Entwurf neuronaler Netze in Matlab und Designflow zur Implementierung auf Intel SoC-FPGAs mit Zugriff auf die Gewichte in dem externen Linux-SDRAM

Berkay Cakir, Heinz-Peter Bürkle

***Zusammenfassung*—Für Maschinelles Lernen mithilfe von Neuronalen Netzen gibt es in Industrie, Medizin oder im Alltag vielfältige Anwendungen. Oft ist auch die Integration in eingebettete Systeme nötig. Auch wenn nicht das Training des Neuronalen Netzes, sondern lediglich die Inferenz im eingebetteten System erfolgt, stellt dies eine Herausforderung bezüglich der Rechenleistung dar. System-on-Chip (SoC) Field Programmable Gate Arrays (FPGAs) gelten als vielversprechende Zielsysteme für neuronale Netze in eingebetteten Systemen, da sie für die spezielle Aufgabenstellung angepasst werden können. Jedoch erweist sich die komplexe Entwurfsmethodik sowie die begrenzte Größe des On-Chip-Speichers oft als problematisch.**

In dieser Arbeit soll ein Designflow vorgestellt werden, welcher eine unkomplizierte Implementierung eines neuronalen Netzes mit externem Linux-SDRAM Zugriff auf einem Intel SoC-FPGA ermöglicht. Dieser Designflow wird beispielhaft anhand dem DE10-Nano als Zielsystem durchgeführt. Das neuronale Netz wird mithilfe von Matlab erstellt und in VHDL-Code umgewandelt. Danach wird dieser Code in ein SoC-FPGA Design mit SDRAM-Transfer eingebunden. Zuletzt erfolgt die Allokation sowie das Beschreiben des Linux-SDRAM-Buffers mit den Gewichten.

***Schlüsselwörter*—SoC, FPGA, neuronale Netze, Matlab, Linux SDRAM**

I. Einleitung

Durch die rasante Entwicklung von Soft- und Hardware hat der Anwendungsbereich von neuronalen Netzen (NN) stark zugenommen. Diese eignen sich insbesondere für die Sprach- sowie Objekterkennung, Optimierungsaufgaben, Klassifikationen und vielen weiteren komplexen Aufgaben[1]. Dabei kommen viele dieser Anwendungsbereiche nicht nur in der Industrie oder Medizintechnik zum Einsatz, sondern mittlerweile auch verstärkt im Alltag.

Neuronale Netze benötigen eine sehr hohe Anzahl von Rechen- sowie Speicherzugriffe. Daher erfolgt deren Implementierung häufig in Graphics Processing Units (GPUs). Eine interessante Alternative hierzu sind FPGAs. Ausschlaggebend dafür sind folgende Gründe:

- Hohe Rechenleistung: Neuere Generationen von FPGAs bieten Rechenleistungen, welche in bestimmten Anwendungsfällen GPUs übertreffen [2].
- Energieeffizienz: GPU-basierte Implementierungen haben einen sehr hohen Leistungsverbrauch. High End GPUs wie die *Nvidia Titan rtx*, welche oftmals bei komplexeren neuronalen Netzen verwendet werden, haben eine Verlustleistung von 280 Watt [3]. Im Vergleich dazu hat ein typisches FPGA mit z.B. einen Chip der *Arria 10* Reihe einen Leistungsverbrauch von nur 70 Watt.
- Rekonfiguration: FPGAs sind im Unterschied zu GPUs noch feingranularer und können bei entsprechender Konfiguration ein größeres Maß an Parallelisierung und Pipelining erzielen. Auch kann die Bitbreite genau an die Aufgabenstellung angepasst werden [4].
- Kosten- und Platzersparnis: GPUs benötigen zusätzlich einen Hostcomputer, welcher weitere Kosten verursacht und einen großen Platz einnimmt. Für kleinere eingebettete Systeme sind handelsübliche GPUs in der Regel nicht geeignet. Jedoch bieten Embedded Systems wie z. B. der Jetson Nano eine sinnvolle Alternative.

Die Nutzung von SoC-FPGAs bringt jedoch zwei wesentliche Nachteile mit sich. Der eine Nachteil der FPGAs ist die Speicherproblematik. Denn für viele Anwendungsfälle reicht der On-Chip-Speicher, welcher bei preiswerten Modellen wie z. B. dem DE10-Nano 3 Mbit betragen, nicht aus. Daher muss der Off-Chip-Speicher genutzt werden. Dieser ist bei FPGAs im Gegensatz zu anderen Hardwareplattformen ein limitierender Faktor, wenn eine hohe Performance erzielt werden soll. So kann die geringe Bandbreite des Off-Chip-Speichers zu einem kritischen Bottleneck führen und somit stark die Leistung senken [5]. Daher muss ein wohldurchdachtes Verfahren für den Speichertransfer entwickelt werden. Ein weiterer Nachteil von SoC-FPGAs ist der erforderliche Hardwareentwurf in einer HDL (Hardware Descrition Language). Diese Entwurfsmethodik ist für Softwareentwickler in der Regel nur schwer zugänglich. Die Entwicklung von

Berkay Cakir, Berkay.Cakir@studmail.htw-aalen.de, Heinz-Peter Bürkle, Heinz-Peter.Buerkle@hs-aalen.de.Hochschule Aalen, Beethovenstraße 1, 73430 Aalen .

neuronalen Netzen sowie die Anbindung von Off-Chip-Speichern mit HDL erfordert große Erfahrungen aufseiten des Entwicklers [6].

Um dem Problem der Entwicklungskomplexität entgegenzuwirken, ist ein aufkommender Trend die High Level Sythesis (HLS). Bei dieser werden FPGAs mit herkömmlichen Programmiersprachen wie OpenCL oder C++ programmiert. Für das Implementieren der Funktionen wird aufgrund der höheren Abstraktionsebene kein spezifisches Wissen über VHDL oder Verilog benötigt [7]. Auch Mathworks bietet für Matlab ein HLS Tool an, um Matlab Code in VHDL oder Verilog Code zu transformieren.

Für den Speichertransfer mit dem Off-Chip-Speicher ermöglicht die Verwendung von Linux Device Treibern und IP-Core Komponenten einen effizienten Entwurf.

In dieser Arbeit soll der gesamte Designflow vorgestellt werden, der zur Realisierung eines neuronalen Netzes mit externem Linux-SDRAM Zugriff auf einem Intel SoC-FPGA führt. Dabei soll das neuronale Netz anhand von Matlab erstellt sowie mithilfe des HLS-Tools in VHDL Code transformiert werden. Daraufhin wird dieser Code zur Synthese der programmierbaren Logik in ein SoC-FPGA Design mit DMA-Transfer auf das Linux-SDRAM eingebunden. Zum Schluss erfolgt die Allokation sowie das Beschreiben des externen Buffers mit den Gewichten mithilfe des Linux Device Treibers *u-dma-buf*.

Im Folgenden werden zuerst die Grundlagen von SoC-FPGAs, neuronalen Netzen, DMA und u-dma-buf erklärt. Im 3. Kapitel geht es um die Erzeugung des neuronalen Netzes als VHDL-Code mithilfe von Matlab. Kapitel 4 erläutert die Einbindung des NNs in ein SoC-FPGA-Design mit SDRAM-Transfer. In Kapitel 5 wird auf die Allokation sowie das Beschreiben der Buffer im Linux-SDRAM eingegangen. Zum Schluss folgt ein kurzes Fazit.

II. Grundlagen

In diesem Kapitel sollen die relevanten Grundlagen dargestellt werden.

A. Neuronale Netze

Aufbau und Funktionsweise von künstlichen neuronalen Netzen leiten sich aus der Vernetzung von Neuronen im Nervensystem von Lebewesen ab. Die NNs bestehen aus mehreren Schichten (Layern). In diesen sind Knotenpunkte enthalten, welche auch Neuronen genannt werden. Die Anzahl dieser kann sich je nach Layer unterscheiden. Die Neuronen der Schichten sind über Gewichtungen miteinander verbunden. Diese bestimmen über die Stärke der Verbindungen zwischen den Neuronen. Während des Trainings werden diese Gewichtungen angepasst. NNs werden prinzipiell in drei Arten von Layern unterteilt: Input Layer, Hidden Layer(s) und Output Layer. Der Input Layer bildet die erste Schicht und enthält die Eingabewerte, welche an die nächste Schicht der Hidden Layer weitergegeben wird. Diese befindet sich zwischen Input und Output Layer. Sind mehrere Hidden Layers enthalten, bezeichnet man das System auch als Deep Learning. In diesen Layern werden die empfangenen Informationen neu gewichtet und zur nächsten Schicht weitergegeben. Der Output Layer ist die letzte Schicht und beinhaltet die Resultate.

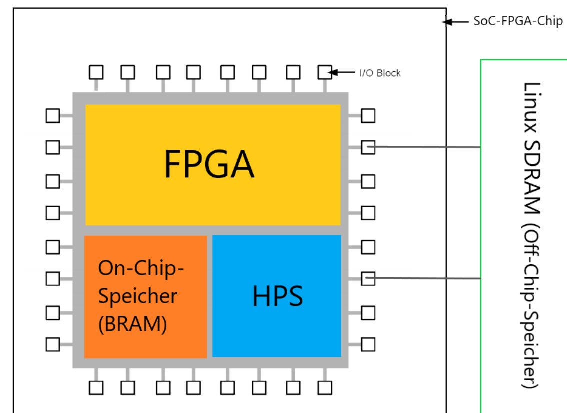

Abbildung 1. Vereinfachter Aufbau SoC-FPGA

B. SoC-FPGAs

SoC-FPGAs vereinen einen Prozessor und eine FPGA-Fabric auf nur einem Chip. Dadurch bieten diese eine höhere Integration, geringeren Leistungsverbrauch, kleinere Boardgröße und eine Kommunikation mit höherer Bandbreite zwischen Prozessor und FPGA [8].

Der Prozessor, auch Hard-Processor-System (HPS) genannt, ist oftmals ein ARM Prozessor mit einem Linux-Betriebssystem. Dieser ist mit dem FPGA über eine AXI Brücke verbunden. Als RAM nutzt das HPS ein SDRAM. Dieses befindet sich nicht auf dem SoC-FPGA-Chip (Off-Chip-Speicher) und hat somit eine höhere Latenz. Jedoch ist SDRAM kostengünstig und wird daher oft bei größeren Datenmengen genutzt.

Gewichte in Neuronalen Netzen lassen sich aufgrund ihres hohen Speicherbedarfs bei größeren Netzen am Besten im SDRAM ablegen. Diese Gewichte lassen sich nun stückweise vom SDRAM auf den On-Chip-Speicher des FPGAs kopieren. Der On-Chip-Speicher wird genutzt, um beim Betrieb des NN die Latenz zu verringern sowie ein Bottleneck zu vermeiden. Als On-Chip-Speicher wird oft BRAM (Block-RAM) verwendet. Abbildung 1 stellt den vereinfachten Aufbau eines SoC-FPGAs mit Speicherzugriff dar.

C. Direct Memory Acess (DMA)

Für das Kopieren der Daten aus dem SDRAM in den On-Chip-Speicher wird DMA genutzt. DMA ist eine Funktion in Computersystemen, die es bestimmten

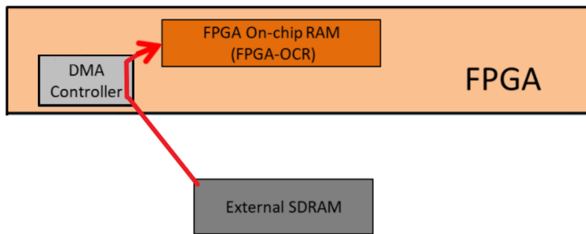

Abbildung 2. DMA-Transfer

Hardware Subsystemen ermöglicht, unabhängig von der CPU auf den Hauptspeicher des Systems (Direktzugriffsspeicher) zuzugreifen. Beim DMA-Transfer werden Daten von einem Adressraum in einen anderen Adressraum kopiert. Ein typisches Beispiel ist das Verschieben eines Blocks aus dem externen Speicher in einen schnelleren Speicherbereich innerhalb eines Systems. Ohne DMA ist die CPU bei einer programmierten Eingabe/Ausgabe in der Regel für die gesamte Dauer des Lese oder Schreibvorgangs voll ausgelastet und steht somit für andere Arbeiten nicht zur Verfügung. Mit DMA leitet die CPU zunächst die Übertragung ein, führt dann andere Operationen durch, während die Übertragung läuft, und erhält schließlich eine Unterbrechung vom Direct Memory Access Controller (DMAC), wenn der Vorgang abgeschlossen ist. Diese Funktion ist immer dann nützlich, wenn die CPU mit der Datenübertragungsrate nicht mithalten kann oder wenn die CPU eine Arbeit ausführen muss [9]. Zusammengefasst ist der Vorteil des DMAs eine schnellere Datenübertragung bei gleichzeitiger Entlastung des Prozessors. Bei SoC-FPGA lassen sich zwei unterschiedliche DMACs nutzen. Einerseits besitzt oftmals das HPS einen DMAC, jedoch kann auch ein DMAC auf Seiten des FPGAs mithilfe einer IP-Core-Komponente angepasst und verwendet werden. In dieser Arbeit wird ein DMAC im FPGA-Design genutzt (wie in Abbildung 2), da dieser eine höhere Datenübertragungsrate bietet als im HPS (ca. Faktor 10).

D. *u-dma-buf (User space mappable DMA Buffer)*

Die Allokation des Off-Chip-Speichers ist der nächste wichtige Entwicklungsschritt. Dabei muss berücksichtigt werden, dass das Betriebssystem auf dem HPS den Speicher verwaltet. Dieser Speicher wird in zwei verschiedene Bereichen unterteilt: User Space und Kernel Space. Prozesse im User Space haben nur Zugriff auf einen begrenzten Teil des Speichers, während der Kernel Zugriff auf den gesamten Speicher hat. Prozesse im User Space haben zudem auch keinen Zugriff auf den Kernel Space. User Space Prozesse können nur auf einen kleinen Teil des Kernels zugreifen, und zwar über eine Schnittstelle, die der Kernel zur Verfügung stellt, nämlich die Systemaufrufe. Aufgrund des eingeschränkten Zugriffs auf den Speicher im User Space

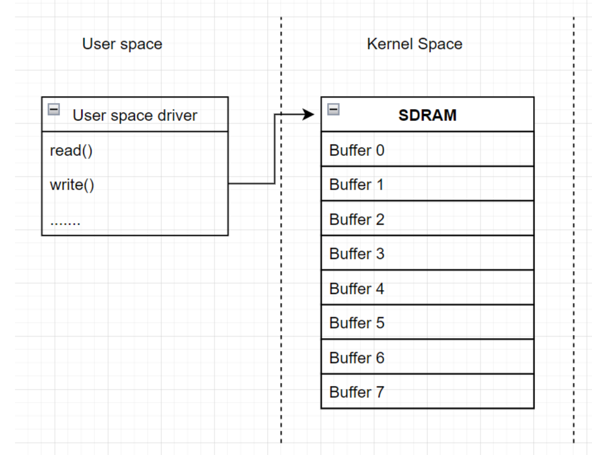

Abbildung 3. Funktionsweise u-dma-buf

und eines nur teilweisen Zugriffs mithilfe von Systemaufrufen erweist sich das Allokieren von Buffern im SDRAM als Äußerst komplex [10]. Jedoch existiert ein Linux Device Treiber mit der Bezeichnung u-dma-buf [11], welcher dies ermöglicht. Mit diesem lassen sich bis zu acht DMA-Buffer im Linux SDRAM (Kernel Space) anlegen, welche vom User space aus verwendet werden können (siehe Abbildung 3).

III. Erzeugung des NNs als VHDL-Code mithilfe von Matlab

In diesem Kapitel soll der Matlab-Designflow zur Entwicklung des VHDL-Codes für ein neuronales Netz beschrieben werden.

A. *Erstellung eines NNs in Matlab*

Zur Entwicklung neuronaler Netze bietet Matlab die *Deep learning Toolbox* an. Mit dieser lassen sich einfache NNs mit einigen wenigen Layern bis hin zu komplexeren tiefen neuronalen Netzen mit hunderten Schichten erstellen.

Da der Designflow unabhängig von der Größe ist, soll hier ein einfaches Feedforward NN mit zwei Layern genügen. Als Anwendungsfall wird ein Klassifizierungsproblem angenommen werden. Für solche neuronale Netze bietet die Toolbox die *Neural Pattern Recognition* App an.

In der Anwendung müssen zunächst Eingabewerte und deren Ausgabewert für das Training des NN geladen werden. In den folgenden Schritten muss die Verteilung der Daten für das Training, die Validation und das Testen sowie die Anzahl der Neuronen in den Hidden Layer festgelegt werden. Nun kann das neuronale Netz trainiert werden und im Anschluss werden die Resultate ausgegeben. Wenn diese im Rahmen der Anforderungen liegen, kann das NN als Matlab-Code angezeigt und gespeichert werden.

Nachdem das trainierte neuronale Netz erfolgreich als Matlab-Code implementiert wurde, muss dieses

noch entsprechend angepasst und erweitert werden, was für die Konvertierung in Festkommadarstellung und in VHDL erforderlich ist.

B. Anpassung und Erweiterung des NNs

Zuerst soll mit der Erweiterung begonnen werden. Dazu wird eine Top-Layer Funktion erstellt. Diese ist erforderlich, da der HDL-Coder keine Vektoren als direkte Eingabewerte akzeptiert. Deshalb muss eine Funktion implementiert werden, welche die Attribute als Eingabeparameter erhält und diese innerhalb der Funktion in einen Spaltenvektor umwandelt sowie mit dieser die NN-Funktion aufruft. Außerdem soll die Top-Layer-Funktion die Kategorie mit der höchsten Wahrscheinlichkeit ermitteln und diese zurückgeben.

Nachdem die Top-Layer Funktion implementiert wurde, kann der Code innerhalb der NN-Funktion angepasst werden. Zunächst werden unnötige Programmausschnitte entfernt.

Betrachtet man die Unterfunktion *softmax_apply*, ist zu erkennen, dass diese erst prüft, ob die Matrizen auf der GPU berechnet werden. Ist dies der Fall, wird die Funktion *iSoftmaxApplyGPU* aufgerufen. Anderenfalls wird davon ausgegangen, dass die Daten in einer CPU berechnet werden und *iSoftmaxApplyCPU* wird ausgeführt. Die beiden Funktionen verarbeiten die Daten auf unterschiedliche Art weiter, jedoch ergeben sich identische Endergebnisse. Da in diesem Beispiel ein FPGA verwendet wird und somit auch keine Unterscheidung erforderlich ist, ob die Matrizen weder in der CPU oder in der GPU berechnet werden, soll nur eine der beiden möglichen *iSoftmaxApply* Funktionen für die Weiterverarbeitung der Matrizen erhalten bleiben.

Im nächsten Schritt werden unnötige Divisionen entfernt. Diese sollen weitestgehend vermieden werden, da nach der Festkomma Konvertierung die Divisionen mithilfe der Funktion *fi_div* ausgeführt werden. Diese beinhaltet dynamische Matrizen und ist somit stark fehleranfällig bei der VHDL-Umwandlung mit dem HDL-Coder Tool. Oftmals werden die Divisionen für die Berechnung der Softmax gebildet und können entweder vernachlässigt oder im HPS berechnet werden. Falls Divisionen unbedingt benötigt werden, müssen HDL-konforme Workarounds angewendet werden.

Danach werden Exponentialfunktionen, welche ganze Vektoren exponentieren, so umgeschrieben, dass diese jeweils nur noch skalare Größen enthalten. Erst danach sollen die Ergebnisse in Vektoren geschrieben werden. Dieser Schritt wird angewandt, da bei der Festkomma Konvertierung keine Exponentialfunktionen mit Vektoren in Lookup Tables (LUT) umgewandelt werden können.

Zum Schluss werden nicht unterstützte Funktionen ersetzt. Ein häufig vorkommendes Beispiel wären Funktionen des Typs *bsxfun*. Diese führen bestimmte elementweise Operationen an zwei Matrizen durch, was vom Festkomma Konverter nicht unterstützt wird. Daher müssen nun die elementweisen Operationen manuell umgeschrieben werden.

Zusammengefasst sind folgende Schritte notwendig, um das NN-Modell als Matlab-Code für die Konvertierungen anzupassen:

- Erstellung einer Top-Layer Funktion
- Entfernung von nicht benutzten Code-Abschnitten
- Entfernung von nicht essenziellen Divisionen
- Umschreibung von Exponentialfunktionen mit Vektoren zu skalaren Größen
- Umschreibung von nicht unterstützten Funktionen wie z. B. bsxfun

Diese Schritte können je nach NN-Modell variieren oder abweichen. Dennoch bietet die Liste einen groben Überblick worauf zu achten ist, wenn ein NN konvertiert werden soll.

C. Erzeugung von VHDL-Code

Die Erzeugung des VHDL-Codes findet mithilfe des Matlab Tools *HDL-Coder* statt. Es wird zuerst eine Festkomma Konvertierung vorgenommen. Die Festkommadarstellung hat den großen Vorteil, dass mit dieser eine höhere Leistung erzielt werden kann. Der Nachteil jedoch ist, dass der Wertebereich begrenzt dargestellt wird. Außerdem wird die Festkommadarstellung für die Implementierung auf dem FPGA benötigt.

Im *Fixed Point Converter* Tool müssen zuerst die Exponentialfunktionen zu LUTs umgewandelt werden. Im nächsten Schritt wird der Matlab-Code analysiert und die vorgeschlagene Wort- sowie Bruchlänge kann angepasst werden. Danach kann der Prozess gestartet werden. Im Anschluss wird eine Übersicht mit der Abweichung des festkommakonvertierten Modells zum vorherigen Modell angezeigt.

Eine Abweichung zwischen den Modellen kann auftreten. Diese mögliche Differenz ist vor allem den Exponentialfunktionen, welche bei der Festkommakonvertierung zu LUTs umgewandelt werden, zuzurechnen. Bei einem NN Modell mit vielen Neuronen könnte die Modellgenauigkeit zwar besser ausfallen, jedoch würde dieses im Gegenzug mehr Exponentialfunktionen besitzen, welche zu LUTs umgewandelt werden müssten und die Abweichungen im Festkomma System wären daher umso größer. Daher ist es wichtig, diese Differenz zu überprüfen sowie den Tradeoff Effekt zwischen höherer Modellgenauigkeit und Abweichung in Festkommadarstellung bei der Parametrisierung der Neuronen zu berücksichtigen.

Nun kann mit der Erzeugung des VHDL-Codes begonnen werden. Dazu muss im nächsten Schritt das Zielgerät sowie das Synthesetool ausgewählt werden. Danach kann die Konvertierung gestartet werden und die benötigten VHDL-Dateien werden generiert.

IV. Einbindung des NNs in ein SoC-FPGA Design mit SDRAM-Transfer

In diesem Kapitel soll der Workflow für ein FPGA Design mit SDRAM Kommunikation, DMAC und On-Chip-Buffer beschrieben werden. Dazu wird zuerst auf den Aufbau im *Quartus Platform Designer*, gefolgt von einem möglichen Zustandsautomaten als Ablaufsteuerung zur Berechnung des NNs eingegangen.

A. SoC-FPGA Architektur mit SDRAM-Transfer

Für die Einbindung der Datenübertragung wird der Quartus *Platform Designer* genutzt. Mit diesem lässt sich leicht ein FPGA-Design für die SDRAM Nutzung erstellen. Das Design besteht aus folgenden fünf Komponenten:

- HPS (hps_0)
- On-Chip-Buffer (onchip_memory2_0)
- Clock mit 50 MHz (clk_0)
- PLL (pll_0)
- DMAC (dma_0)

Im ersten Schritt müssen alle diese Komponenten in das Design eingefügt werden. Danach werden diese wie folgt konfiguriert. Abbildung 4 zeigt die Verbindungen der Komponenten.

Zuerst wird die Clock Source auf 50 MHz gesetzt und mit der PLL verbunden. Bei dieser wird die Ausgangsfrequenz auf 100 MHz eingestellt. Der Vorteil der höheren Taktfrequenz ist, dass Lese und Schreibzugriffe, welche zwei Takte benötigen, innerhalb eines Taktes der Grundfrequenz von 50 MHz abgeschlossen sind.

Anschließend wird das *FPGA-to-HPS SDRAM* Interface im HPS aktiviert. Dazu wird ein Port mit dem Namen *f2h_sdram0* mit der Typenbezeichnung *Avalon-MM Bidirectional* hinzugefügt. Dieses Interface wird genutzt, um auf die Daten im SDRAM zuzugreifen. Nun müssen noch die verschiedenen Clocks im HPS mit der PLL verbunden werden. Beim DMAC müssen keine besonderen Konfigurationen gesetzt werden. Als Clock wird der PLL verwendet. Der *control_port_slave* ist für die Steuerung der DMAC verantwortlich. Dieser Port wird exportiert, um die Steuerung im FPGA durchzuführen. Der *read_master* liest die Daten aus und wird daher mit dem SDRAM Interface verbunden. Zum Schluss wird der *write_master* mit dem On-Chip-Buffer verknüpft, da in diesem die Daten zwischengespeichert sowie von diesem aus verarbeitet werden sollen. Zudem wird für den On-Chip-Speicher noch die Option *Dual-Port-Access* aktiviert, da zwei Slaves benötigt werden: Ein Port ist für das Speichern der Daten vom DMAC und ein weiterer exportierter Port ist für das Auslesen des Buffers in der programmierbaren FPGA-Logik vorgesehen.

Abschließend wird die Clock des On-Chip-Speichers mit der PLL verknüpft. Optional kann der Slave-port für das Beschreiben des Buffers mit der AXI Brücke

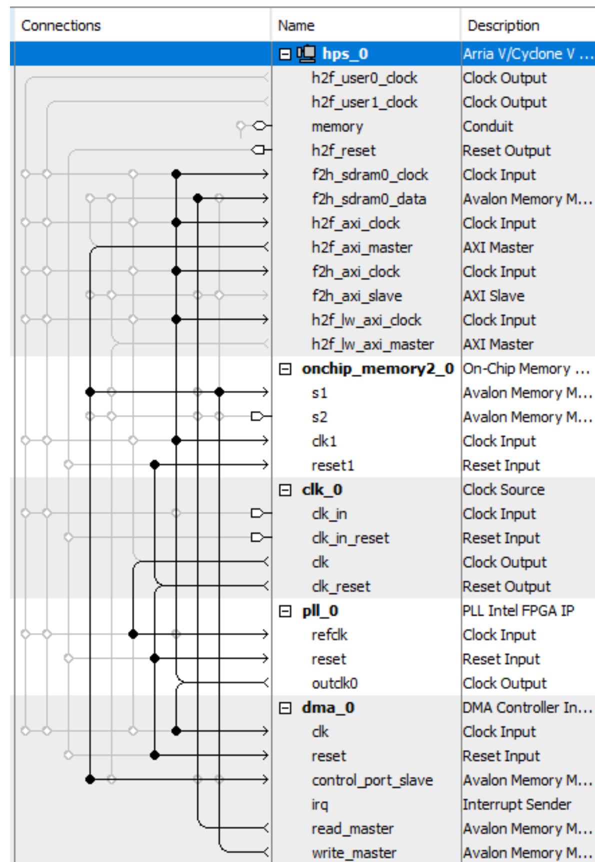

Abbildung 4. SoC-FPGA-Design im Platform Designer

verbunden werden, um diesen im Anwendungsprogramm auszulesen oder zu debuggen.

B. Implementierung des Zustandsautomaten

Im Folgenden soll ein möglicher Zustandsautomat vorgestellt werden, welcher die Kommunikation mit dem SDRAM für den FPGA steuert und den Algorithmus des NNs ausführt. Der Zustandsautomat ist von vielen verschiedenen Faktoren abhängig wie z. B. der Größe des NNs. Bei tiefen neuronalen Netzen müssen die einzelnen Teilnetze partiell berechnet werden, da aufgrund der Größe nicht die gesamten Gewichte und LUTs im On-Chip-Buffer gespeichert werden können. Ein beispielhafter Automat kann wie in Abbildung 5 aussehen.

Im Rahmen der vorliegenden Arbeit lässt sich das NN mit nur zwei Layern innerhalb eines Zustandes als Ganzes ausführen und wurde einfachheitshalber auch mit einem solchen Zustandsautomaten implementiert.

Bei der Implementierung der Automaten muss unbedingt die richtige Taktfrequenz für die Zustände ermittelt und ggf. angepasst werden.

V. Reservierung von Buffern im Linux SDRAM mit u-dma-buf

In diesem Abschnitt wird die Implementierung des Linux Device Treibers *u-dma-buf* beschrieben. Dazu

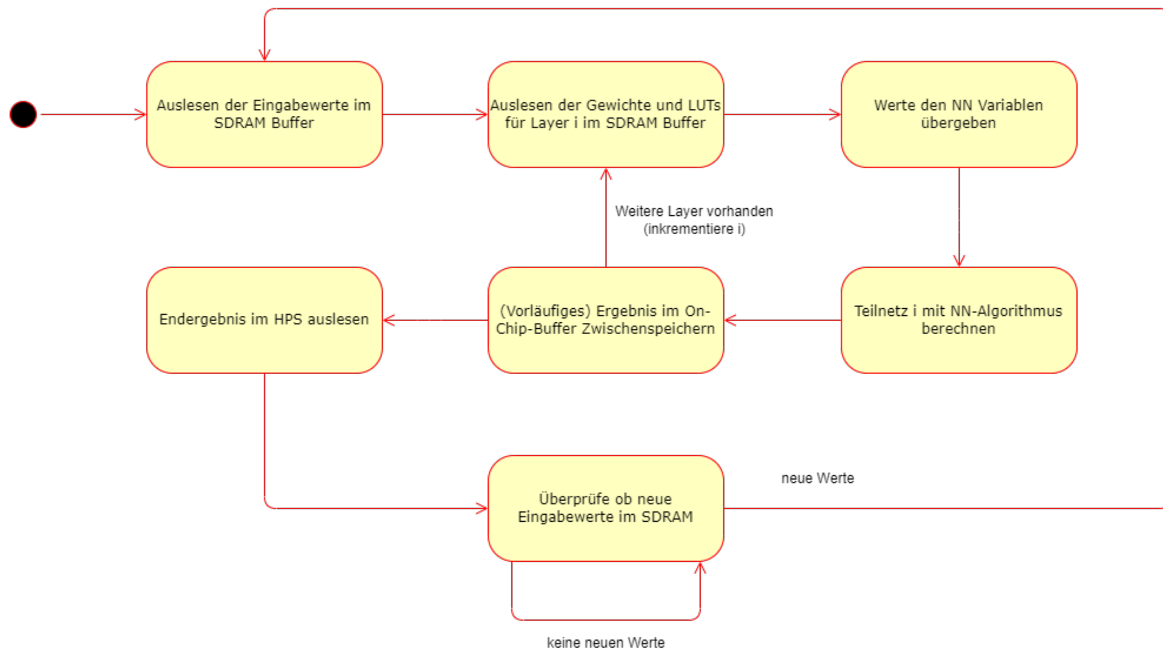

Abbildung 5. FPGA Zustandsautomat für größere neuronale Netze mit partieller Berechnung von Teilnetzen auf dem FPGA

wird zuerst dargestellt, wie durch diesen Treiber der Off-Chip-Speicherplatz allokiert wird. Danach folgt eine Erläuterung, wie dieser Buffer mithilfe eines Anwendungsprogramms im HPS beschrieben wird.

A. *Allokation der Buffer mithilfe des Linux Device Treibers*

Für die Implementierung des *u-dma-buf* Treibers wird für das Linux Betriebssystem im HPS bestimmte Kernelversionen vorausgesetzt (3.18, 4.4, 4.8, 4.12, 4.14, 4.19, 5.4). Falls keines dieser Versionen vorhanden ist, muss zuerst eine passende Kernelversion kompiliert werden.

Nachdem das Kernelimage erfolgreich kompiliert wurde, kann nun der Device Treiber für *u-dma-buf* erstellt werden.

Mit diesem Treiber kann im HPS des SoC-FPGAs nun SDRAM-Speicher allokiert werden. Dazu gibt es es drei verschiedene Möglichkeiten.

Die erste Methode ist die Konfiguration mithilfe des Device Tree Files. Mit dem Device Tree können Treiber bereits beim Booten des HPS geladen werden. Dabei können im Device Tree die Anzahl, die Größe und sogar die Adressen der Buffer und vieles mehr definiert werden.

Die naheliegende Möglichkeit ist es, den Speicher mit dem *u-dma-buf* Manager zu allokieren. Dabei kann der Treiber mit dem Anwendungsprogramm geladen und dann der Speicher für das NN erstellt werden.

Die nachfolgende Methode ist die schnellste und einfachste Möglichkeit: Der Device Treiber erstellt mithilfe des *insmod* Befehls in der Linux-Konsole die Buffer.

B. *HPS Software für das Befüllen der Buffer*

Zuletzt soll die Software für den HPS erstellt werden. Diese soll den Buffer im SDRAM beschreiben sowie den On-Chip-Buffer des FPGAs auslesen.

Zuerst werden die virtuellen Adressen für den Zugriff des HPS auf die programmierbare Logik des FPGA mithilfe von mmap generiert. Es wird ein Adressraum von 1 GB abgebildet, der den Bereich von 0xC0000000 bis 0xFFFFFFFF abdeckt. Durch dieses Mapping können nun die einzelnen virtuellen Adressen der betreffenden Module auf dem FPGA berechnet werden.

Im nächsten Schritt müssen im FPGA Register *FPGA2DRAMC* bestimmte Konfigurationsbits auf 1 gesetzt werden. Dies ist notwendig, damit sich die FPGA-to-SDRAM Ports nicht zurücksetzen. Standardmäßig befinden sich diese in einem Reset-Zustand und können dann nicht genutzt werden.

Nun wird in der Software auf den Linux Device Treiber zugegriffen und folgende Aktionen ausgeführt:

Der CPU-Cache wird zuerst deaktiviert. Dies ist notwendig, da die Daten zwischen dem Prozessor und dem DMAC nicht kohärent sind.

Danach dürfen die allokierten SDRAM-Buffer beschrieben werden. Dies geschieht durch eine einfache *read* Anweisung.

Nachdem der externe Linux-SDRAM allokiert und beschrieben wurde, kann das FPGA-Design mit dem NN in Betrieb genommen werden.

VI. FAZIT

Im Rahmen dieser Arbeit wurde erfolgreich ein Designflow geschaffen und implementiert, mit wel-

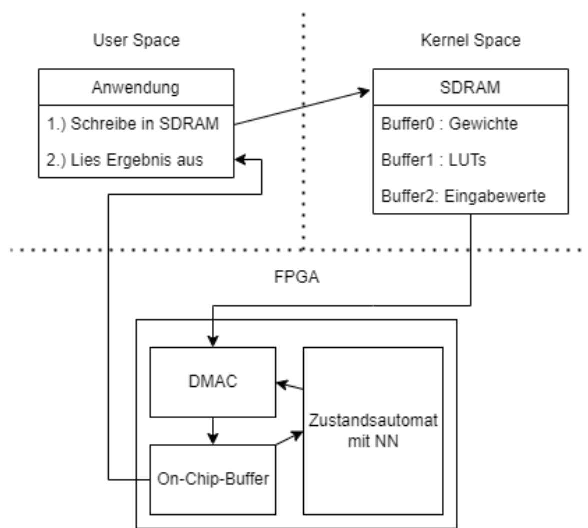

Abbildung 6. Aufbau und Funktion des Gesamtsystems

chem ein in Matlab erstelltes NN auf einem Intel SoC-FPGA mit Zugriff auf Linux-SDRAM entwickelt werden kann.

Dabei wurde aufgezeigt, wie ein solches NN-Modell mithilfe von Matlab erstellt und angepasst werden kann. Ein FPGA Design, welches mit einem DMA-Transfer den Inhalt des Linux-SDRAMs in den On-Chip-Speicher des FPGAs kopiert, wurde erstellt. Durch den Linux Device Treiber u-dma-buf können bis zu acht Buffer im SDRAM angelegt und beschrieben werden. Abbildung 6 zeigt den Aufbau und die Funktionsweise des Gesamtsystems.

Dieser Designflow ermöglicht eine unkomplizierte Implementierung eines NNs. Mit der Nutzung eines Off-Chip-Buffers wird der limitierende Faktor des geringen On-Chip-Buffers für das Speichern der Gewichte von tiefen neuronalen Netzen hinfällig.

Der Workflow und das System wurden auf dem DE10-Nano mit einem Feedforward NN mit zwei Layern getestet, lässt sich jedoch ohne Probleme auf weitere *Cyclone 5* SoC-FPGA Boards und größere neuronale Netze übertragen.

Literaturverzeichnis

[1] Michaela Tiedemann: *KI, künstliche neuronale Netze, Machine Learning, Deep Learning: Wir bringen Licht in die Begriffe rund um das Thema „Künstliche Intelligenz“* URL: https://tinyurl.com/2whbwt2v - Letzter Zugriff 05.01.2021.

[2] Eriko Nurvitadhi u. a. : *"Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?“ In: Proceedings of the 2017 ACM/SIGDA Interna- tional Symposium on Field-Programmable Gate Arrays. FPGA ’17. Monterey, California, USA: Association for Computing Machinery, 2017, S. 5–14. ISBN: 9781450343541.* URL: https://doi.org/10.1145/3020078.3021740 - Letzter Zugriff 05.01.2021.

[3] Nvidia *Titan rtx Specifications* URL: https://www.nvidia.com/content/dam/en-zz/Solutions/titan/documents/titan-rtx-for-creators-us-nvidia-1011126-r6-web.pdf - Letzter Zugriff 04.01.2021

[4] Intel *FPGA vs. GPU for Deep Learning* URL: https://tinyurl.com/28atztht - Letzter Zugriff 04.01.2021

[5] Xuechao Wei, Yun Liang und Jason Cong: *„Overcoming Data Transfer Bottlen- ecks in FPGA-based DNN Accelerators via Layer Conscious Memory Manage- ment“. In: 2019 56th ACM/IEEE Design Automation Conference (DAC). 2019, S. 1–6.* - Letzter Zugriff 22.02.2022

[6] Atze van der Ploeg: *Why use an FPGA instead of a CPU or GPU? 2018.* URL: https://blog.esciencecenter.nl/why-use-an-fpga-instead-ofa-cpu-or-gpu-b234cd4f309c - Letzter Zugriff 04.01.2021

[7] Margit Kuther Andreas Widder: *OpenCL – neuer Ansatz für die Programmierung von SoC-FPGAs. 2016.* URL:https://www.embedded-software-engineering.de/opencl-neuer-ansatz-fuer-die-programmierung-von-soc-fpgas/ - Letzter Zugriff 04.01.2021

[8] Intel. *What is an SoC FPGA?* URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ab/ab1_soc_fpga.pdf - Letzter Zugriff 05.01.2021

[9] FPGAKey: *DMA Dircet Memory Access.* URL: https://www.fpgakey.com/wiki/details/308 Letzter Zugriff 20.02.2022.

[10] User Space vs. Kernel Space. URL: https://unix.stackexchange.com/questions/87625/what-is-difference-between-user-space-andkernel-space - Letzter Zugriff 20.02.2022.

[11] Github: *u-dma-buf.* URL: https://github.com/ikwzm/udmabuf - Letzter Zugriff 20.02.2022.

Berkay Cakir Berkay Cakir erhielt den akademischen Grad des Bachelor of Engineering in Elektrotechnik im Jahr 2021 von der Hochschule Aalen. Dort studiert er seit 2021 den Master Machine Learning and Data Analytics. Aktuell arbeitet er an seiner Masterarbeit im Bereich Predictive Maintenance.

Heinz-Peter Bürkle Heinz-Peter Bürkle bekam den akademischen Grad Dipl.-Ing. in Elektrotechnik im Jahr 1991 von der Universität Stuttgart und den Grad Dr.-Ing. 1997 ebenfalls von der Universität Stuttgart. Nach einigen Jahren in Forschung und Entwicklung bei Alcatel ist er seit 2003 Professor für Schaltungstechnik und rechnergestützten Schaltkreisentwurf an der Hochschule Aalen. Als Prorektor Digitalisierung treibt er zudem diverse Projekte im Feld der Künstlichen Intelligenz voran.