeff_array is array (integer range <>) of sfixed(15 downto -16);
    signal HAMMING_COEFF : coeff_array(0 to WINDOW_SIZE - 1) := (to_sfixed
        (0.080000), sfixed(0.080057), sfixed(0.080228), sfixed(0.080513), sfixed
        (0.080912), sfixed(0.081425), sfixed(0.082052), sfixed(0.082792), sfixed
        (0.083645), sfixed(0.084612), sfixed(0.085692), sfixed(0.086884), sfixed
        (0.088189), sfixed(0.089605), sfixed(0.091134), sfixed(0.092773), sfixed
        (0.094524), sfixed(0.096385), sfixed(0.098356), sfixed(0.100437), sfixed
        (0.102626), sfixed(0.104924), sfixed(0.107330), sfixed(0.109843), sfixed
        (0.112463), sfixed(0.115189), sfixed(0.118020), sfixed(0.120956), sfixed
        (0.123996), sfixed(0.127139), sfixed(0.130384), sfixed(0.133731), sfixed
        (0.137178), sfixed(0.140726), sfixed(0.144372), sfixed(0.148117), sfixed
        (0.151959), sfixed(0.155897), sfixed(0.159930), sfixed(0.164058), sfixed
        (0.168278), sfixed(0.172591), sfixed(0.176995), sfixed(0.181489), sfixed
        (0.186072), sfixed(0.190743), sfixed(0.195500), sfixed(0.200343), sfixed
        (0.205270), sfixed(0.210280),

... [*Gekürzt für Dokumentation*]

sfixed(0.181489), sfixed(0.176995), sfixed(0.172591), sfixed(0.168278), sfixed
    (0.164058), sfixed(0.159930), sfixed(0.155897), sfixed(0.151959), sfixed
```

```
    (0.148117), sfixed(0.144372), sfixed(0.140726), sfixed(0.137178), sfixed
    (0.133731), sfixed(0.130384), sfixed(0.127139), sfixed(0.123996), sfixed
    (0.120956), sfixed(0.118020), sfixed(0.115189), sfixed(0.112463), sfixed
    (0.109843), sfixed(0.107330), sfixed(0.104924), sfixed(0.102626), sfixed
    (0.100437), sfixed(0.098356), sfixed(0.096385), sfixed(0.094524), sfixed
    (0.092773), sfixed(0.091134), sfixed(0.089605), sfixed(0.088189), sfixed
    (0.086884), sfixed(0.085692), sfixed(0.084612), sfixed(0.083645), sfixed
    (0.082792), sfixed(0.082052), sfixed(0.081425), sfixed(0.080912), sfixed
    (0.080513), sfixed(0.080228), sfixed(0.080057), sfixed(0.080000));
    -- mit python script ermittelt, gespeichert im BRAM zur Optimierung
    signal index : integer range 0 to WINDOW_SIZE - 1 := 0;

begin
    process(clk, reset)
    begin
        if reset = '1' then
            index <= 0;
            m_axis_tdata <= (others => '0');
            s_axis_tready <= '0';
        elsif rising_edge(clk) then
            if s_axis_tvalid = '1' and s_axis_tready = '1' then
                -- Hamming Gewichtung durchfuehren
                m_axis_tdata <= std_logic_vector(to_signed(
                    (to_sfixed(signed(s_axis_tdata), 15, -16)) * HAMMING_COEFF(index
                        ), 32));
                m_axis_tvalid <= '1';
                s_axis_tready <= '0';

                index <= index + 1;
                if index = WINDOW_SIZE - 1 then
                    index <= 0;
                end if;
            elsif m_axis_tready = '1' then
                -- Output akzeptiert, Signal zuruecksetzen
                m_axis_tvalid <= '0';
                s_axis_tready <= '1';
            end if;
        end if;
    end process;

end HammingWindowVerhalten;
```

Listing B.5: Quellcode für die Hamming-Fensterung via VHDL.

MagnitudeCalculator.vhd

```
-- Autor: Moses Dimmel

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
```

```
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity MagnitudeCalculator is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           fft_tdata : in STD_LOGIC_VECTOR (31 downto 0);
           fft_tvalid : in STD_LOGIC;
           fft_tready : inout STD_LOGIC;
           magnitude_tdata : out STD_LOGIC_VECTOR (31 downto 0);
           magnitude_tvalid : out STD_LOGIC;
           magnitude_tready : in STD_LOGIC);
end MagnitudeCalculator;

architecture MagnitudeCalculatorVerhalten of MagnitudeCalculator is

    signal re, im : signed(15 downto 0); -- Real- und Imaginaeranteil
    signal abs_re, abs_im : unsigned(15 downto 0); -- Absolute Werte
    signal magnitude : unsigned(31 downto 0); -- Berechnete Magnitude
    signal valid_next : std_logic; -- Next valid Zustand
begin
    process(clk, reset)
    begin
        if reset = '1' then
            magnitude_tdata <= (others => '0');
            magnitude_tvalid <= '0';
            fft_tready <= '1';
        elsif rising_edge(clk) then
            -- Pruefen ob input valide und bereit ist
            if fft_tvalid = '1' and fft_tready = '1' then
                -- Reellen und Imaginaeren Teil extrahieren
                re <= signed(fft_tdata(31 downto 16));
                im <= signed(fft_tdata(15 downto 0));

                -- Betraege verarbeiten
                abs_re <= unsigned(abs(re));
                abs_im <= unsigned(abs(im));

                -- Magnitude Berechnen (Angenaehert durch: |Re| + |Im|)
                magnitude <= abs_re + abs_im;

                -- Ergebnisse ausgeben wenn Ready-Signal gegeben wurde
                if magnitude_tready = '1' then
                    magnitude_tdata <= std_logic_vector(magnitude);
                    magnitude_tvalid <= '1';
                else
                    -- Valide-Signal halten bis Ready-Signal gegeben wurde
                    valid_next <= '1';
                end if;
```

```
            else
                -- Valide-Signal zuruecksetzen wenn kein Inputsignal existiert oder
                    Output nicht valide ist
                magnitude_tvalid <= '0';
            end if;
        end if;
    end process;

end MagnitudeCalculatorVerhalten;
```

Listing B.6: Quellcode für die Berechnung des Betrages via VHDL.

dct.vhd

```
-- Autor: Moses Dimmel

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;


entity DCT is
    generic (
        N_POINTS : integer := 400; -- Anzahl an Input Punkte der Mel filterbaenke
        N_COEFFS : integer := 13; -- Anzahl an output-Koeffizienten (MFCCs)
        DATA_WIDTH : integer := 32 -- Bitbreite von input und ouput-Daten
    );
    port (
        clk : in std_logic;
        reset : in std_logic;
        dct_tvalid : in std_logic; -- Valider-Input-Signal
        dct_tready : out std_logic; -- Bereit-zu-Empfangen-Signal
        input_data : in std_logic_vector((N_POINTS * DATA_WIDTH) - 1 downto 0); --
            Input vector
        dct_tvalid_out : out std_logic; -- Output valid signal
        dct_output : out std_logic_vector((N_COEFFS * DATA_WIDTH) - 1 downto 0) --
            Output vector
    );
end DCT;


architecture Behavioral of DCT is
    -- Konstantendeklaration (Vorberechnete Kosinus-Matrix)
    --type cosine_matrix_type is array(0 to N_COEFFS-1, 0 to N_POINTS-1) of real;
```

```
   type cosine_matrix_type is array(0 to 12, 0 to 399) of real;

   --TODO: Speicherung der Cosinus-Matrix in BRAM
   constant COSINE_MATRIX : cosine_matrix_type := (
   (1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000,
      1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000,
      1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000,
      1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000,
      1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000,
      1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000,
      1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000,
      1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000,
      1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000,
      1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000,
      1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000,
      1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000,
      1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000,
      1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000,
      1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000,
      1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000,
      1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000,
      1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000,
      1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000,
      1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000,
      1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000,
      1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000,
      1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000, 1.0000000000,

... [*Für die Dokumentation gekürzt*]

-0.6494480483, -0.5750052520, -0.4954586684, -0.4115143586, -0.3239174182,
   -0.2334453639, -0.1409012319, -0.0471064507, 0.0471064507, 0.1409012319,
   0.2334453639, 0.3239174182, 0.4115143586, 0.4954586684, 0.5750052520,
   0.6494480483, 0.7181262978, 0.7804304073, 0.8358073614, 0.8837656301,
   0.9238795325, 0.9557930148, 0.9792228106, 0.9939609555, 0.9998766325,
   0.9969173337, 0.9851093262, 0.9645574185, 0.9354440308, 0.8980275758,
   0.8526401644, 0.7996846585, 0.7396310950, 0.6730125135, 0.6004202253,
   0.5224985647, 0.4399391699, 0.3534748438, 0.2638730500, 0.1719291003,
   0.0784590957, -0.0157073173, -0.1097343111, -0.2027872954, -0.2940403252,
   -0.3826834324, -0.4679298143, -0.5490228180, -0.6252426563, -0.6959127966,
   -0.7604059656, -0.8181497174, -0.8686315144, -0.9114032766, -0.9460853588,
   -0.9723699204, -0.9900236577, -0.9988898750, -0.9988898750, -0.9900236577,
   -0.9723699204, -0.9460853588, -0.9114032766, -0.8686315144, -0.8181497174,
   -0.7604059656, -0.6959127966, -0.6252426563, -0.5490228180, -0.4679298143,
   -0.3826834324, -0.2940403252, -0.2027872954, -0.1097343111, -0.0157073173,
   0.0784590957, 0.1719291003, 0.2638730500, 0.3534748438, 0.4399391699,
   0.5224985647, 0.6004202253, 0.6730125135, 0.7396310950, 0.7996846585,
   0.8526401644, 0.8980275758, 0.9354440308, 0.9645574185, 0.9851093262,
   0.9969173337, 0.9998766325, 0.9939609555, 0.9792228106, 0.9557930148,
   0.9238795325, 0.8837656301, 0.8358073614, 0.7804304073, 0.7181262978,
   0.6494480483, 0.5750052520, 0.4954586684, 0.4115143586, 0.3239174182,
   0.2334453639, 0.1409012319, 0.0471064507, -0.0471064507, -0.1409012319,
```

```
    -0.2334453639, -0.3239174182, -0.4115143586, -0.4954586684, -0.5750052520,
    -0.6494480483, -0.7181262978, -0.7804304073, -0.8358073614, -0.8837656301,
    -0.9238795325, -0.9557930148, -0.9792228106, -0.9939609555, -0.9998766325,
    -0.9969173337, -0.9851093262, -0.9645574185, -0.9354440308, -0.8980275758,
    -0.8526401644, -0.7996846585, -0.7396310950, -0.6730125135, -0.6004202253,
    -0.5224985647, -0.4399391699, -0.3534748438, -0.2638730500, -0.1719291003,
    -0.0784590957, 0.0157073173, 0.1097343111, 0.2027872954, 0.2940403252,
    0.3826834324, 0.4679298143, 0.5490228180, 0.6252426563, 0.6959127966,
    0.7604059656, 0.8181497174, 0.8686315144, 0.9114032766, 0.9460853588,
    0.9723699204, 0.9900236577, 0.9988898750)
);

    -- Signale fuer interne Berechnungen
    signal input_fixed : array(0 to 400-1) of signed(DATA_WIDTH-1 downto 0);
    signal output_fixed : array(0 to 13-1) of signed(DATA_WIDTH-1 downto 0);
    signal ready : std_logic := '1';
    signal valid_out : std_logic := '0';

begin
    dct_tready <= ready;

    -- Convert input vector into array
    process(clk)
    begin
        if reset = '1' then
            dct_output <= (others => '0')
            ready <= '1';
            valid_out <= '0';
        elsif rising_edge(clk) then
            if dct_tvalid = '1' and ready = '1' then
                -- Convert flat vector into array of signed numbers
                for i in 0 to N_POINTS-1 loop
                    input_fixed(i) <= signed(input_data(DATA_WIDTH * (i + 1) - 1
                        downto DATA_WIDTH * i));
                end loop;

                -- Perform DCT calculation
                for k in 0 to N_COEFFS-1 loop
                    output_fixed(k) <= (others => '0'); -- Reset output
                    for n in 0 to N_POINTS-1 loop
                        output_fixed(k) <= output_fixed(k) +
                            signed(to_integer(unsigned(input_fixed(n))) *
                            to_integer(real_to_sfixed(COSINE_MATRIX(k, n), DATA_WIDTH,
                                DATA_WIDTH-1)));
                    end loop;
                end loop;

                valid_out <= '1';
                ready <= '0';
            elsif valid_out = '1' then
                -- Reset after data is sent out
                valid_out <= '0';
                ready <= '1';
```

```
            end if;
        end if;
    end process;

    -- Output valid signal
    dct_tvalid_out <= valid_out;

    -- Convert array to flat vector for output
    dct_output <= (others => '0');
    process(clk)
    begin
        if rising_edge(clk) then
            if valid_out = '1' then
                for k in 0 to N_COEFFS-1 loop
                    dct_output(DATA_WIDTH * (k + 1) - 1 downto DATA_WIDTH * k) <=
                        std_logic_vector(output_fixed(k));
                end loop;
            end if;
        end if;
    end process;

end Behavioral;
```

Listing B.7: Quellcode für die Berechnung der DCT via VHDL.

generate_coeff.py

```python
# Author: Moses Dimmel
import numpy as np

WINDOW_SIZE = 400

# Berechne Hamming-Koeffizienten
hamming_coeff = 0.54 - 0.46 * np.cos(2 * np.pi * np.arange(WINDOW_SIZE) / (WINDOW_SIZE - 1))

# Generiere VHDL-Konstante
#vhdl_array = ", 32, 0), sfixed(".join(f"{coeff:.6f}" for coeff in hamming_coeff)
#print(f"constant HAMMING_COEFF : real_vector(0 to {WINDOW_SIZE-1}) := (to_sfixed({vhdl_array}, 32, 0));")

vhdl_array = "), sfixed(".join(f"{coeff:.6f}" for coeff in hamming_coeff)
print(f"constant HAMMING_COEFF : real_vector(0 to {WINDOW_SIZE-1}) := (to_sfixed({vhdl_array}));")
```

Listing B.8: Hilfs-Script zur Generierung der Hamming-Coeffizienten als VHDL-Konstante.

generat_cosine_values.py

```python
# Author: Moses Dimmel
import numpy as np

def generate_cosine_vhdl(N, K):
    """
    Generiert eine Liste von vorberechneten Kosinus Werten fuer die DCT entsprechend der VHDL
    Syntax.

    Parameters:
    - N: int, Anzahl an Input-Samples.
```

```
    - K: int, Anzahl an DCT-Koeffizienten.

    Returns:
    - Einen String mit dem VHDL-Code.
    """

    constant_name="COSINE_VALUES"

    # Kosinus-Werte berechnen
    cosine_matrix = np.zeros((K, N))
    for k in range(K):
        for n in range(N):
            cosine_matrix[k, n] = np.cos((np.pi / N) * (n + 0.5) * k)

    # Werte in VHDL-Syntax formatieren
    vhdl_lines = []
    vhdl_lines.append(f"constant {constant_name} : array(0 to {K-1}, 0 to {N-1}) of real := (")

    for k in range(K):
        row_values = ", ".join(f"{value:.10f}" for value in cosine_matrix[k])
        vhdl_lines.append(f"    ({row_values})" + ("," if k < K-1 else ""))

    vhdl_lines.append(");")
    return "\n".join(vhdl_lines)


N = 400
K = 13
vhdl_code = generate_cosine_vhdl(N, K)

with open("cosine_values_vhdl.txt", "w") as file:
    file.write(vhdl_code)

print("Cosine Values have been written to cosine_values_vhdl.txt")
```

Listing B.9: Hilfs-Script zur Generierung von Kosinus-Werten für die DCT-Berechnung als VHDL-Konstante.

B.1.4 HLS

mel_coefficients.cpp

```
// Author: Moses Dimmel
#include <hls_math.h>
#include "mel_coefficients.h"
#include "mel_filterbank_lut.h"
#include "hamming_window.h"

#define FFT_SIZE 512
#define MEL_BANDS 20


void dct(fixed_t mel_energy[MEL_BANDS], fixed_t mfcc[MEL_BANDS], axis_fft_stream
    &fft_out_axis, axis_fft_stream &fft_in_axis) {
        int DOUBLE_MEL_BANDS = 2 * MEL_BANDS;

        // TODO: configure fft scaling
        int factor = 8;

        // send input to fft ip block
        for (int i = 0; i < MEL_BANDS; i++) {
                #pragma HLS UNROLL

                while (fft_in_axis.full());
                axis_fft_t in_sample;

                in_sample.data.range(63, 32) = 0;
                in_sample.data.range(31, 0) = mel_energy[i];
                in_sample.keep = 0xF;
                fft_in_axis.write(in_sample);
        }
        for (int i = MEL_BANDS-1 ; i >= 0; i--) {
                #pragma HLS UNROLL

                while (fft_in_axis.full());
                axis_fft_t in_sample;

                in_sample.data.range(63, 32) = 0;
                in_sample.data.range(31, 0) = mel_energy[i];
                in_sample.keep = 0xF;
                in_sample.last = (i == 0);
                fft_in_axis.write(in_sample);
        }

        // read ouput of fft ip block
        for (int i = 0; i < DOUBLE_MEL_BANDS; i++) {
                #pragma HLS UNROLL

                while (fft_out_axis.empty());
                fixed_t real;
                axis_fft_t out_sample;
```

```
            // get real part
            out_sample = fft_out_axis.read();
            real.range(31, 0) = out_sample.data.range(31, 0);

            // undo scaling from fft ip block output
            real = real * factor;

            // add to output array
            if (i < MEL_BANDS) {
                  mfcc[i] = real;
            }
      }
}

// Aufruf des FFT IP Blocks
void fft_transform(fixed_t windowed_audio[FFT_SIZE], fixed_t fft_magnitude[
   FFT_SIZE], axis_fft_stream &fft_out_axis, axis_fft_stream &fft_in_axis) {
      // TODO: Configure scaling of fft output
      int factor = 8;

      for (int i = 0; i < FFT_SIZE; i++) {
            #pragma HLS UNROLL

            while (fft_in_axis.full());
            axis_fft_t in_sample;

            in_sample.data.range(63, 32) = 0;
            in_sample.data.range(31, 0) = windowed_audio;
            in_sample.keep = 0xF;
            in_sample.last = (i == FFT_SIZE-1);
            fft_in_axis.write(in_sample);
      }


      for (int i = 0; i < FFT_SIZE; i++) {
            #pragma HLS UNROLL

            while (fft_out_axis.empty());
            fixed_t real, imag;
            axis_fft_t out_sample;

            out_sample = fft_out_axis.read();
            real.range(31, 0) = out_sample.data.range(31, 0);
            imag.range(31, 0) = out_sample.data.range(63, 32);

            // undo scaling from fft ip block output
            real = real * factor;
            imag = imag * factor;

            // calculate magnitude
            fft_magnitude[i] = hls::sqrt(real * real + imag * imag);
      }
}
```

```
// Mel-Filterbank-Berechnung
void mel_cepstral_coefficient_calculation(axis_stream &audio_in_axis, axis_stream
     &mfcc_out_axis, axis_fft_stream &fft_out_axis, axis_fft_stream &fft_in_axis)
    {
    #pragma HLS INTERFACE axis port=audio_in_axis
    #pragma HLS INTERFACE axis port=mfcc_out_axis
       #pragma HLS INTERFACE axis port=fft_out_axis
       #pragma HLS INTERFACE axis port=fft_in_axis
    #pragma HLS PIPELINE

       fixed_t windowed_audio[FFT_SIZE];
       fixed_t fft_real[FFT_SIZE];
       fixed_t fft_imag[FFT_SIZE] = {0};
       fixed_t fft_magnitude[FFT_SIZE];
       fixed_t mel_energy[MEL_BANDS];
       fixed_t mfcc[MEL_BANDS];

       // Input und Hamming-Fensterung
       axis_t in_sample;
       for (int i = 0; i < FFT_SIZE; i++) {
              #pragma HLS UNROLL factor=4
              while (audio_in_axis.empty());

              in_sample = audio_in_axis.read();

              if (in_sample.keep == 0xF) {
                     continue;
              }

              windowed_audio[i] = in_sample.data * hamming_window[i];

              if (in_sample.last) {
                     break;
              }
       }

       // FFT-Berechnung
       fft_transform(windowed_audio, fft_magnitude, fft_out_axis, fft_in_axis);

       // Mel-Filterbank anwenden
       for (int m = 0; m < MEL_BANDS; m++) {
              #pragma HLS UNROLL
              mel_energy[m] = 0.0;

              for (int k = 0; k < FFT_SIZE; k++) {
                     mel_energy[m] += fft_magnitude[k] * mel_filterbank_lut[m][k];
              }

              // Logarithmus-Approximation
              mel_energy[m] = hls::log(hls::abs(mel_energy[m]) + fixed_t(1e-6));
       }
```

```
	// DCT fuer MFCCs
	dct(mel_energy, mfcc, fft_out_axis, fft_in_axis);

	// MFCC-Werte ausgeben
	axis_t out_sample;
	for (int i = 0; i < MEL_BANDS; i++) {
		#pragma HLS UNROLL
		while (mfcc_out_axis.full());

		out_sample.data = mfcc[i];
		out_sample.keep = 0xF;
		out_sample.last = (i == MEL_BANDS-1);
		mfcc_out_axis.write(out_sample);
	}
}
```

Listing B.10: Quellcode für die Mel Koeffizienten Berechnung via HLS.

mel_coefficients.h

```
// Author: Moses Dimmel
#ifndef MEL_FILTERBANK_LUT_H
#define MEL_FILTERBANK_LUT_H

#include <ap_fixed.h>
#include <ap_axi_sdata.h>
#include <hls_stream.h>

#define MEL_BANDS 20
#define FFT_SIZE 512
#define SAMPLE_RATE 48000

typedef ap_fixed<32, 16> fixed_t;

// AXI Stream Datentyp

typedef ap_axis<32,2,5,6> axis_t;
typedef ap_axis<64,2,5,6> axis_fft_t;

typedef hls::stream<axis_t> axis_stream;
typedef hls::stream<axis_fft_t> axis_fft_stream;

extern void dct_approx(fixed_t* , fixed_t*);
extern void mel_cepstral_coefficient_calculation(axis_stream&, axis_stream&);

#endif // MEL_FILTERBANK_LUT_H
```

Listing B.11: Quellcode für die Header-Datei der Mel Koeffizienten Berechnung via HLS.

B.2 Audio

B.2.1 record.py

```python
#Authors: Fynn Schur, Jesse Gollub, Mattes Bielefeld
from enum import Enum

from pynq.overlays.base import BaseOverlay


class AudioSource(Enum):
    """
    Audio sources

    MIC: Microphone jack
    LINEIN: Line in jack
    """

    MIC = 0
    LINEIN = 1


class Audio:
    def __init__(self, base: BaseOverlay, audio_source: AudioSource):
        self.base = base
        self.audio = base.audio
        if audio_source == AudioSource.MIC:
            self.audio.select_microphone()
        elif audio_source == AudioSource.LINEIN:
            self.audio.select_linein()

    def record(self, duration: float):
        self.audio.record(duration)
        return self.buffer.copy()

    @property
    def buffer(self):
        return self.audio.buffer
```

Listing B.12: Quellcode für die kontinuierliche Audioaufzeichnung

B.2.2 audio_io.py

```python
#Authors: Fynn Schur, Jesse Gollub, Mattes Bielefeld
from numpy import ndarray
from pynq import MMIO


def write(mmio: MMIO, address: int, data: ndarray):
    bytes = data.tobytes()
    print("bytelength", len(bytes))
    mmio.write(address, bytes)


def read(mmio: MMIO, address: int, size: int):
    return mmio.read(address, size)
```

Listing B.13: Quellcode für die Übertragung der Audiodaten

B.2.3 main.py

```python
#Authors: Fynn Schur, Jesse Gollub, Mattes Bielefeld

from collections import deque

import numpy as np
from pynq import MMIO
from pynq.overlays.base import BaseOverlay

from audio_io import read, write
from record import Audio, AudioSource

IP_BASE_ADDRESS = 0x40000000
ADDRESS_RANGE = 0x2000
ADDRESS_OFFSET = 0x10
WINDOW_SIZE = 2


def main():
    audio = Audio(BaseOverlay("base.bit"), AudioSource.MIC)
    mmio = MMIO(IP_BASE_ADDRESS, ADDRESS_RANGE)
    queue = deque(maxlen=WINDOW_SIZE)

    while True:
        data = audio.record(0.01)

        queue.append(data)
        # if queue is not full, continue
        if len(queue) < WINDOW_SIZE:
            continue

        write(mmio, ADDRESS_OFFSET, np.concatenate(queue))

        print(read(mmio, ADDRESS_OFFSET, WINDOW_SIZE))


if __name__ == "__main__":
    main()
```

Listing B.14: Integratio der Audioaufzeichnung und -übertragung

B.2.4 test_continuous_audio_in.py

```python
#Author: Mattes Bielefeld
import numpy as np
import math
import wave
import struct
from time import sleep
from collections import deque
from pynq.overlays.base import BaseOverlay

from record import Audio, AudioSource

SAMPLE_RATE = 48000
TEST_SECONDS = 5

# Diese Methode ist inspiriert und zum Teil kopiert von den Dateien zum BaseOverlay von PYNQ,
# zu finden im PYNQ GitHub unter pynq/lib/audio.py in der Methode save der Klasse AudioADAU1761
def write_file(buffer, file, sample_len, sample_rate):
    samples_4byte = buffer.tobytes()
    byte_format = ("%ds %dx " % (3, 1)) * sample_len * 2
    samples_3byte = b"".join(struct.unpack(byte_format, samples_4byte))
    with wave.open(file, "wb") as wav_file:
        # Set the number of channels
```

```
        wav_file.setnchannels(2)
        # Set the sample width to 3 bytes
        wav_file.setsampwidth(3)
        # Set the frame rate to sample_rate
        wav_file.setframerate(sample_rate)
        # Set the number of frames to sample_len
        wav_file.setnframes(sample_len)
        # Set the compression type and description
        wav_file.setcomptype("NONE", "not compressed")
        # Write data
        wav_file.writeframes(samples_3byte)


def main():
    audio = Audio(BaseOverlay("base.bit"), AudioSource.MIC)
    test_duration = TEST_SECONDS * 1000 # to scale recording block up to 5 second (5000ms)
    sample_len = math.ceil(TEST_SECONDS * SAMPLE_RATE)
    file1 = "/home/xilinx/jupyter_notebooks/base/audio/test_new.wav"
    file2 = "/home/xilinx/jupyter_notebooks/base/audio/audio_comparison_test_new.wav"
    buffer = np.zeros(0, dtype=np.int32)

    print("start recording 1")
    data1 = audio.record(TEST_SECONDS)
    write_file(data1, file2, sample_len, SAMPLE_RATE)

    print ("start recording 2")
    for i in range(test_duration):
        duration = 0.01
        data2 = audio.record(duration)
        buffer = np.append(buffer, data2, axis=None)

    write_file(buffer, file1, sample_len, SAMPLE_RATE)


if __name__ == "__main__":
    main()
```

Listing B.15: Test der Audioaufzeichnung

B.2.5 test_data_transmission.py

```
#Author: Mattes Bielefeld
from collections import deque
import numpy as np
from pynq import MMIO
from pynq.overlays.base import BaseOverlay

from audio_io import read, write
from record import Audio, AudioSource

IP_BASE_ADDRESS = 0x40000000
ADDRESS_RANGE = 0x2000
ADDRESS_OFFSET = 0x10
WINDOW_SIZE = 2
TEST_DATA = b'00000000010101011111000000001111'

def main():
    mmio = MMIO(IP_BASE_ADDRESS, ADDRESS_RANGE)
    queue = deque(maxlen=WINDOW_SIZE)


    # Inlcude Windwoing for the Test Data
    while True:
        array = np.zeros(0)
```

```
24          array = np.append(array, TEST_DATA)
25          queue.append(array)
26          # if queue is not full, continue
27          if len(queue) < WINDOW_SIZE:
28              continue
29
30          write(mmio, ADDRESS_OFFSET, np.concatenate(queue))
31
32          print(read(mmio, ADDRESS_OFFSET, 2))
33
34
35  if __name__ == "__main__":
36      main()
```

Listing B.16: Test der Datenübertragung

B.2.6 flake.nix

```
# Author: Jesse Gollub
{
    description = "Python development template";

    inputs = {
        nixpkgs.url = "github:NixOS/nixpkgs/nixos-unstable";
        utils.url = "github:numtide/flake-utils";
    };

    outputs = {
        nixpkgs,
        utils,
        ...
    }:
        utils.lib.eachDefaultSystem (system: let
            pkgs = import nixpkgs {inherit system;};
            devDeps = with pkgs.python312Packages; [
                pytest
                black
                isort
                numpy
            ];
        in {
            devShells.default = pkgs.mkShell {
                buildInputs = devDeps;
            };

            checks.format =
                pkgs.runCommand "python-format" {
                    src = ./.;
                    buildInputs = devDeps;
                } ''
                    black --check $src
                    isort $src --check --diff --known-local-folder "src" --profile "
                        black"
                    touch $out
                '';
```

```
      });
}
```

Listing B.17: Entwicklungsumgebung mit Nix

B.2.7 README.md

````
# Audio

> Verfasst von Jesse Gollub

Dieses Verzeichnis enthält den Code zur Audioaufnahme mit [PYNQ](https://github.
    com/Xilinx/PYNQ) in Python.

## Entwicklung

Um alle Abhängigkeiten zu installieren, führen Sie den folgenden Befehl aus:

```bash
nix develop
```

Alternativ kann direnv verwendet werden, um die Umgebung automatisch zu laden.
    Dazu gibt es eine .envrc-Datei, die mit folgendem Befehl aktiviert wird:

```bash
direnv allow
```

> Hinweis: Die devShell enthält nicht die PYNQ-Bibliothek, da diese nicht über
    herkömmliche Paketmanager installiert werden kann.

Alternativ kann die Entwicklung direkt auf dem PYNQ-Board erfolgen, da dort
    bereits alle benötigten Bibliotheken vorliegen.

## Nutzung

Zum Starten des Projekts führen Sie folgenden Befehl aus:

```bash
python -m src
```

## Tests

Die Testdateien befinden sich im Verzeichnis tests und können mit folgendem
    Befehl ausgeführt werden:

```bash
python -m tests
```
````

Listing B.18: Erklärung zur Nutzung des Audio-Codes

B.3 FINN-Framework

B.3.1 create_ip_poc.ipynb

create_ip_poc

February 21, 2025

1 Notebook zum Erzeugen eines IP-Blocks aus einem quantisierten neuronalen Netz

Autoren: Marvin Hoyer, Thomas Klein und Simon Sajnog

Dieses Jupyter-Notebook erzeugt aus einem quantisierten neuronalen Netz im ONNX-Format einen IP-Block, welcher in die Vivado Design Suite eingebunden werden kann. Da dieses Notebook das FINN-Framework verwendet, muss es in einem Docker-Container ausgeführt werden, welcher die Abhängigkeiten zu FINN enthält.

Dieses Notebook stellt ein Proof of Concept zum Erzeugen eines IP Blocks dar, da das endgültige neuronale Netz des Projektes zum Zeitpunkt des Schreibens dieser Dokumentation noch nicht vorhanden war.

Der Code ist eine gemeinsame Arbeit und basiert auf verschiedenen Tutorial-Notebooks von FINN. Der Export zu einem IP-Block ist eine angepasste Version eines Code-Teils aus dem FINN-Notebook 3-build-accelerator-with-finn.ipynb.

1.1 Haupt-Dependencies importieren

[1]:
```
import onnx
import torch
```

1.2 Laden des neuronalen Netzes aus einer ONNX-Datei

Das neuronale Netz wird mithilfe der Klasse `ModelWrapper` in FINN als Modell geladen.

[2]:
```
import os
from qonnx.core.modelwrapper import ModelWrapper
from qonnx.core.datatype import DataType
from finn.transformation.qonnx.convert_qonnx_to_finn import ConvertQONNXtoFINN

# Das Verzeichnis der ONNX-Datei
model_dir = "."
# Das relative Verzeichnis für die Ausgabegabedatein des Notebooks
builder_dir = model_dir + "/builder_output"
# Der Dateiname der ONNX-Datei
ready_model_filename = model_dir + "/DefaultModel.onnx"
```

1

[ ]:
```
from finn.util.visualization import showInNetron, showSrc

# ModelWrapper wird zum Laden der ONNX-Datei in FINN benoetigt
# Zudem werden hierdurch verschiedene Hilfsfunktionen fuer das Modell zur␣
 ↪Verfuegung gestellt
model = ModelWrapper(ready_model_filename)

# Visualisierung des neuronalen Netzes
showInNetron(ready_model_filename)
```

```
Stopping http://0.0.0.0:8081
Serving './DefaultModel.onnx' at http://0.0.0.0:8081
```

[ ]:
```
<IPython.lib.display.IFrame at 0x7795d7e57af0>
```

1.3 Notwendige Transformationen auf das Modell anwenden

Für die weiteren Verarbeitungsschritte müssen einige Transformationen auf das importierte Modell angewandt werden.

[5]:
```
from qonnx.transformation.general import GiveUniqueNodeNames,␣
 ↪GiveReadableTensorNames, RemoveStaticGraphInputs
from qonnx.transformation.infer_datatypes import InferDataTypes
from qonnx.transformation.infer_shapes import InferShapes
from qonnx.transformation.fold_constants import FoldConstants

model = model.transform(InferShapes()) # Ableiten der Tensoren-Formen aus den␣
 ↪Modell-Eigenschaften
model = model.transform(FoldConstants()) # Faltung von konstanten␣
 ↪Netzwerkteilen => Vereinfachung des Netzwerks
model = model.transform(GiveUniqueNodeNames()) # Vergabe von eindeutigen␣
 ↪Bezeichnern fuer die Netzwerkt-Nodes
model = model.transform(GiveReadableTensorNames()) # Vergabe von fuer Menschen␣
 ↪lesbare Namen fuer Tensoren
model = model.transform(InferDataTypes()) # Ableiten der Tensoren-Datentypen␣
 ↪aus den Modell-Eigenschaften
model = model.transform(RemoveStaticGraphInputs()) # Entfernen von␣
 ↪Netzwerk-Eingaengen, welche schon ONNX-Initialisierer verwenden

# Speichern des transformierten Modells als Zwischenausgabe
model.save(builder_dir + "/model_tidy.onnx")
```

[6]:
```
# Visualisierung des transfomierten Modells
model = ModelWrapper(builder_dir + "/model_tidy.onnx")
showInNetron(builder_dir + "/model_tidy.onnx")
```

```
Stopping http://0.0.0.0:8081
Serving './builder_output/model_tidy.onnx' at http://0.0.0.0:8081
```

[6]:
```
<IPython.lib.display.IFrame at 0x7795d7e564a0>
```

1.4 Umwandlung der Netzwerk-Layer in Hardware-Layer

[7]:
```
from finn.transformation.fpgadataflow.create_dataflow_partition import␣
↪CreateDataflowPartition

model = ModelWrapper(builder_dir + "/model_tidy.onnx")
parent_model = model.transform(CreateDataflowPartition())
parent_model.save(builder_dir + "/parent.onnx")
```

[8]:
```
# Visualisierung des Modells mit Hardware-Layern
model = ModelWrapper(builder_dir + "/parent.onnx")
showInNetron(builder_dir + "/parent.onnx")
```

```
Stopping http://0.0.0.0:8081
Serving './builder_output/parent.onnx' at http://0.0.0.0:8081
```

[8]:
```
<IPython.lib.display.IFrame at 0x7795d7e56bf0>
```

1.5 Finale Vorbereitung des Modells für die IP-Block-Erstellung

[9]:
```
# Auflistung der verfuegbaren FPGA-Boards
from finn.util.basic import pynq_part_map
print(pynq_part_map.keys())
```

```
dict_keys(['Ultra96', 'Ultra96-V2', 'Pynq-Z1', 'Pynq-Z2', 'ZCU102', 'ZCU104',
'ZCU111', 'RFSoC2x2', 'RFSoC4x2', 'KV260_SOM'])
```

[10]:
```
# FPGA-Board festlegen
pynq_board = "Pynq-Z2"
# FPGA-Board-Informationen auswaehlen
fpga_part = pynq_part_map[pynq_board]
# Takt-Frequenz festlegen
target_clk_ns = 10
```

[11]:
```
from finn.transformation.fpgadataflow.make_zynq_proj import ZynqBuild
model = ModelWrapper(builder_dir + "/parent.onnx")

# Ausgabe der einzelnen Netzwerk-Nodes
for node in model.graph.node:
    print(node)
```

```
input: "global_in"
input: "MatMul_0_param0"
output: "MatMul_0_out0"
name: "MatMul_0"
op_type: "MatMul"
domain: ""
```

```
input: "MatMul_0_out0"
input: "Mul_0_param0"
output: "Mul_0_out0"
name: "Mul_0"
op_type: "Mul"

input: "Mul_0_out0"
input: "MultiThreshold_0_param0"
output: "MultiThreshold_0_out0"
name: "MultiThreshold_0"
op_type: "MultiThreshold"
attribute {
  name: "out_dtype"
  s: "UINT4"
  type: STRING
}
domain: "qonnx.custom_op.general"

input: "MultiThreshold_0_out0"
input: "Mul_1_param0"
output: "Mul_1_out0"
name: "Mul_1"
op_type: "Mul"

input: "Mul_1_out0"
input: "MultiThreshold_1_param0"
output: "MultiThreshold_1_out0"
name: "MultiThreshold_1"
op_type: "MultiThreshold"
attribute {
  name: "out_dtype"
  s: "UINT4"
  type: STRING
}
domain: "qonnx.custom_op.general"

input: "MultiThreshold_1_out0"
input: "Mul_2_param0"
output: "Mul_2_out0"
name: "Mul_2"
op_type: "Mul"

input: "Mul_2_out0"
input: "MatMul_1_param0"
output: "MatMul_1_out0"
name: "MatMul_1"
op_type: "MatMul"
```

```
domain: ""

input: "MatMul_1_out0"
input: "Mul_3_param0"
output: "Mul_3_out0"
name: "Mul_3"
op_type: "Mul"

input: "Mul_3_out0"
input: "BatchNormalization_0_param0"
input: "BatchNormalization_0_param1"
input: "BatchNormalization_0_param2"
input: "BatchNormalization_0_param3"
output: "global_out"
name: "BatchNormalization_0"
op_type: "BatchNormalization"
attribute {
  name: "epsilon"
  f: 9.999999747378752e-06
  type: FLOAT
}
attribute {
  name: "momentum"
  f: 0.8999999761581421
  type: FLOAT
}
attribute {
  name: "training_mode"
  i: 0
  type: INT
}
```

[12]:
```
from finn.transformation.fpgadataflow.prepare_ip import PrepareIP

# Vorbereitung des Modells zur Implementierung in einem IP-Block
model = ModelWrapper(builder_dir + "/parent.onnx")
model = model.transform(PrepareIP(fpga_part, target_clk_ns))
model.save(builder_dir + "/model_prepared.onnx")
```

[13]:
```
from finn.transformation.qonnx.convert_qonnx_to_finn import ConvertQONNXtoFINN

# Umwandlung des Modells in ein fuer FINN und die weiteren Schritte notwendiges␣
  ↪Format
model_file = builder_dir + "/model_prepared.onnx"
model = ModelWrapper(model_file)
model = model.transform(ConvertQONNXtoFINN())
```

```
model.save(builder_dir + "/model_file_ready.onnx")
```

1.6 Erstellung des IP-Blocks

[14]:
```
import finn.builder.build_dataflow as build
import finn.builder.build_dataflow_config as build_cfg
import os
import shutil

# Laden des fuer die IP-Block umgewandelten Modells
model_file = builder_dir + "/model_file_ready.onnx"
model = ModelWrapper(model_file)
rtl_output = model_dir + "/rtl_output"

# Loeschen von vorherigen Ausgaben des Build-Prozesses
if os.path.exists(rtl_output):
    shutil.rmtree(rtl_output)
    print("Previous run results deleted!")

# Erstellen der Konfiguration zur Erzeugung des IP-Blocks
cfg_stitched_ip = build.DataflowBuildConfig(
    output_dir          = rtl_output,
    mvau_wwidth_max     = 80,
    target_fps          = 1000000,
    synth_clk_period_ns = target_clk_ns,
    fpga_part           = fpga_part,
    generate_outputs=[
        build_cfg.DataflowOutputType.STITCHED_IP,
    ],
    auto_fifo_depths = False,
    stitched_ip_gen_dcp = True
)
```

```
Previous run results deleted!
```

[15]:
```
%%time
# Starten des Build-Prozesses fuer den IP-Block
build.build_dataflow_cfg(
    model_file,
    cfg_stitched_ip,
)
```

```
Building dataflow accelerator from ./builder_output/model_file_ready.onnx
Intermediate outputs will be generated in /tmp/finn_dev_root
Final outputs will be generated in ./rtl_output
Build log is at ./rtl_output/build_dataflow.log
Running step: step_qonnx_to_finn [1/19]
Running step: step_tidy_up [2/19]
```

```
Running step: step_streamline [3/19]
Running step: step_convert_to_hw [4/19]
Running step: step_create_dataflow_partition [5/19]
Running step: step_specialize_layers [6/19]
Running step: step_target_fps_parallelization [7/19]
Running step: step_apply_folding_config [8/19]
Running step: step_minimize_bit_width [9/19]
Running step: step_generate_estimate_reports [10/19]
Running step: step_hw_codegen [11/19]
Running step: step_hw_ipgen [12/19]
Running step: step_set_fifo_depths [13/19]
Running step: step_create_stitched_ip [14/19]
Running step: step_measure_rtlsim_performance [15/19]
Running step: step_out_of_context_synthesis [16/19]
Running step: step_synthesize_bitfile [17/19]
Running step: step_make_pynq_driver [18/19]
Running step: step_deployment_package [19/19]
Completed successfully
CPU times: user 5.06 s, sys: 1.27 s, total: 6.32 s
Wall time: 1min 11s
```

```
[15]: 0
```

Anhang C

Blockdiagramme

C.1 FFT-Block Blockdiagramm

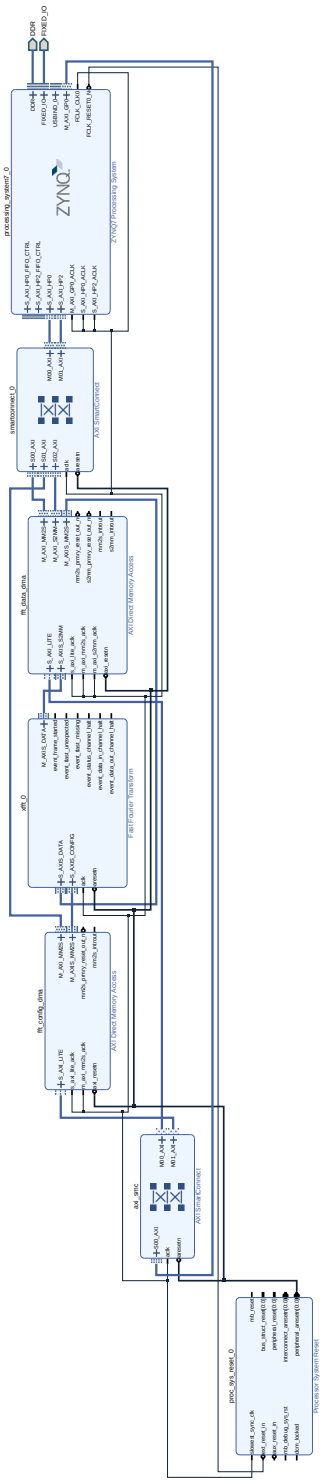


Abbildung C.1: Die Abbildung zeigt den FFT-IP-Block in dem erstellten Overlay um mit diesen über Python die Fast Fourier Transformation von Eingabedaten zu bestimmen.

C.2 Gesamt-Blockdiagramm

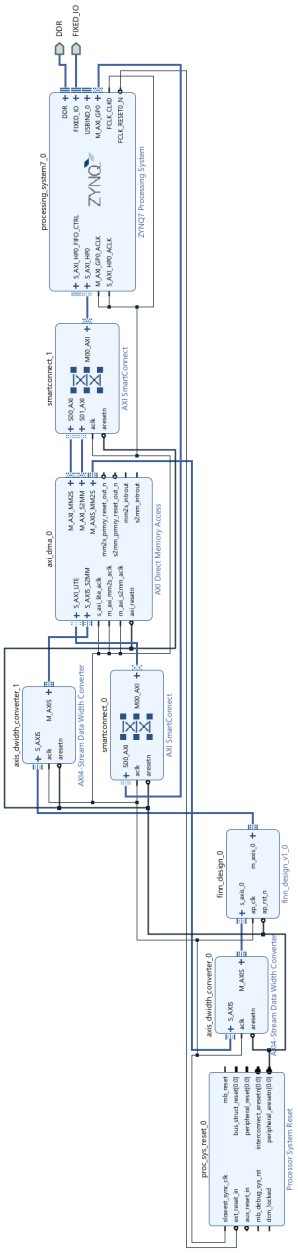


Abbildung C.2: Die Abbildung zeigt das vollständige Blockdiagramm des Gesamtprojekts nach Integration des neuronalen Netzes als IP-Block.