technische universität
dortmund

Bachelorarbeit

DeepRacin auf FPGAs - Ein Framework zur Inferenz von DeepLearning Modellen auf FPGAs

Andreas Bühner

Gutachter:
Prof. Dr. Katharina Morik
Sebastian Buschjäger

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Künstliche Intelligenz (LS-8)
http://www-ai.cs.uni-dortmund.de

In Kooperation mit:
Lehrstuhl für Graphische Systeme (LS-7)
http://ls7-www.cs.uni-dortmund.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

Tiefe neuronale Netze haben eine große Verbreitung in Anwendungsbereichen wie der Bilderkennung, Websuche oder Spracherkennung gefunden [22, Seite 436]. Die Idee künstlicher neuronaler Netze besteht bereits seit den 1940er Jahren und wurde besonders in den 80er und 90er Jahren weiterentwickelt [25, Seite 13]. Zum alltäglichen Einsatz kommen künstliche neuronale Netze erst seit einigen Jahren, da eine hohe Rechenleistung für das Training notwendig ist, welche zum Beispiel durch *GPUs* (Graphics Processing Units) bereitgestellt werden kann [30, Seite 8], [22, Seite 439]. Heute finden neuronale Netzte sogar auf mobilen Geräten mit stark begrenzten Energie- und Rechenressourcen wie Smartphones Verwendung. Unter Anderem bei der Gesichtserkennung mittels FaceID von Apple [2]. Viele Anwendungen, welche mit neuronalen Netzen umgesetzt werden können, wie echtzeit Objekterkennung sind für den Einsatz auf eingebetteten Systemen, beispielsweise in autonomen Fahrzeugen oder Robotern interessant [29]. Besonders rechenintensiv ist das Training von neuronalen Netzen, während die Inferenz, also die Vorhersage mittels eines trainierten Netzes, weniger Rechenzeit in Anspruch nimmt. Das Framework *DeepRacin* (Deep Resource-Aware OpenCL Inference Networks) [23] wurde entwickelt, um die Inferenz eines bereits trainierten neuronalen Netzes auf ressourcenarmer Hardware auszuführen. Dadurch ist es möglich das Training auf leistungsstarker Serverhardware mit Frameworks wie TensorFlow [11] durchzuführen und anschließend DeepRacin für die Inferenz auf eingebetteten Systemen zu verwenden. Um DeepRacin auf heterogener Hardware ausführen zu können, wurde es mit *OpenCL* (Open Computation Language) implementiert. DeepRacin wurde für die Verwendung mit GPUs optimiert. Es bietet allerdings, aufgrund der Implementierung mit OpenCL, die Möglichkeit, die Unterstützung und Optimierung für weitere Hardware zu umfassen.

Eine solche Anpassungen kann für *FPGAs* (Field Programmable Gate Arrays), welche häufig eine energiesparsamere Alternative zu klassischen Prozessoren darstellen, erfolgen. FPGAs bestehen aus einem Gitter von programmierbaren Logikblöcken, deren Konfiguration und Verknüpfung es ermöglicht Funktionen direkt auf der Hardware auszuführen.

Hierdurch kann eine hohe Anzahl von Operationen pro verbrauchtem Joule Energie erreicht werden. Synthesetools von Xilinx [34, Seite 9] oder Intel [14] ermöglichen es, OpenCL Programme für FPGAs zu programmieren. Um dies für DeepRacin durchzuführen, sind einige Änderungen notwendig und verschiedene Optimierungen möglich. Im Zuge dieser Arbeit wurden solche Änderungen am DeepRacin Framework vorgenommen, um damit die Inferenz neuronaler Netze auf FPGAs durchführen zu können.
Im Folgenden werden zuerst die Grundlagen von neuronalen Netzen, spezialisierter Hardware (insbesondere FPGAs) und OpenCL erklärt. Nachfolgend werden vergleichbare Arbeiten betrachtet. Im vierten Kapitel geht es um die Implementierung von DeepRacin und die Änderungen, welche für die FPGA Unterstützung notwendig waren. Das letzte Kapitel zeigt die Ergebnisse der Tests mit der entstandenen Software und gibt einen Ausblick auf möglichr, künftige Entwicklungen von DeepLearning Frameworks mit FPGA Unterstützung.

Kapitel 2

Grundlagen

2.1 Maschinelles Lernen

Das *maschinelle Lernen* (ML) ist ein Teilbereich der künstlichen Intelligenz, bei dem Probleme gelöst werden, indem sich am Lernverhalten von Lebewesen orientiert wird. In klassischen Ansätzen lösen Computerprogramme Probleme, indem alle notwendigen Informationen und Abfragen fest einprogrammiert werden. Beim maschinellen Lernen hingegen werden Algorithmen verwendet, welche anhand von vorhandenen Daten lernen, wie das vorliegende Problem zu lösen ist. Dies ist in vielen Bereichen notwendig, in denen die betrachtete Struktur zu komplex ist, um explizite Regeln zu formulieren [31, Preface]. Verwendung finden diese Verfahren in alltäglichen Bereichen wie dem autonomen Fahren, der Gesichtserkennung mit Smartphones, sowie in wissenschaftlichen Anwendungen wie in der Bioinformatik, der Medizin oder der Astrophysik. Wenn für das Training benannte Daten verwendet werden, also Daten, bei denen die Zuordnung feststeht und das Programm diese Zuordnung lernen soll, spricht man von *überwachtem Lernen* (supervised Learning). Von *unüberwachtem Lernen* (unsupervised Learning) spricht man, wenn das Programm selber erkennen soll, welche Auffälligkeiten bei den vorgegebenen Daten vorhanden sind [31, Seite 23]. Eine Anwendung, die in der Regel ein überwachtes Lernverfahren nutzt, ist die *Klassifikation*. Hierbei sollen Daten wie Fotos, Videos, Audiodaten, Texte, etc. bestimmten Klassen zugeordnet werden. Zum Beispiel ist bei der Handschrifterkennung von Ziffern jede Ziffer von 0 bis 9 eine Klasse. Allgemein formuliert ist das Ziel für gegebene Trainingsdaten $D = \{(x_i, y_i) | x_i \in \mathbb{R}^d, y_i \in \{0, 1\}\}$ Vorhersagefunktionen $f : \mathbb{R}^d \to \{0, 1\}$ zu finden, welche die Daten mit einer hohen Genauigkeit zuordnen. Nachdem eine solche Funktion gefunden wurde, kann diese auf neue Daten angewendet werden. Bei der Ermittlung einer Funktion, welche kontinuierliche Daten ausgibt spricht man hingegen von einer Regression [19, Seite 431]. Auch die Regression ist meist ein überwachtes Lernverfahren. Ein Beispiel für ein typischerweise unüberwachtes Lernverfahren ist das *Clustering*. Die Eingabedaten werden beim Clustering in Gruppen eingeordnet, welche durch das Verfahren anhand von

beobachteten Auffälligkeiten festgelegt werden [31, Seite 407 f.].
Ein Verfahren, welches Vorhersagen treffen soll, benötigt zur Bestimmung des Fehlers der Vorhersage $\hat{y}$ zu einer tatsächlichen Beobachtung y eine *loss-Funktion* (Fehlerfunktion) $E(\hat{y}, y)$. Hier muss eine geeignete Funktion wie der *RMSE* (Root Mean Squared Error) gewählt werden:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}$$

Um die Zeit für die Berechnung der Wurzel zu sparen, kann anstelle des RMSE auch der *MSE* (Mean Squared Error) verwendet werden. Bei nominalen Klassen lassen sich MSE und RMSE nur nutzen, wenn zwei Merkmalsausprägungen vorhanden sind. Ebenfalls für Daten mit zwei möglichen Merkmalsausprägungen, aber auch für nominale Klassen mit mehr als zwei Ausprägungen, kann die Kreuzentropie verwendet werden:

$$H = -\frac{1}{N}\sum_{i=1}^{N}(y_i ln(\hat{y_i}) + (1 - y_i) ln(1 - \hat{y_i}))$$

Bei der Wahl der Fehlerfunktion ist also das zu lösende Problem und die damit zusammenhängende Art der Daten (nominal, quantitativ) zu berücksichtigen.

2.2 Tiefe neuronale Netze

Ein überwachtes Lernverfahren im Bereich des maschinellen Lernens sind die künstlichen neuronalen Netze. Diese eignen sich für das Lösen von Klassifikationsproblemen, wofür die Gesichtserkennung in Smartphones ein Beispiel ist, bei dem das erkannte Gesicht der Klasse “Gesicht des Besitzers” oder “Anderes Gesicht” zugeordnet werden soll.
Ein neuronales Netz besteht aus mehreren *Layern* (Schichten) in denen sich ein oder mehrere Neuronen befinden. Jedes Neuron erhält Eingabewerte aus dem Layer vorher, führt Berechnungen mit Hilfe der erlernten Parameter durch und gibt seinen Ausgabewert an die nächste Schicht weiter. Man unterscheidet zwischen *feedforward* Netzen, bei denen sich die Daten nur in eine Richtung bewegen können und *recurrent* (rekurrenten) Netzen, welche es erlauben ein Layer innerhalb eines Durchlaufs mehrmals zu passieren [19, Seite 35]. Während rekurrente Netze besser für die Verarbeitung von Zeitreihen geeignet sind, werden feedforward Netze im Bereich der Mustererkennung eingesetzt. Da es in DeepRacin insbesondere um die Verarbeitung von Bilddaten geht, werden im folgenden nur feedforward Netze betrachtet. Neuronale Netze verfügen über unterschiedliche Arten von Layern. Bei Fully Connected Layern sind die Neuronen einer Schicht jeweils mit allen Neuronen der vorherigen Schicht verbunden (siehe Abbildung 2.1). Diesen Verbindungen werden Gewichtungen zugeordnet, wodurch die Verbindung mancher Neuronen stärker ist als die Verbindung zu Anderen. Während des Trainings werden diese Gewichtungen angepasst, wodurch ein feedforward Netzwerk aus Fully Connected Layern die Klassifikation

von Daten vornehmen kann. Die erste Schicht wird als Input Layer bezeichnet, da diese die Eingabewerte enthält und an die nächste Schicht weitergibt. Die letzte Schicht ist das Output Layer. Die Anzahl der Neuronen dieser Schicht entspricht der Anzahl der vorhandenen Klassen. Jedes Neuron dieser Schicht gibt dabei die Wahrscheinlichkeit der Zugehörigkeit zu der entsprechenden Klasse aus.
Um ohne tiefgreifendes domänenspezifisches Wissen *Features* (Merkmale) durch das neuronale Netzwerk zu extrahieren, lassen sich convolutional Layer nutzen. Diese lernen Gewichtungen, mit denen zum Beispiel aus einem Bild mehrere Bilder gemacht werden, um mehrere Features getrennt betrachten zu können [22, Seite 439].
Die Rechenlast kann durch Pooling Layer reduziert werden. Diese dienen der Vorverarbeitung von Daten. Beispielsweise können Bilder mit Hilfe von Pooling Layern komprimiert werden, wodurch weniger Eingabewerte an die Fully Connected und convolutional Layer gelangen. Die Verwendung vieler Layer, wie Convolutional Layer, Pooling Layer oder Fully Connected Layer, hintereinander führt zu dem Begriff der tiefen neuronalen Netze und des Deep Learnings.

2.2.1 Fully Connected Layer

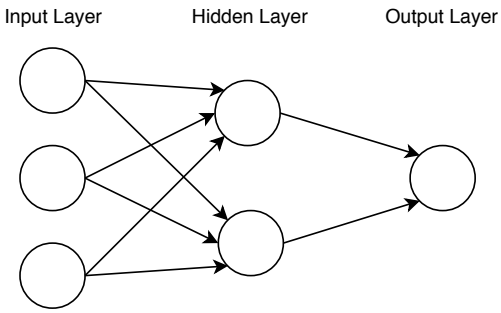


Abbildung 2.1: Feed Forward Fully Connected Netzwerk mit drei Schichten

In einem Netzwerk mit L fully connected Layern, in denen sich jeweils M^l Neuronen befinden, verfügt jedes Layer l über Gewichtungen $w_{i,j}^{(l)}$ mit $j \in [0; M^{(l)}], i \in [0; M^{(l+1)}]$ der einzelnen Input-Werte. Außerdem besitzt jedes Neuron einen Bias-Wert $b_j^{(l)}$.
Der Ausgabewert eines Neurons wird durch eine Aktivierungsfunktion h bestimmt. Um eine kontinuierliche Output-Funktion zu erhalten, kann die sigmoide Aktivierungsfunktion gewählt werden:

$$\sigma(z) = \frac{1}{1 + e^{-\beta \cdot z}}, \beta \in \mathbb{R}_{>0}$$

Eine weitere verbreitete Aktivierungsfunktion ist die *ReLu* (rectified linear) Funktion:

$$h(z) = max(0, z)$$

Der Output eines Neurons berechnet sich nun durch die Anwendung der Aktivierungsfunktion auf die gewichtete Summe der Input Werte:

$$f_j^{(l)} = h(y_j^{(l)})$$

$$y_j^{(l)} = \sum_{i=0}^{M^{(l-1)}} w_{i,j}^{(l)} \cdot f_i^{(l-1)} + b_j^{(l)}$$

Das Ziel beim Trainieren des neuronalen Netzwerkes ist es, die Gewichte und Bias-Werte zu optimieren. Hierbei wird das ganze Netzwerk mit Hilfe des Backpropagation-Algorithmus rekursiv durchlaufen und die Werte für die Gewichtungen werden aktualisiert. Backpropagation basiert auf dem Gradientenabstieg, welcher ein numerisches Verfahren ist, um Optimierungsprobleme zu lösen.
Gradienten Schritt [4]:

$$w_{i,j}^{(l)} = w_{i,j}^{(l)} - \alpha \cdot \delta_j^{(l)} \cdot f_i^{(l-1)}$$

$$b_j^{(l)} = b_j^{(l)} - \alpha \cdot \delta_j^{(l)}$$

Rekursion [4]:

$$\delta_j^{(L)} = \frac{\partial E(\hat{y}_i^{(L)}, y)}{\partial y_i^{(L)}} \cdot \frac{\partial h(y_i^{(L)})}{\partial y_i^{(L)}}$$

$$\delta_j^{(l-1)} = \frac{\partial E(\hat{y}_i^{(l-1)}, y)}{\partial y_i^{(l-1)}} \cdot \sum_{k=1}^{M^{(l)}} \delta_k^{(l)} \cdot w_{j,k}^{(l)}$$

α ist die Lernrate bzw. die Schrittweite des Gradienten Schrittes. Da die ReLu Funktion an der Stelle 0 nicht differenzierbar ist, wird in der Praxis häufig der Wert 0 für die Ableitung an dieser Stelle verwendet. Ein Netzwerk, welches nur aus Fully Connected Layern besteht, ist grundsätzlich bereits geeignet Klassifizierungsprobleme zu lernen. Um hohe Genauigkeiten zu erreichen und die Rechenlast zu reduzieren, werden weitere Layer verwendet.

2.2.2 Pooling Layer

Für die Vorverarbeitung von Bildern mit Pooling Layern muss die Art der Repräsentation dieser Bilder berücksichtigt werden. Hierzu gibt es zwei Möglichkeiten. Die eine Variante sind Vektorgrafiken, bei denen die Formen im Bild durch Formeln beschreiben werden. Zum Beispiel wird ein Kreis durch die Position des Mittelpunktes, den Radius und die Liniendicke, -farbe beschrieben. Bei der anderen Variante, den Pixelgrafiken, besteht ein Bild aus einem Raster von Bildpunkten, welche jeweils einen oder mehrere Farbkanäle mit Zahlenwerten für die Intensität der Farbe besitzen. Fotos und Ausschnitte von Videoaufnahmen bestehen immer aus Pixelgrafiken, da es zu aufwendig ist reale Situationen durch

Vektorgrafiken zu beschreiben. Für die Objekterkennung sind daher Pixelgrafiken die relevantere Variante und werden im Folgenden betrachtet. Für die Klassifikation von Bildern mit einem reinen Fully Connected Netzwerk ist die Rechenlast sehr hoch, da Bilder in der Regel über mehrere tausend Pixel mit jeweils drei Farbkanälen verfügen. Es ist also in vielen Fällen notwendig, vor dem ersten Fully Connected Layer die Bilder zu verkleinern. Hierzu können Pooling Layer verwendet werden, welche nach verschiedenen Verfahren Bilddaten komprimieren. Eine Möglichkeit ist, von jeweils $r \times r$ Pixeln den Durchschnittswert zu bilden [4]:

$$k = \frac{1}{r^2} \sum_{i=0}^{r} \sum_{j=0}^{r} p_{i,j}$$

k ist dabei der Wert des neu entstandenen Pixels, während die zusammengefassten Pixel die Werte $p_{i,j}$ haben.
Eine andere Methode ist das max-Pooling, bei dem nicht der Durchschnitt, sondern der größte Wert entscheidend ist:

$$k = max(p_{i,j}, p_{i,j+1}, ..., p_{i,j+r}, p_{i+1,j}, ..., p_{i+r,j+r})$$

Eine weitere Komprimierung ist durch die Summierung der Werte möglich:

$$k = \sum_{i=0}^{r} \sum_{j=0}^{r} p_{i,j}$$

Mit diesen Verfahren kann ein ganzes Bild komprimiert werden, indem ein Raster Schritt für Schritt über das Bild geschoben wird. Die Größe des Rasters entspricht der Anzahl der Pixel, die zusammengefasst werden. Bei der Implementierung dieser Methode muss berücksichtigt werden, ob bestimmte Pixel mehrfach in die Berechnung neuer Pixel einbezogen werden (Overlapping) und was passiert, wenn Pixel außerhalb des Bildbereiches für die Berechnung notwendig sind (Padding). Diese Methoden lassen sich ebenfalls zum Zusammenfassen der drei Farbkanäle nutzen, da in manchen Fällen keine Unterscheidung zwischen verschiedenen Farben notwendig ist (zum Beispiel: Handschrifterkennung). Anstatt das Maximum der drei Farbkanäle (r, g, b) zu verwenden, kann auch die Helligkeit des Bildpunktes berechnet werden. Eine Variante dafür ist die *Lightness* [4]:

$$p_{i,j} = (max(r_{i,j}, g_{i,j}, b_{i,j}) - min(r_{i,j}, g_{i,j}, b_{i,j}))/2$$

i,j sind die Indizes in der Pixelmatrix.
Ein Verfahren, welches auf gewichtete Summen setzt, ist die Berechnung der Luminanz [4]:

$$p_{i,j} = 0.21 r_{i,j} + 0.72 g_{i,j} + 0.07 r_{i,j}$$

Anstelle der Faktoren 0.21, 0.72, 0.07, können die Gewichtungen für die Summe mit Hilfe der Backpropagation gelernt werden, um eine problemspezifischere Komprimierung durchzuführen.

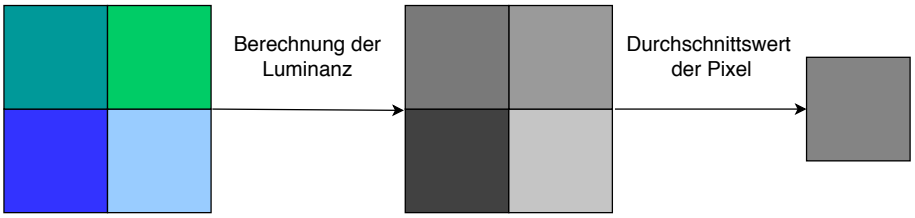


Abbildung 2.2: Komprimierung von vier RGB Pixeln in ein graustufen Pixel

2.2.3 Convolutional Layer

Um die Komprimierung von Bildern mit Gewichtungen durchzuführen und um Features zu extrahieren, können Layer, welche die Convolution anwenden, verwendet werden. Ursprünglich wurde dieses Verfahren unabhängig voneinander in der Statistik und der Signalverarbeitung entwickelt. In der Statistik wird es verwendet, um die Ähnlichkeit von zwei Funktionen zu bestimmen, und in der Signalverarbeitung, um die Reaktion eines Systems mit der Transferfunktion f auf ein Eingangssignal g zu berechnen:

$$f * g = \int f(\tau)g(t-\tau)d\tau$$

In der diskreten Datenverarbeitung wird das Integral zu einer Summierung. Für die Funktion f werden die Gewichtungen eingesetzt und für g werden die Pixelwerte eingesetzt. Dadurch ergibt sich folgende gewichtete Summe mehrerer Pixel:

$$k_c = \sum_{i=1}^{r} w_i \cdot c_i = \vec{w} * \vec{c}$$

Da ein quadratisches Gitter von Pixeln für die Berechnung betrachtet wird, sind zwei Summen notwendig. Analog zu der Berechnung der Ausgabe eines Fully Connected Neurons ergibt sich der Output eines Convolutional Neurons mit Aktivierungsfunktion h [4]:

$$f_{i,j}^{(l)} = h\left(\sum_{i'=0}^{M(l)}\sum_{j'=0}^{M(l)} w_{i,j}^{(l)} \cdot f_{i+i',j+j'}^{(l-1)} + b_{i,j}^{(l)}\right) = h\left(w^{(l)} * f^{(l-1)} + b^{(l)}\right)$$

Auch hier kann zusätzlich ein Bias-Wert $b_{i,j}$ aus einer Bias-Matrix aufaddiert werden. Die Backpropagation für ein Convolution Layer besteht aus folgenden Schritten: Gradienten Schritt [4]:

$$w_{i,j}^{(l)} = w_{i,j}^{(l)} - \alpha \cdot \delta_j^{(l)} * rot180(f)^{(l-1)} \cdot f_{i,j}^{(l-1)}$$

$$b_j^{(l)} = b_j^{(l)} - \alpha \cdot \delta_j^{(l)}$$

Rekursion:

$$\delta^{(l+1)} = \delta^{(l)} * rot180(w^{(l+1)}) \cdot \frac{\partial h(y_i^{(l)})}{\partial y_i^{(l)}}$$

Ein Rekursionsschritt für das letze Layer ist hier nicht notwendig, da ein Convolution Layer keine Vorhersage für eine Klassifikation trifft und nach der Convolution weitere Layer folgen. Die Gewichtungsmatrix eines Convolution-Neurons bezeichnet man auch als Kernelmatrix. Durch die Verwendung verschiedener Kernelmatrizen, werden unterschiedliche Filtereffekte auf die Daten angewendet. Abbildung 2.3 zeigt ein Bild, auf welches die Convolution mit zwei unterschiedlichen Kernelmatrizen angewandt wurde. Die erste Convolution sorgt für eine Schärfung des Bildes, während die Zweite eine Kantenerkennung durchführt. Einen guten Kompromiss zwischen einer hohen Genauigkeit und einer reduzierten Rechenlast lässt sich häufig durch die abwechselnde Verwendung von Pooling und convolutional Layern erzielen.

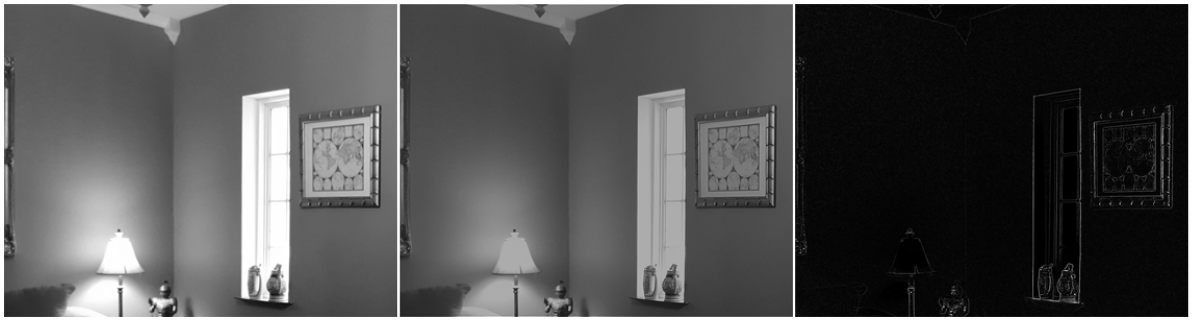


Abbildung 2.3: Beispiele für die Anwendung von zwei verschiedenen Convolution Filtern. Links befindet sich das original Bild, auf das Bild in der Mitte wurde ein Convolution Filter zur Schärfung angewandt und bei dem rechten Bild wurde eine Kantenerkennung durchgeführt.

2.3 Spezialisierte Hardware

Die Verbreitung tiefer neuronaler Netze geht mit der Entwicklung schnellerer Hardware einher, da insbesondere das Training sehr rechenintensiv ist und eine schnellere Hardware mehr Trainingszyklen und größere Eingabedaten erlaubt. Daraus folgt, dass schwierigere Probleme gelernt werden können und die Genauigkeit bei den Vorhersagen steigen kann. Klassische Single Core CPUs stellen in der Regel nicht die benötigte Rechenleistung zur Verfügung, um Probleme mit tiefen neuronalen Netzen zu trainieren. Heute sind die meisten Computer mit Multi Core CPUs ausgestattet, welche über 2 - 8 Kerne verfügen, also 2-8 Berechnungen gleichzeitig ausführen können. Um alltägliche Anwendungen zu Beschleunigen, verfügen heutige CPUs über ein hardwareseitiges Hyperthreading. Hierbei sind die CPU Kerne mit jeweils zwei Load/Store Einheiten, Registern und anderen Komponenten ausgestattet, während die Anzahl der *ALUs* (Arithmetisch Logische Recheneinheiten) nicht verdoppelt wird[16, Seite224]. So können zwei verschiedenartige Befehle zeitgleich aus dem Speicher geladen, gleichzeitig ausgeführt und schließlich gleichzeitig zurückgeschrieben werden. Dies ist nur möglich, wenn es sich bei den Berechnungen um unterschiedliche

Operationen handelt (Beispielsweise Addition und Multiplikation), da sonst die gleiche Recheneinheit beansprucht wird. Für Vektoroperationen ist daher fast kein Performance-Gewinn durch das Hyperthreading vorhanden, da hier mehrfach die gleichen Operationen durchgeführt werden.
Moderne CPUs sind auf eine hohe Leistung bei unterschiedlichen Befehlen für unterschiedliche Daten ausgelegt (multi instruction, multiple data, MIMD). Neuronale Netze benötigen hingegen eine hohe Performance bei Vektor- und Matrixoperationen. Seit 2008 existiert eine Erweiterung des x86-Befehlssatzes mit dem Ziel Vektoropertionen auf CPUs zu beschleunigen. Diese Erweiterungen mit dem Namen *AVX* (Advanced Vector Extensions) existieren in unterschiedlichen Varianten (AVX, AVX2, AVX512) in modernen Prozessoren. Sie ermöglichen es Operationen auf Daten in speziellen, größeren Registern auszuführen. In [13] wurden verschiedene Implementierungen von Vektoroperationen verglichen und es wurde festgestellt, dass AVX eine bessere Leistung erzielt, aber die höchste Geschwindigkeit mit GPUs zu erreichen ist. Für SIMD Operationen optimierte Hardware ist daher in den meisten Fällen besser für die Berechnungen von neuronalen Netzen geeignet.
Die folgende Tabelle gibt einen auf [12, Seite 10] basierenden Überblick über die verschiedenen Typen von paralleler Hardware mit Beispielen und Einsatzgebieten:

Parallelisierung	Beispiele	Einsatzgebiet
SISD (Single Instruction Single Data)	Single-Core CPU	Sequentielle Computerprogramme
SIMD (Single Instruction Multiple Data)	GPU, Vektorprozessor	Berechnung von Vektoren/Matrizen, Multimedia Anwendungen
MISD (Multiple Instruction Single Data)	Keine kommerziellen Prozessoren vorhanden	Datenstrom, welcher Berechnungsmatrix durchläuft (systolisches Array)
MIMD (Multiple Instruction Multiple Data)	Multi-Core CPU	Berechnung mehrerer Programme gleichzeitig, Programmen mit verschiedenartigen Operationen

Die Art der Parallelisierung bei FPGAs wird durch die Konfiguration bestimmt. Daher sind FPGAs prinzipiell als MIMD Rechner geeignet. Es können allerdings Optimierungen für verschiedene Arten von Parallelisierungen durchgeführt werden. Die parallele Berechnung verschiedenartiger Funktionen, welche auf MIMD Hardware möglich ist wird auch als *task level* Parallelität bezeichnet. Eine Parallelisierung einer einzelnen Aufgabe, indem die zu berechnenden Daten auf mehrere Recheneinheiten aufgeteilt werden (SIMD), entspricht dagegen der *data level* Parallelität.

2.3.1 GPUs

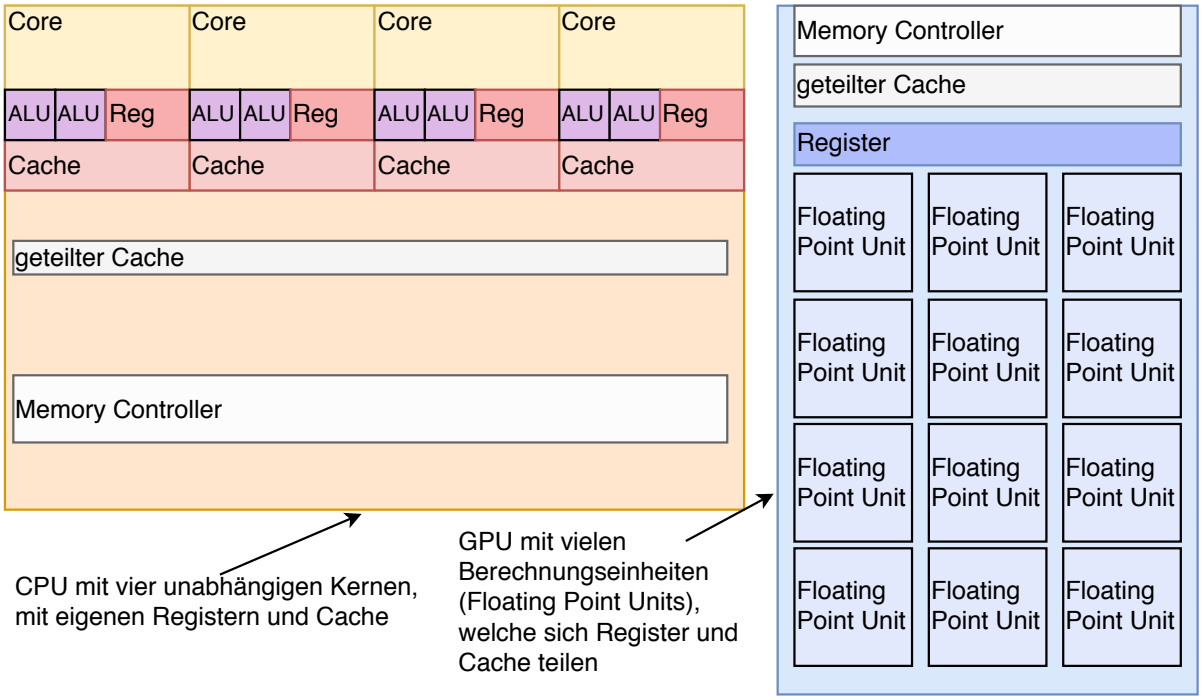


Abbildung 2.4: Vergleich des schematischen Aufbaus von Multi-Core CPUs und GPUs.

Grafikprozessoren verfügen über eine Architektur, welche viele Berechnungen mit der gleichen Operation gleichzeitig ausführen kann. Während CPUs darauf optimiert sind, Latenzen, beispielsweise durch Speicherzugriffe, zu verbergen, sind GPUs für einen möglichst hohen Datendurchsatz optimiert [28, Seite 879]. Diese Eigenschaft ist durch den ursprünglichen Einsatzbereich der Grafikprozessoren bedingt, denn in der Regel müssen in 3D Anwendungen viele Objekte gleichzeitig gerendert werden. Berechnungen mit großen Vektoren sind also auf GPUs deutlich effizienter, weswegen diese für tiefe neuronale Netze gut geeignet sind. In Abbildung 2.4 sind schmeatische Abbildungen von einer Multi-Core CPU und einer GPU nebeneinander dargestellt. Hier ist die höhere Anzahl von Berechnungseinheiten (Floating Point Units) als bei CPUs (Cores), sowie das gemeinsame Register erkennbar. In [28, Seite 880ff.] wird die Entwicklung der GPUs von der Verwendung einer reinen Grafikpipeline hin zu einer freier programmierbaren Architektur beschrieben. Ursprünglich verfügten GPUs über eine Pipeline, welche in einer festen Abfolge Dreiecke in einem 3D Koordinatensystem berechnet. Heute ist die GPU ein zum größten Teil vom Nutzer programmierbarer Prozessor, welcher zusätzlich einige spezialisierte Recheneinheiten besitzt. In diesem Zusammenhang spricht man heute häufig von *General Purpose GPUs* (GPG-

PUs). Diese Entwicklung ging ursprünglich von komplizierter werdenden Schatten- und Lichtberechnungen aus, die eine flexiblere Programmierung erforderten. Heute ist die Rechenleistung von GPUs nicht nur für grafische Berechnungen interessant. Mit den NVIDIA Titan V [27] und AMD Radeon Instinct MI25 Accelerator [1] Grafikkarten werden heute sogar GPUs angeboten, welche für Berechnungen abseits von grafischen Anwendungen optimiert sind. Ein Nachteil der hohen Rechenleistung von GPUs ist der hohe Energieverbrauch im Vergleich zu FPGAs [3]. Es werden zwar auch GPUs für das mobile Segment angeboten, welche deutlich sparsamer sind, aber bei stark begrenzten Energieressourcen kann auch dieser Verbrauch zu hoch sein.

2.3.2 ASICs

Die ursprüngliche Idee der GPUs war es, eine anwendungsspezifische Schaltung für grafische Anwendungen zu entwickeln. Mittlerweile gehören GPUs genau wie CPUs zu programmierbaren Hardwarekomponenten, welche man auch *PLDs* (programmable logic devices) nennt. Im Gegensatz zu *ASICs* (application-specific integrated circuits) kann, auch nachdem der Chip produziert wurde, entschieden werden, welche Aufgaben dieser lösen soll. Bei einem ASIC wird die logische Schaltung auf einen einzigen Algorithmus angepasst. Wenn der ASIC erst einmal hergestellt wurde, sind keine Änderungen am Algorithmus möglich. Der Vorteil dieser spezialisierten Chips ist, dass diese über eine hohe Leistung bei geringem Energiebedarf verfügen. Sie erreichen die höchste Anzahl von *GOP* (giga operations) pro Joule [24, Seite 133]. Ein großer Nachteil von ASICs ist die geringe Flexibilität, da eine Änderung des verwendeten Algorithmus einen neuen Chip erfordert. Außerdem lohnt sich die Produktion von ASICs erst ab einer sehr großen Stückzahl, da die Fixkosten für die Produktion sehr hoch sind und die Entwicklung der Schaltkreise einen großen Arbeitsaufwand erfordert [24, Seite 135]. Der hohe Arbeitsaufwand sorgt neben hohen Entwicklungskosten dafür, dass der Entwicklungszeitraum sehr lang ist. Dies birgt die Gefahr, dass das implementierte Verfahren veraltet ist, wenn die Chips in die Produktion gehen.
ASICs, welche für Algorithmen im Bereich des DeepLearnings entwickelt wurden, sind beispielsweise die *TPUs* (Tensor Processing Units) von Google, die nur für neuronale Netze verwendet werden können [18].

2.3.3 FPGAs

FPGAs liegen mit ihrer Leistung, dem Energieverbrauch und der Programmierbarkeit genau zwischen CPUs/GPUs auf der einen und ASICs auf der anderen Seite. Ihre Leistung ist höher als bei klassischen Prozessoren und sie lassen sich im Gegensatz zu ASICs rekonfigurieren. Dafür sind sie etwas langsamer und weniger sparsam als ASICs und schwieriger zu programmieren als PLDs. Aus diesen Gründen sind FPGAs einerseits gut geeignet, um als Prototypen für ASICs zu dienen, da die Produktion von nicht rekonfigurierbaren Chips

als Prototypen zu teuer wäre. Andererseits lassen sich FPGAs dort einsetzen, wo sich die Produktion von ASICs nicht lohnt, da wenige Anwendungen benötigt werden oder die Verfahren sich häufig ändern.
In [24, Seite 152 ff.] wird die Funktionsweise von FPGAs kurz erklärt. Ein FPGA besteht aus einem Gitter von *CLBs* (Configurable Logic Blocks). Diese können durch die Konfiguration des FPGAs miteinander verbunden und getrennt werden. Jeder CLB besteht aus slices, welche mit Speicherzellen ausgestattet sind, in denen Look-Up Tables hinterlegt sind. Die Größe der Look-Up Tables entspricht der Anzahl aller möglichen boolschen Funktionen, die mit der vorhandenen Anzahl von Ein- und Ausgängen umsetzbar sind. Durch die Konfiguration der look-up table können logische Funktionen wie AND, OR, XOR, NAND, etc. innerhalb von CLBs implementiert werden. Es können auch deutlich komplexere Funktionen implementiert werden, da ein Slice typischerweise mehr als 2 Eingänge besitzt und ein CLB aus zwei Slices besteht. Die Speicherblöcke lassen sich auch als regulärer RAM verwenden. Da es sich hier bei den meisten FPGAs um sehr schnellen SRAM handelt, ist dieser Speicher für sehr kleine Datenmengen geeignet, auf die häufig zugegriffen wird (vergleichbar mit den Registern einer CPU). Die Konfiguration eines FPGAs ergibt sich also aus den Verbindungen der CLBs, dem Inhalt der SRAM Zellen und der Einstellung der Slice-Multiplexer. Da eine hohe Anzahl von Komponenten benötigt wird, um Multiplizierer auf einem FPGA zu platzieren, werden *DSP* (Digital Signal Processor) Slices als eigenständige Blöcke auf FPGAs verbaut. DSP Slices sind vergleichbar mit den Floating Point Units von GPUs. Abbildung 2.5 zeigt eine vereinfachte schematische Ansicht eines FPGAs mit CLBs, DSP Slices, BRAM Blöcken und IO-Blöcken für die Kommunikation mit externer Hardware. Durch das Mapping von Funktionen und Speicher auf den FPGA ergibt sich eine Recheneinheit, die mit ASICs vergleichbar ist, allerdings einen Overhead durch nicht vollständig genutzte Bereiche und längere Kommunikationswege besitzt. Dieser Overhead sorgt für eine geringere Rechenleistung pro Joule als bei einem ASIC. Um den Einsatz von FPGAs flexibler zu gestalten, wird bei vielen Modellen zusätzlicher DRAM mit dem FPGA verbunden. Dieser ist für größere Datenmengen geeignet als der SRAM in den CLBs. Die Programmierung eines FPGAs kann mit einer Hardwarebeschreibungssprache wie VHDL oder Verilog erfolgen. Hierbei wird jede logische Funktion beschrieben, welche auf dem FPGA implementiert werden soll. Um den Programmieraufwand zu verringern, können High Level Synthese Tools eingesetzt werden. Diese erzeugen aus dem Programmcode höherer Programmiersprachen wie C, C++ oder OpenCL Code in einer Hardwarebeschreibungssprache und mappen diesen auf den FPGA. Hierbei muss beachtet werden, dass in dem Code der höheren Programmiersprache keine dynamischen Datenstrukturen verwendet werden, denn zum Synthesezeitpunkt muss feststehen, welche Bereiche die Komponenten auf dem FPGA einnehmen. Während der Laufzeit kann an der FPGA Konfiguration nichts mehr geändert werden.
Die größten Hersteller von FPGAs sind Intel (früher Altera) und Xilinx. Beide Hersteller

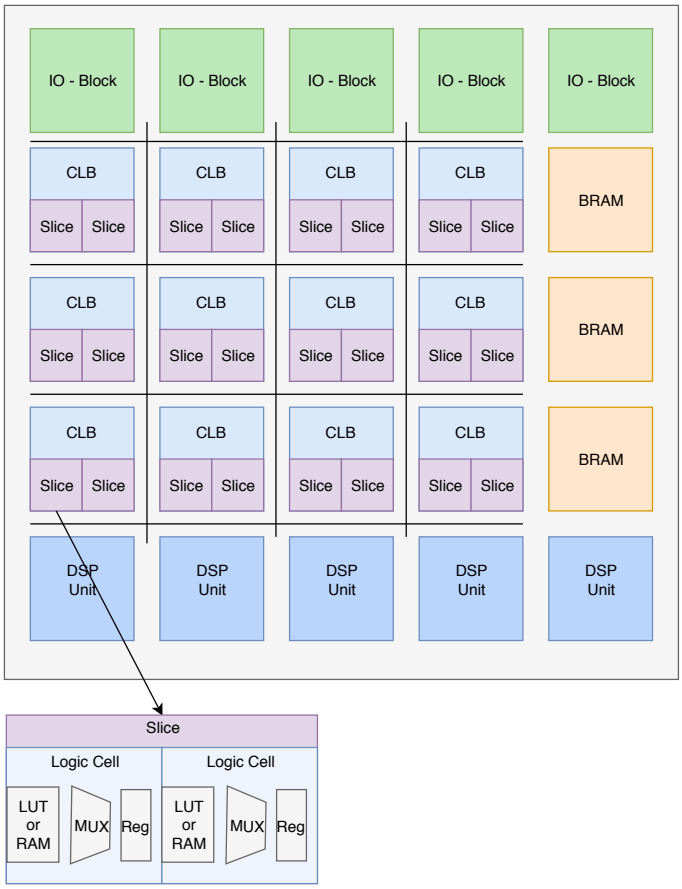


Abbildung 2.5: Schematischer Aufbau eines FPGAs ohne eingezeichnete Verbindungen.

bieten High Level Synthese Tools an, welche unter Anderem C, C++ und OpenCL unterstützen. Um die Verwendbarkeit von FPGAs für Deep Learning zu verbessern, müssen die Speicher weiter vergrößert und der Datenaustausch bei der Verbindung mehrerer FPGAs miteinander beschleunigt werden [20].

2.4 OpenCL

OpenCL ist eine Programmiersprache, welche für heterogene Systeme entwickelt wurde. Die erste Spezifikation wurde 2008 von den Unternehmen AMD, Apple, IBM, Intel und Nvidia erarbeitet. Die Intention dieser Unternehmen war, die Leistung ihrer Hardwareprodukte für ihre Softwareprodukte verfügbar zu machen und Entwicklern die Möglichkeit zu geben eine Parallelisierung ihrer Programme auf heterogener Hardware durchzuführen. Umgesetzt wird dies, indem Hardwarehersteller für ihre Produkte (Multi Core CPUs, GPUs, FPGAs, DSPs, etc.) eigene OpenCL Compiler entwickeln. Diese Compiler übersetzen den standardisierten OpenCL Code in die entsprechende Assemblersprache, welche die

Vorteile der parallelen Hardware nutzt. Die Kompilierung des OpenCL Codes erfolgt erst zur Laufzeit. Das hat den Vorteil, dass erst die vorhandene Hardware ermittelt werden kann und der entsprechende Compiler den Code auf diese Hardware zuschneiden kann. Auf diese Weise kann der Programmierer seinen OpenCL Code ohne Anpassungen auf allen Geräten nutzen, welche OpenCL unterstützen. Es ist ebenfalls möglich verschiedene Hardware wie eine Multi-Core CPU und eine GPU gleichzeitig innerhalb eines Programms zu verwenden. Heute sind deutlich mehr Unternehmen an der Entwicklung von OpenCL Compilern beteiligt als 2008. Als Hersteller von FPGAs bietet auch Xilinx mit dem xocc Compiler [35, Seite 34] eine Unterstützung für OpenCL an. Die Syntax von OpenCL basiert auf der C Syntax. Es werden allerdings zusätzliche Datentypen wie *half* für 16-bit Gleitkommazahlen, Vektordatentypen oder Datentypen für Bilder. Im Gegensatz zu C gibt es in OpenCL keine Funktionszeiger. Des Weiteren dürfen Arrays in OpenCL keine variable Länge haben. Diese sind wie zuvor beschrieben, bei der Hardwaresynthese für FPGAs ebenfalls nicht erlaubt.

2.4.1 Berechnungsmodell

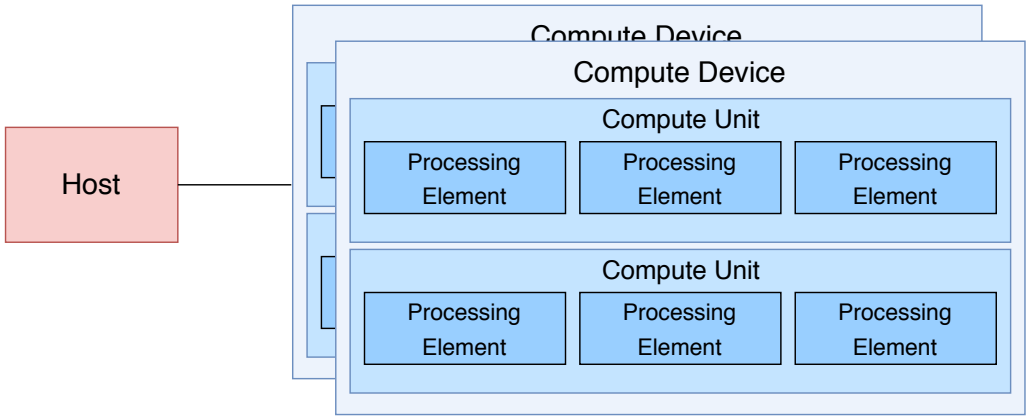


Abbildung 2.6: Host (links) und OpenCL Kernels (rechts).

Ein Programm, in dem OpenCL genutzt wird, besteht immer aus einem Host- und einem Kernelcode. Der Hostcode wird in C oder C++ geschrieben und vorab kompiliert, während der Kernelcode in OpenCL geschrieben und zur Laufzeit kompiliert wird. Koprozessoren werden innerhalb der OpenCL Terminologie als *Compute Devices* bezeichnet. Abbildung 2.6 zeigt den Aufbau eines Compute Devices im OpenCL Modell. Ein Host kann auf mehrere Compute Devices zugreifen, beispielsweise auf mehrere GPUs oder eine CPU und eine GPU. Compute Devices bestehen aus mehreren Compute Units. Compute Units werden parallel ausgeführt und sind bei CPUs mit den Prozessorkernen gleichzusetzen. Auf der untersten Ebene stehen die Processing Elements. Jede Compute Unit besteht in der Regel aus mehreren dieser Einheiten [17, Seite 1]. Processing Elements führen einzelne

Operationen durch und entsprechen somit bei CPUs den ALUs. Im Hostcode findet der Aufruf der OpenCL Kernels statt. Dabei werden die Daten auf den Koprozessor kopiert und die auszuführenden Befehle an die Compute Units übertragen [32, Folie 42].

In [17, Seite 2f] wird die parallele Ausführung von Aufgaben im OpenCL Berechnungsmo-

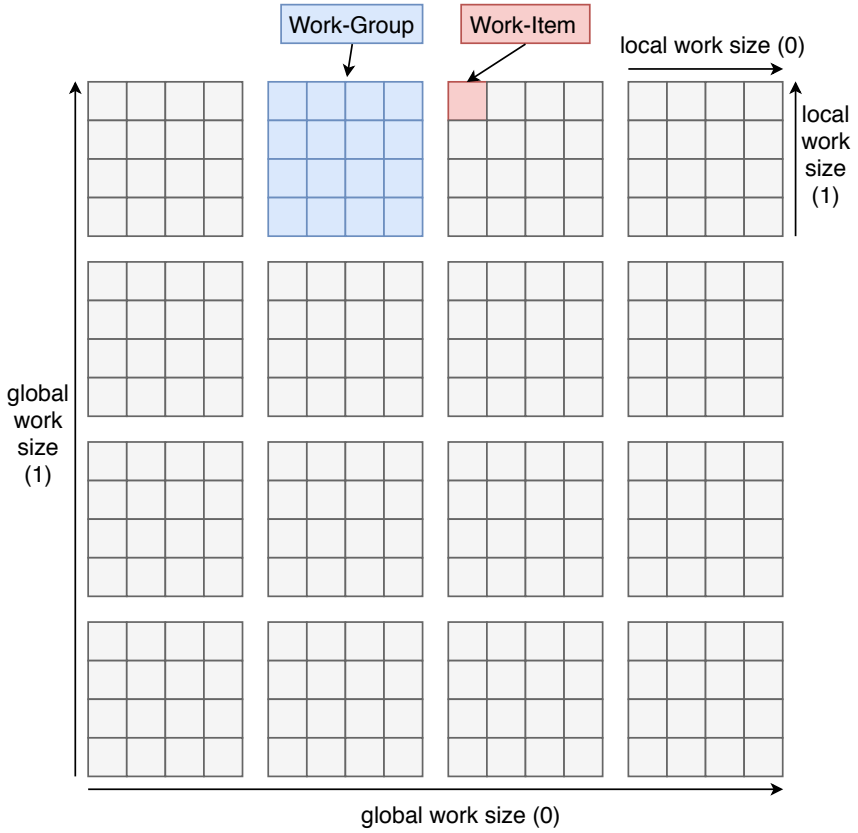


Abbildung 2.7: Aufbau eines zweidimensionalen Berechnungsfeldes in OpenCL.

dell erklärt. Der Host definiert ein N-dimensionales Berechnungsfeld, dessen Indizes jeweils ein Work-Item bestimmen. Jedes Work-Item führt den gleichen Kernelcode aus und gehört zu einer Work-Group. Diese Struktur ermöglicht eine Thread-Level Parallelisierung für die Work-Groups, da jede Gruppe einen eigenen Speicher hat, den sich die zugehörigen Work-Items teilen können. Auf der Ebene der Work-Items existiert eine Daten-Level-Parallelität. Bei der Programmierung von OpenCL Kernels ist zu beachten, dass innerhalb dieser Kernels keine Rekursionen möglich sind. Das Berechnungsmodell erlaubt dies nicht, da Rekursionen dem Konzept von parallelisierbaren Iterationen, welche vom Host in der CommandQueue verwaltet werden, widersprechen.

Beispiel: Parallelisierung einer for-Schleife, welche jeden Wert in einem Array um 1 erhöht:

```
for (int i = 0; i<maxVal; i++){
 arr[i]++; }
```

Die Kernels werden im Hostcode mit der Funktion:

```
clEnqueueNDRangeKernel(commandQueue, example_function,..., &maxVal, ...);
```

aufgerufen. Diese Funktion fügt die auszuführenden Befehle der commandQueue hinzu, startet den Kernel mit dem Namen example_function und übergibt mit &maxVal die Anzahl der Kernels. Für jeden Schleifendurchlauf wird nun ein Kernel erstellt. Die globale ID eines Kernels entspricht dem Wert der Laufvariablen, der for-Schleife. Der zugehörige Kernel Code sieht folgendermaßen aus:

```
__kernel void example_function(__global const int *arr) {
 arr[get_global_id(0)]++;
}
```

2.4.2 Speichermodell

Es gibt folgende vier Speichertypen in OpenCL: Host Memory, Global Memory, Local Memory und Private Memory. In Abbildung 2.8 ist die Zuordnung von Speichertypen zu Host, Work-Groups und Work-Items zu sehen. Die Daten des Host Speichers werden im

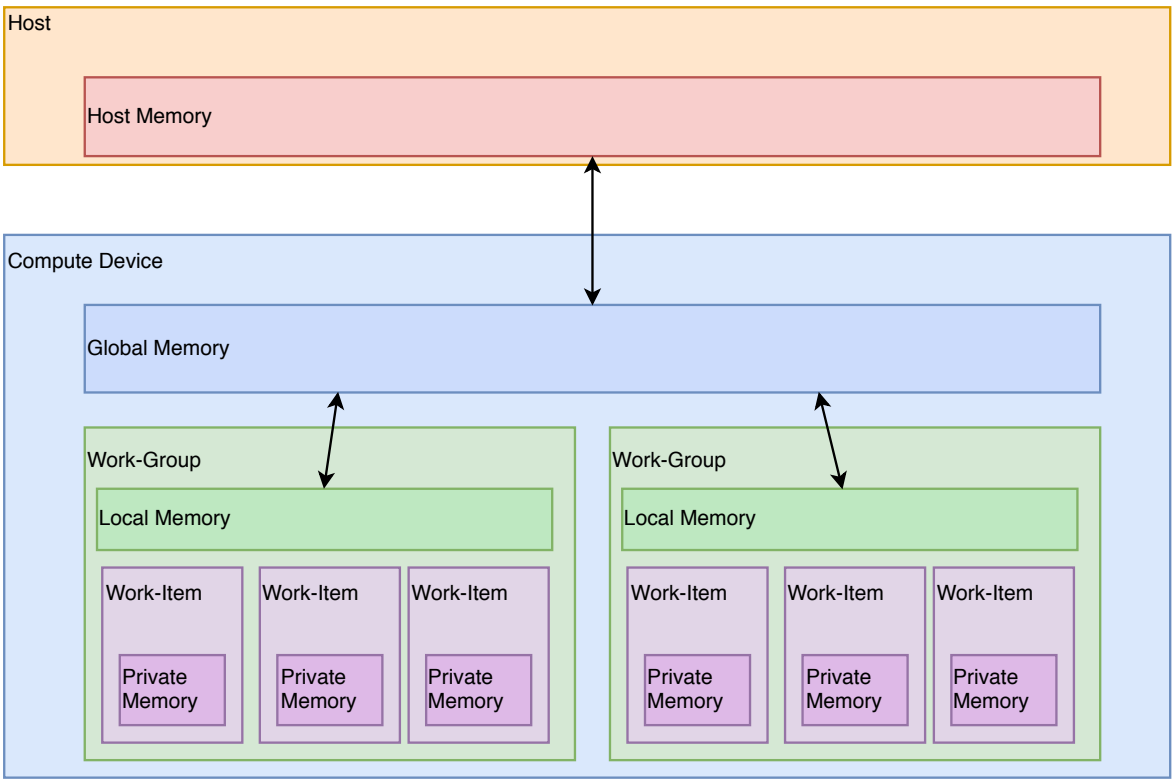


Abbildung 2.8: OpenCL Speichermodell

RAM der Host CPU abgelegt und im Host Programm kann jederzeit darauf zugegriffen werden. Die anderen drei Speichertypen befinden sich auf dem Compute Device. In den globalen Speicher des Compute Devices werden die Daten aus dem Host Memory kopiert. Hierzu dient die Funktion

```
clEnqueueWriteBuffer(commandQueue, in_mem_obj, ... , in, ...);
```

mit in_mem_obj als Buffer, in den die Daten geschrieben werden und dem Array in, welches die Daten enthält. Analog werden die berechneten Daten mit folgender Funktion zurückgeholt:

```
clEnqueueReadBuffer(command_queue, out_mem_obj, ... , out ,...);
```

Auf den globalen Speicher kann innerhalb des Compute Device jederzeit zugegriffen werden. Innerhalb von Workgroups gibt es den lokalen Speicher. Das einmalige Kopieren der Daten vom globalen in den lokalen Speicher kann einen deutlichen Performancevorteil bringen, denn der Zugriff auf den globalen Speicher ist sehr langsam. Variablen, welche während der Berechnungen in den Work-Items angelegt werden, werden im privaten Speicher des Work-Items abgelegt. Grundsätzlich muss bei einem OpenCL Programm immer berücksichtigt werden, dass ein Kopieren der Daten vom Host auf ein Compute Device sehr viel Zeit kostet. Bei geringem oder schlecht parallelisierbarem Rechenaufwand ist die Berechnung auf der Host CPU unter Umständen schneller. Für die Optimierung von OpenCL Programmen auf FPGAs sollten häufig verwendete Daten in den lokalen Speicher einer Workgroup kopiert werden. Dieser Speicher sollte bestenfalls direkt auf dem FPGA umgesetzt werden, um die schnellen SRAM Zellen der CLBs zu verwenden. Im Kontext von neuronalen Netzen können die Kopiervorgänge im Speicher reduziert werden, indem die Output-Werte eines Layers im lokalen Speicher behalten werden, bis diese für die Berechnungen im nächsten Layer verwendet werden. Je nach Größe bzw. Anzahl der miteinander verbundenen FPGAs sollten auch Gewichtungen im lokalen Speicher gehalten werden.

Kapitel 3

Verwandte Arbeiten

Vergleichbare Arbeiten zeigen in der Regel zwei verschiedene Ansätze. Der eine Ansatz ist die Entwicklung oder Erweiterung von Frameworks mit der Unterstützung für FPGAs. Diese Frameworks sollen für die Nutzer einfach im Umgang sein, verfügen allerdings häufig über eine geringere Leistung als mit anderen FPGA Implementierungen erreicht werden kann. Außerdem sind häufig nur Teile der Frameworks wie die Convolution Kernel für FPGAs angepasst. Ein anderer Ansatz zeigt die hohe erreichbare Leistung und den geringen Energiebedarf von hochoptimierten FPGA Implementierungen. Diese bieten allerings kein einfach zu händelndes Framework, welches zum Beispiel von Domänenexperten verwendet werden kann. Im Folgenden werden einige Arbeiten dieser beiden Kategorien vorgestellt. Zuerst werden Frameworks mit dem Fokus auf der Bedienbarkeit mit FPGA Integration vorgestellt.
In [6] wird eine Erweiterung des Caffe Frameworks vorgestellt, welche eine Implementierung der Winograd Convolution für FPGAs enthält. Der Artikel zeigt die Möglichkeit ein vorhandenes Framework für FPGAs zu erweitern. Dabei wird eine geringere Performance als bei vergleichbaren FPGA Implementierungen und eine geringere Performance als bei den CPU und GPU Implementierungen des Caffe Frameworks erzielt. Die angegebene Auslastung der Look Up Tables (LUT) liegt bei 52.9% und lässt somit die Implementierung weiterer Layer und/oder eine Verbesserung der Winograd Implementierung zu. Ein Vergleich des Energieverbrauchs fand nicht statt. LeFlow ist ein weiteres Framework für neuronale Netze auf FPGAs und wird in [5] präsentiert. Hier wurde eine Reihe von Tools hintereinander verknüpft, um aus Python Code Hardware Code zu generieren. Der Nutzer definiert hier ein neuronales Netzwerk in TensorFlow und der Google *XLA* (Accelerated Linear Algebra) Compiler generiert aus dem TensorFlow Python Code entsprechende Kernels in *LLVM* (Low Level Virtual Machine) Code. Die Autoren verwenden für den weiteren Verlauf den unoptimierten Code des XLA Compilers, da die Optimierungen es schwierig machen die Ein- und Ausgabe zu extrahieren. Im nächsten Schritt soll der Code mit dem LegUp Synthese Tool zu Verilog Code synthetisiert werden. Hierzu werden erst einige

Transformierungen durch LeFlow durchgeführt, um eine Schnittstelle zwischen XLA und LegUp herzustellen. Das Synthesetool LegUp bietet dem Nutzer Optimierungsmöglichkeiten, welche im Zuge von LeFlow bereits durch den Nutzer im ersten Schritt übergeben werden können. Die Autoren sehen für künftige Entwicklungen Optimierungspotential bei der Generierung der Kernels durch XLA, da diese bislang CPU optimiert sind. Außerdem könnten den Nutzern weitere Optimierungsmöglichkeiten, wie die Verwendung von individuell festgelegten Datentypgrößen, gegeben werden. Der in [9] vorgestellte Compiler *hls4ml* nutzt die Vivado High Level Synthese Tools, um aus Modellen neuronaler Netze *Register-Transfer Level* (RTL) Code zu erzeugen, welcher auf FPGAs ausgeführt werden kann. Das Hauptaugenmerk lag auf der Optimierung für eine geringe Latenz bei der Inferenz. Es wird beschrieben, dass diese für die echtzeit Verarbeitung von Daten in der Teilchenphysik notwenig ist. Um dies möglich zu machen, werden bei der Berechnung alle Layer unabhängig betrachtet, was ein Pipelining von mehreren Inferenzen ermöglicht. Für nicht-triviale Aktivierungsfunktionen, wie die sigmoide Funktion, werden vorabberechnete Werte in BRAM Blöcken gespeichert. Die ReLu Aktivierungsfunktion wurde direkt in Hardware implementiert. Zur weiteren Beschleunigung wurde eine reduzierte Genauigkeit bei Kommazahlen verwendet. Für die Tests wurde ein Xilinx Kintex Ultrascale FPGA verwendet. Dieser ist besonders für Berechnungen, welche viele DSPs in Anspruch nehmen, geeignet. Da neuronale Netze viele Multiplikationen benötigen, ist ein FPGA mit vielen DSPs gut dafür geeignet. Die Untersuchung der Auslastung von FPGA Komponenten in dem Artikel ergab, dass für das gewählte Beispiel nicht genügend DSP Einheiten vorhanden sind, wenn diese nicht mehrfach innerhalb der Berechnung eines Layers genutzt werden. Die LUT und FF (Flip Flop) Auslastung ist hingegen sehr gering. Für künftige Arbeiten stellen sich die Autoren unter Anderem eine Erweiterung für weitere Typen von Neuronalen Netzen, wie Convolutional Neural Networks vor. Mit hls4ml konnten viele Performanceoptimierungen erfolgreich durchgeführt werden, um eine höhere Geschwindigkeit als mit CPUs zu erreichen. Allerdings ist mit hls4ml nur die Möglichkeit der Inferenz auf FPGAs gegeben. Die Verwendung anderer Hardware wie Multi-Core CPUs oder GPUs ist mit diesem Framework nicht möglich. Ein Framework mit dem Fokus auf eingebetteten Systemen ist das fpgaConvNet [33]. Es bietet ein Caffe Frontend und verschiedene Optimierungen für einen hohen Durchsatz oder eine geringe Latenz. Die Autoren beschreiben, dass die Latenz deutlich verringert werden kann, wenn auf den Overhead durch *batch processing*, also der gleichzeitigen Inferenz mehrerer Daten, verzichtet wird. Es werden Vergleichstests mit einer Tegra X1 embedded GPU und einem Xilinx Zynq 7045 FPGA durchgeführt. Der FPGA erreicht dabei eine höhere Leistung pro Watt als die GPU bzw. bei gleicher Energieverfügbarkeit eine höhere Performance. Intel stellt in [26] einen Performancevergleich eines Intel Stratix 10 FPGA mit einer Titan X GPU vor. Es wurden dabei aktuelle Trends in der Entwicklung neuronaler Netze betrachtet und explizit auf die Verwendung kleinerer Datentypen und der Berechnung von *Sparse* (dünnbesetzten) Matrizen eingegangen. Der für den Ver-

gleich verwendete Stratix 10 FPGA verfügt über eine hohe Anzahl von DSPs, einen großen on-chip RAM von 28 MB und einen externen HBM (High Bandwidth Memory) Speicher. Diese Ausstattungsmerkmale sind insbesondere bei der Matrixmultiplikation von Vorteil. Untersucht wurde jeweils die Anzahl von Operationen pro Sekunde bei der Multiplikation von Matrizen, sowie die Anzahl der Operation pro Sekunde pro Watt. Der erste Vergleich zeigt die Performance bei der Berechnung von Matrizen klassischer neuronaler Netze mit 32-bit floating point Datentypen. Hier erreichen die Intel FPGAs nicht die theoretische Performance der NVIDIA GPU, allerdings kann der Stratix10 FPGA mehr Berechnungen pro Watt ausführen. Im zweiten Vergleich werden Optimierungen bei der Berechnung dünnbesetzter Matrizen untersucht. Diese Matrizen verfügen über besonders viele Einträge mit Nullen. Es wurden Verfahren verwendet, welche die Matrix eines convolutional Layers auf einen ca. 85 prozentigen Anteil von Nullen bringen. Dabei geht weniger als 1 % Genauigkeit verloren. Die Nullwerte bei der Matrixberechnung werden im FPGA übersprungen. Bei der GPU sind hier keine hohen Optimierungen möglich, da diese als SIMD-Hardware keine unterschiedliche Behandlung der gleichzeitig durchgeführten Berechnungen zulässt. Das Ergebnis des Vergleichs sind mehr Operationen pro Sekunde des Stratix10 FPGAs bei einer Taktung von 500 MHz als bei der Titan X GPU. Ein ähnliches Ergebnis wird bei der Verwendung eines 6-bit Datentypen anstelle der 32-bit Datentypen erreicht. Da die GPU keine beliebig großen Datentypen unterstützt, wird auf dieser ein vorgegebener 8-bit Datentyp verwendet. Ein weiterer Vergleich zeigt eine deutlich höhere Leistung von FPGAs bei der Verwendung von binarisierten neuronalen Netzen, bei denen 1-bit Datentypen verwendet werden. Die Autoren kommen zu dem Schluss, dass unter Berücksichtigung neuer Trends bei neuronalen Netzen (binarisierte Netze, kleine Datentypen, Erzeugung dünnbesetzter Matrizen) FPGAs ein hohes Optimierungspotential gegenüber GPUs besitzen. Ein weiterer Vergleich wurde in [10] durchgeführt. Hier wurde eine FPGA Implementierung für Sliding Window Anwendungen mit Implementierungen für CPUs und GPUs verglichen. Hierbei erreichte der FPGA eine 11× höhere Geschwindigkeit als die GPU und eine 57× höhere Geschwindigkeit als die CPU. Der Energieverbrauch des FPGAs fiel dabei geringer aus und war bei veränderter Kernelgröße nahezu konstant, was den Energiebedarf des FPGAs gut abschätzbar macht.

Es wurden Arbeiten vorgestellt, welche FPGA Implementierungen mit einer hohen Rechenleistung pro verbrauchter Energie zeigen und es wurden Arbeiten vorgestellt, welche es dem Nutzer einfach machen sollen RTL Code für FPGAs zu generieren. Diese Arbeiten zeigen, dass die Möglichkeit besteht leistungsstarke Frameworks für eine einfache Nutzung von FPGA Hardware zu entwickeln.

Kapitel 4

Umsetzung

4.1 DeepRacin

Das Framework DeepRacin [23] wurde am Lehrstuhl für Graphische Systeme an der TU Dortmund entwickelt und dient der Inferenz von tiefen neuronalen Netzen auf ressourcenarmer Hardware.
Es bietet dem Nutzer die Möglichkeit ein Netzwerk zu definieren und zusammen mit den vorab trainierten Gewichtungen und Bias-Werten in das Programm zu laden. Das Training eines Netzwerkes ist mit DeepRacin nicht möglich, sondern muss mit einer anderen Software wie TensorFlow erfolgen. Die Inferenz des in DeepRacin geladenen Netzwerkes, lässt sich anschließend auf mobilen und eingebetteten Systemen ausführen, welche über beschränkte Rechenleistung und Energieressourcen verfügen. Der DeepRacin Code besteht aus drei verschiedenen Teilen. Einem Python-Interface für die Definition eines DeepLearning Netzwerkes, dem Host Code, welcher der Verwaltung der Datenstrukturen und des Programmablaufs dient und OpenCL Kernels, welche optimierten Code für die rechenintensiven Aufgaben enthalten. Das Python-Interface erlaubt dem Nutzer die Definition seines neuronalen Netzes in Python mit einer vergleichbaren Struktur wie in TensorFlow. Die vom Nutzer angelegte Python Datei erzeugt mit Hilfe des Interfaces eine Datei, die den *Computation Graph* (Berechnungsgraphen) enthält und .var Dateien, welche die gelernten Werte enthalten. Der Berechnungsgraph gibt DeepRacin die Struktur des Netzes für die Inferenz vor. Für die Verwendung in DeepRacin können die Gewichtungen mit Hilfe des NumPy-Pakets in einem mehrdimensionalem Array gespeichert werden.
Wenn der Graph geladen wurde, werden Funktionen des Hosts aufgerufen, welche die Ein- und Ausgabewerte für die Inferenz bestimmen, eine OpenCL Umgebung erzeugen und schließlich die Datenverarbeitung starten. Der Host Code ist in der Programmiersprache C geschrieben. Dieser kann unabhängig von einer Python Installation ausgeführt werden, wenn Dateien mit dem Berechnungsgraphen und den Gewichtungen vorhanden sind.
Für verschiedene rechenintensive Operationen sind OpenCL Kernels vorhanden. Diese er-

lauben es Berechnungen auf spezialisierter Hardware zu parallelisieren. Es existieren OpenCL Kernels für folgende Layer: 1x1 Convolution, 2x2 und 3x3 Winograd Convolution, Max- und Avg-Pooling und Fully Connected. Außerdem sind OpenCL Kernels für weitere Berechnungen, wie die Addition, Subtraktion, Multiplikation und Division von Tensoren, sowie Operationen zur Vorverarbeitung von Bildern, wie Upscaling, RGBtoGray und zur Normalisierung vorhanden.

4.1.1 Verwendung

Zuerst erfolgt das Training des Netzwerkes und der Export der Gewichtungen. In TensorFlow lässt sich dies mit Hilfe des NumPy-Pakets durchführen, welches unter Anderem das Speichern und Laden von mehrdimensionalen Arrays ermöglicht. Ein Berechnungsgraph wird in TensorFlow mit der Funktion session.run() ausgeführt. Als Parameter können dabei die Variablen übergeben werden, welche man in einem NumPy Array speichern möchte. Um die Daten in eine Struktur zu bringen, welche einfach in DeepRacin geladen werden kann, sollten die Gewichtungen in einem mehrdimensionalen Array des folgenden Formates gespeichert werden:

```
net[
  layer1[weights[],biases[]],
  layer2[weights[],biases[]],
   ...
]
```

Das Array enthält also für jede Schicht des Netzes ein Array, welches jeweils ein Array für Gewichtungen und ein Array für Bias-Werte enthält. Die entstandene Struktur kann mit Hilfe der Funktion numpy.save(file, array) in eine Datei gespeichert werden. Als nächstes müssen in einem Python Script das Netzwerk definiert und die Gewichtungen aus dem NumPy Array geladen werden. Die NumPy Funktion numpy.load(file) erlaubt es, die Datei mit den gespeicherten Daten einzulesen. Hierzu muss das DeepRacin Interface in das Python Script importiert, und dann ein leerer Graph erstellt werden. Um den Graph zu füllen, müssen die Schichten des neuronalen Netzes definiert werden. Die erste Schicht ist immer eine feed_node, welche die Eingabewerte an das neuronale Netz weitergibt. Darauf können Convolutional, Fully Connected, Pooling oder andere Schichten folgen. Diese erhalten als Parameter die vorangegangene Schicht, die Form (Dimensionen und Größe) der Eingabewerte, die verwendete Aktivierungsfunktion und die Gewichtungen. Optional können Bias-Werte übergeben werden. Convolutional Layer benötigen zusätzlich die Kernelgröße. Wenn alle Layer definiert wurden, werden noch die Output-Layer markiert, der Graph gespeichert und das Setup und Scheduling aufgerufen. Der folgende Python Code ist ein Beispiel für die Definition eines Graphen mit feed_node und einem Fully Connected Layer. Die Eingabewerte sind dabei 2-dimensional im Format $28 \times 28 \times 1$.

```
import deepracin as dr
graph = dr.create_graph()
w = np.load('weights.npy', encoding='latin1').item()
feed = dr.feed_node(graph, shape=(28, 28, 1))
fc = dr.Fully_Connected(feed, w.shape , 'relu', w)
dr.mark_as_output(fc)
dr.save_graph(graph,model_path)
dr.prepare(graph)
```

In einer C-Datei kann nun wahlweise eine neue OpenCL Umgebung generiert werden oder eine Vorhandene verwendet werden. Um eine neue OpenCL Umgebung zu erzeugen, muss die Funktion dR_initCL(net) aufgerufen werden, hierdurch werden die vorhandene Hardware ermittelt und der OpenCL Kontext erstellt. Die Funktion dR_prepare(net) sorgt als nächstes dafür, dass die Kernels kompiliert und die Buffer mit Werten gefüllt werden. Durch den Aufruf von dR_getOutputBuffers(net,outbuffers) wird ein Pointer auf den Ausgabepuffer erzeugt. Zuletzt kann der Aufruf der Inferenz in einer for-Schleife erfolgen. Der folgende Code ist ein Beispiel für das Erzeugen einer neuen OpenCL Umgebung und das Durchführen der Inferenz in DeepRacin:

```
net = dR_NewGraph();
dR_initCL(net);
dR_loadGraph(net,"deepRacinModels/",\&nodeslist,\&numnodes,\&feedlist,\&numfeeds);
dR_setAsOutput(net,nodeslist[numnodes-1]);
dR_prepare(net);
dR_getOutputBuffers(net,outbuffers);
for(int i = 0; i<number;i++)
{
    //Eingabewerten in den Graphen geben
    dR_feedData(net,feedlist[0],(cl_float*)data[i],0,buffersize/sizeof(cl_float));
    //Inferenz durchfuehren
    dR_apply(net);
    //Ausgabewerte auslesen
    dR_downloadArray(net,"",outbuffers[0],0,out_size*sizeof(cl_float),data_out);
}
```

Das Nutzen eines bereits existierenden OpenCL Kontextes und von vorhandenen Puffern kann mit Hilfe der Funktionen dR_setClEnvironment(net, clContext, clPlatformId, clCommandQueue, clDeviceId), dR_setDataFeedNodeBuffer(net,feedlist[0],existingCLMemPointer1) und dR_setPreexistingOutputBuffer(net,nodeslist[numnodes-1],existingCLMemPointer2) erfolgen.

4.1.2 Implementierungsdetails

DeepRacin enthält mehrere Optimierungen, um die Verwendung von Koprozessoren zu beschleunigen. Eine dieser Optimierung ist es, die Ausgabewerte eines Layers, sowie die Gewichtungen und Bias-Werte im Speicher des Compute Devices zu belassen und keine neuen *Speicherpuffer* (cl_mem) für jedes Layer anzulegen. Beim festlegen der Work-Group Größe wird erst die maximal zulässige Größe des Compute Devices ermittelt, damit die Work-Groups so groß wie möglich sind und ein hoher Parallelisierungsgrad auf der Datenebene existiert.
Die Implementierung der Convolution Kernels wurde mit Hilfe des Winograd Minimal Filtering Algorithmus, welcher in [21] vorgestellt wird, vorgenommen. Dieser Algorithmus ermöglicht es die Convolution mit einer geringeren Anzahl von Multiplikationen als bei klassischen Ansätzen durchzuführen. Diese Optimierung sorgt sowohl bei GPUs als auch bei FPGAs für eine Beschleunigung, da Multiplikationen in Hardware sehr aufwändig sind.
Das Verwenden der OpenCL Funktion mad24(x,y,z), welche x und y multipliziert und z hinzuaddiert, ist eine weitere Optimierung der Kernels. Diese Optimierung nutzt eine Operation im Befehlssatz der Floating Point Units von GPUs bzw. der DSPs von FPGAs, um die Multiplikation und Addition in einem einzigen Zyklus auszuführen.
Eine Optimierung, welche besonders bei der Verwendung von FPGA Hardware geeignet ist, ist die Verwendung von Datentypen mit einer geringeren Größe, da diese weniger Speicherplatz belegen und Rechenoperationen auf diesen Datentypen weniger Zeit kosten. Die Recheneinheiten, welche auf dem FPGA platziert werden, nehmen dadurch weniger Platz ein und es können mehr Komponenten auf dem FPGA umgesetzt oder mehr Blöcke als Speicherzellen verwendet werden. OpenCL bietet hierzu den Datentyp *half* an. Im Gegensatz zum *float* Datentyp mit 32-bit verwendet der half Datentyp nur 16-bit. Die Berechnungen werden dadurch etwas unpräziser, da die Zahlen über weniger Stellen verfügen.

4.2 Verwendete Hardware

Das entstandene Programm wurde auf einem *ZedBoard*(Zynq Evaluation and Development Board) getestet. Es ist mit einem Xilinx XC7Z020-1CLG484C ausgestattet, bei dem es sich um ein *System on a Chip* (SoC) handelt, welches mit einer Dual-Core CPU und einem FPGA ausgestattet ist. Der Chip hat folgende Ausstattungsmerkmale:

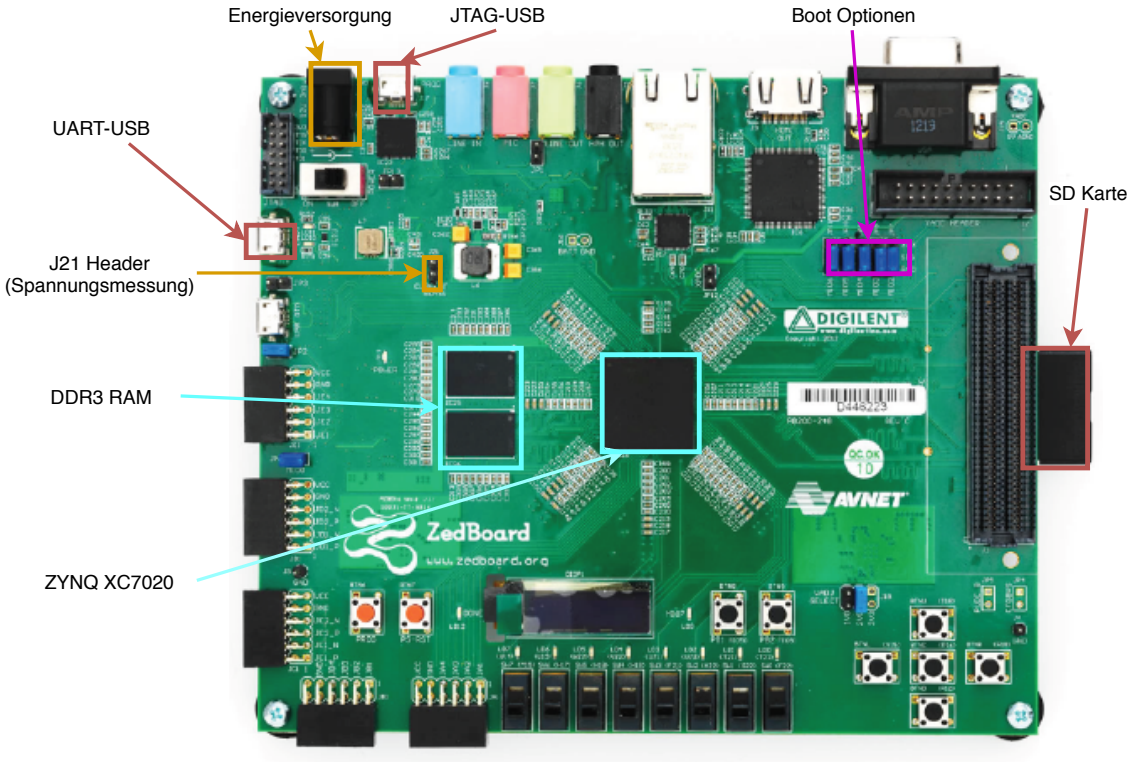


Abbildung 4.1: ZedBoard

CPU	ARM Cortex A9	
	Dual-Core	2x 1 GHz
	Cache	32 KB L1 pro CPU-Core, 512 KB geteilter L2 Cache
	On-Chip RAM	256 KB
FPGA	Xilinx Artix-7	
	Programmable Logic Cells	85.000
	LUTs	53.200
	BRAM	4,9 Mb (140 Blöcke)
	DSP Slices	220

Mit der CPU sind zusätzlich 512 MB DDR3 RAM verbunden. Zugriffe vom FPGA auf den RAM sind nur über die CPU möglich. Dies sorgt dafür, dass die Speicherzugriffe vom FPGA aus langsamer sind als bei einer direkten Verbindung zum RAM. Wenige Zugriffe auf den Arbeitsspeicher können die Geschwindigkeit der FPGA Software daher deutlich erhöhen. Für die Kommunikation besitzt das ZedBoard einen USB-JTAG Anschluss, welcher es ermöglicht, ein Programm über ein USB Kabel von einem Host PC auf den FPGA des Boards zu laden. Des Weiteren verfügt das Board über eine Ethernet Schnittstelle, HDMI und VGA Display Ports und Audio Anschlüsse. Für die Kommunikation zum Host PC ist

eine USB-UART Bridge vorhanden. Über diese können Konsolenbefehle an das ZedBoard gesendet und Ausgaben vom ZedBoard empfangen werden. Am Host PC lassen sich diese Daten mit dem Programm picocom einlesen. Als Speichermedium kann eine SD Karte verwendet werden. Diese kann mit einem Betriebssystem ausgestattet werden, welches dann auf dem Board verwendet werden kann.

4.3 Verwendete Software

Als Entwicklungsumgebung wurde die *Xilinx SDSoC* (Software-Defined System-on-Chip) Environment verwendet. Diese Umgebung ist in der Xilinx SDx Software enthalten und basiert auf der Eclipse IDE und verfügt zusätzlich über die Integration von mehreren Xilinx Tools für die FPGA-Programmierung. SDx ermöglicht die Programmierung von FPGAs in C, C++ und OpenCL und nutzt das Vivado High-Level Synthesis (HLS) Tool, um C/C++/OpenCL Funktionen in programmierbare Logik zu übersetzen. Die integrierten Compiler sind der sdscc Compiler für C-Code, der sds++ Compiler für C++ Code und der xocc Compiler für OpenCL Code. Um ein Projekt mit OpenCL Code anzulegen, muss als System-Konfiguration "OpenCL Linux" ausgewählt werden [34, Seite 22]. OpenCL Programme lassen sich also nur verwenden, wenn ein Linux System auf dem ZedBoard läuft. Für reine C/C++ Programme kann auch ein anderes System oder eine Standalone Variante ohne vollständiges Betriebssystem ausgewählt werden. Dies hat zur Folge, dass OpenCL Programme nicht per USB-JTAG auf den FPGA geladen werden können, sondern nur mit Hilfe der SD Karte.
SDSoC bietet die Möglichkeit auszuwählen, welche Funktionen für die Verwendung auf dem FPGA synthetisiert werden sollen. Im Gegensatz zu C/C++ Programmen ist die Funktionalität des OpenCL Programms abhängig von dieser Auswahl. Die nicht ausgewählten Funktionen werden bei C/C++ Programmen automatisch für die Verwendung mit der ARM CPU kompiliert. Bei OpenCL Funktionen muss der Entwickler selber eine Alternative zu der FPGA Implementierung im Host Code festlegen. Das kann durch das Kompilieren zur Laufzeit von OpenCL Funktionen für die CPU oder durch zusätzliche C/C++ Funktionen erfolgen.

4.4 Offline OpenCL in DeepRacin

Um die Möglichkeit zu geben, die OpenCL Kernels auf einem FPGA auszuführen, müssen Änderungen am Host Code vorgenommen werden. Im Regelfall wird OpenCL Code zur Laufzeit kompiliert, damit vorher die vorhandenen Koprozessoren ermittelt werden können und dann der entsprechende Compiler die OpenCL Kernels in Maschinencode übersetzen kann. Bei der Verwendung von höheren Programmiersprachen für FPGAs muss allerdings eine Synthese stattfinden, welche viele Stunden in Anspruch nehmen kann. Eine Kompi-

lierung zur Laufzeit ist daher ungünstig. Um keine Synthese während der Laufzeit durchzuführen, kann der Kernelcode vorab kompiliert werden und als binary Datei gespeichert werden. Nachdem im Hostcode die Hardware analysiert wurde, kann dann wahlweise Code für CPUs/GPUs kompiliert oder die vorhandenen Binarys mit dem FPGA-Code eingelesen werden. Mit dem Xilinx xocc Compiler lassen sich binary-Dateien im .xclbin Format aus OpenCL Code generieren.
Mit Hilfe der Funktion *clGetPlatformInfo(platform_id, CL_PLATFORM_NAME, ...)* lässt sich der Name einer OpenCL Platform ermittelt. Eine OpenCL Platform entspricht einer Installation von OpenCL eines Hardwareherstellers. Wenn die Funktion als Namen Xilinx liefert, kann das Einlesen des vorab (offline) kompilierten Codes starten. Um aus den .xclbin Dateien ein cl_program Objekt zu erzeugen, muss zunächst die Datei eingelesen werden und in einem char Array abgelegt werden. Das cl_program Objekt erhält man nun als Rückgabewert der Funktion *clCreateProgramWithBinary(...,const unsigned char **binaries,...)*. Die Funktion *clBuildProgram(cl_program program,...)* erzeugt danach aus den binary-Daten ein OpenCL Programm. Falls mehrere Compute Devices vorhanden sind und nur ein bestimmtes genutzt werden soll, kann mit Hilfe der Funktion *clGetDeviceInfo(..., CL_DEVICE_NAME, ...)* innerhalb einer Schleife der Name jedes Devices ausgegeben und durch einen Vergleich der Strings, die ID des gewünschten Devices ermittelt werden. Für die Verwendung des ZedBoards ist die Ausgabe beispielsweise: “zed”. Das erzeugte cl_program Objekt lässt sich nun genauso verwenden, wie ein cl_program, welches zur Laufzeit kompiliert wurde.

4.5 Weitere Anpassungen

Des Weiteren war es notwendig einige Funktionen der glib2.0, welche von DeepRacin verwendet wird, zu ersetzen, da die Library nicht für die Verwendung mit dem ARM Prozessor des ZedBoards kompiliert werden konnte. Die meisten verwendeten Funktionen und Datentypen der glib2.0 unterscheiden sich nur wenig von standard-Funktionen und konnten mit Hilfe von Präprozessoranweisungen ersetzt werden. Unter Anderem zum Lesen von Dateien mussten einige Funktionen nachimplementiert werden. Diese Funktionen können aus der Nachfolgenden Tabelle entnommen werden.

Glib2 Funktionsname	Verwendung in DeepRacin	Vorgehensweise beim Ersatz
g_build_filename	erzeugt einen Dateipfad String aus übergebenen Ordner und Dateinamen. (Für die Pfade zu Graph und Gewichtungen)	Konkatenation übergebener Strings mit "/" als Trennzeichen.
g_get_current_time	Rückgabe der aktuellen Zeit in einem GTimeVal Objekt für die Messung der Dauer einer Kernelberechnung	Systemfunktion time(0) verwendet.
g_file_get_contents	Ausgabe des Inhalts und der Länge einer Datei für das Einlesen von Graph und Gewichtungen	Länge mit fseek() ermitteln und Inhalt mit fread() auslesen.
g_path_is_absolute	Gibt true zurück, falls der übergebene Pfad absolut ist.	Prüfen, ob das erste Zeichen ein "/" ist.
g_get_current_dir	Gibt das aktuelle Verzeichnis zurück, um den den Pfad von benötigten Dateien zu ermitteln.	Wurde konstant auf "/mnt" gesetzt, da sich das Programm auf einem ZedBoard immer in diesem Verzeichnis befindet.

Die Funktionen g_malloc, g_free und g_print konnten per Define durch die standard Funktionen malloc, free und printf ersetzt werden, da diese glib2.0 Funktionen nur in Kombination mit Datentypen wie gint, gchar, gpointer, etc. relevant sind und diese Datentypen durch standard Datentypen ersetzt wurden. Als weitere Optimierung wurde in DeepRacin die Funktion *G_LIKELY(expr)* verwendet. Dieser Funktion wird eine boolsche Bedingung übergeben und dem Compiler wird mitgeteilt, dass diese Bedingung wahrscheinlich true zurückgibt. Vom Compiler wird diese Information zur Optimierung der Branch-Prediction verwendet. Diese Funktion wurde mit folgendem Define ersetzt:

```
#define G_LIKELY(expr) (__builtin_expect ((expr), 1))
```

__builtin_expect(expr, value) teilt dem Compiler mit welcher Wert bei dem übergebenen Ausdruck zu erwarten ist [8, Seite 57]. In dem oben genannten Define wurde dieser Wert auf 1 gesetzt, um der Funktionsweise von *G_LIKELY* zu entsprechen.

Kapitel 5

Ergebnisse

5.1 Datensatz und Netzwerk

Für die Messungen wurde ein Netzwerk verwendet, welches für die Klassifikation von handgeschriebenen Ziffern geeignet ist. Der zugrundeliegende Datensatz ist der mnist-Datensatz [36], welcher graustufen-Bilder von handgeschriebenen Ziffern mit einer Auflösung von 28×28 Pixeln enthält. In dem Datensatz sind 60.000 Trainings- und 10.000 Testbeispiele enthalten. Das getestete Netzwerk verfügt über drei Layer. Das erste Layer hat 140 Fully Connected Neuronen, welche als Eingabe die 784 graustufen-Werte des zu klassifizierenden Bildes erhalten. Das zweite Layer hat 80 Fully Connected Neuronen und das letzte Layer verfügt über 10 Neuronen, welche die jeweilige Wahrscheinlichkeit der Zuordnung zu den 10 Klassen ausgeben. Testdurchläufe in TensorFlow haben für dieses Netzwerk eine Genauigkeit von 98,1% auf den mnist Testdaten ergeben. In den Tests mit DeepRacin wurden die Gewichtungen und Input-Werte mit zufälligen Zahlen belegt, da zum Testzeitpunkt kein Einlesen der Testdateien möglich war. Das Ein- und Auslesen von Daten in Convolutional und Pooling Layer war zum Testzeitpunkt im Hostcode ebenfalls noch nicht möglich, weshalb ein Netzwerk verwendet wurde, welches nur aus Fully Connected Layern besteht.

5.2 Synthetisierter Code

Der OpenCL Kernel für Fully Connected Layer verfügt über zwei Funktionen. Diese Funktionen wurden beide synthetisiert, um auf dem FPGA ausgeführt werden zu können. Es wurde je eine Compute Unit für diese Funktionen auf dem FPGA umgesetzt. Die Belegung von Hardwarekomponenten des ZedBoards durch diese Funktionen kann den nachfolgenden Tabellen entnommen werden.

Funktion:	matrixVectorMultFirst	
Hardwarekomponente	Anzahl belegter Komponenten	Auslastung der vorhandenen Komponenten in %
BRAM_18K Blöcke	45	16%
DSP48E Blöcke	42	19%
Flip Flops	14670	13%
Look Up Tables	12616	23%

Funktion:	matrixVectorMultSecond	
Hardwarekomponente	Anzahl belegter Komponenten	Auslastung der vorhandenen Komponenten in %
BRAM_18K Blöcke	30	10%
DSP48E Blöcke	37	16%
Flip Flops	13169	12%
Look Up Tables	21976	41%

Die Multiplikationen finden in der Funktion *matrixVectorMultFirst* statt, während in der Funktion *matrixVectorMultSecond* Additionen ausgeführt werden. In den Syntheseberichten der beiden Funktionen wird deutlich, dass für die zweite Funktion mehr Hardwarekomponenten erstellt wurden. Hierdurch kommt die größere Anzahl von verwendeten Look-Up-Tables bei dieser Funktion zustande. Auf Datenebene lassen sich die Operationen der *matrixVectorMultSecond* Funktion deutlich besser parallelisieren, weswegen hier viele Addierer verwendet wurden. Die Operationen der Funktion *matrixVectorMultFirst* lassen sich hingegen nicht so gut auf Datenebene parallelisieren bzw. erfordern eine hohe Anzahl von DSP-Slices. Um die Berechnungen zu beschleunigen, könnten also mehr Compute Units für die Funktion *matrixVectorMultFirst* erstellt werden, denn diese ermöglichen eine Task-Parallelisierung. Die Gesamtauslastung der Komponenten ergibt für den BRAM 26%, für die DSP Einheiten 35%, für die Flip Flops 25% und für die Look-Up-Tables 64%. Der FPGA bietet also Platz für die Erstellung mehrerer Compute Units, weitere Optimierungen der Fully Connected Funktionen oder für die Implementierung weiterer Layertypen in Hardware. Eine Compute Unit, welche die 2x2 Winograd Convolution berechnet, konnte zusätzlich auf dem FPGA umgesetzt werden.

5.3 Messmethoden

5.3.1 Zeitmessung

Für die Messung der Dauer einer Inferenz, wird die Funktion *clock()* aus der *time* C-Library verwendet. Diese Funktion gibt die Anzahl von Taktschlägen seit Start des Programms

zurück. Wenn dieser Wert zum Startzeitpunkt und zum Endzeitpunkt der Inferenz gespeichert, die Differenz gebildet und durch die Anzahl von Taktschlägen pro Sekunde geteilt wird, erhält man die Zeit, welche für die Inferenz in Anspruch genommen wurde.

5.3.2 Energiemessung

Um die Leistungsaufnahme der Testsysteme zu messen, wurde ein Brennenstuhl PM 231 Energiemessgerät verwendet, welches die Leistungsaufnahme an einer Steckdose misst. Mit dieser Methode wird die Leistungsaufnahme des gesamten Systems und nicht nur der relevanten Komponenten gemessen. Außerdem wird auch die durch Hintergrundprozesse verursachte Leistungsaufnahme betrachtet.
Eine genauere Methode zur Messung des Energieverbrauchs eines ZedBoards ist die Spannungsmessung am J21 Header. Hier befindet sich ein $R_{J21} = 10m\Omega$ Widerstand, welcher in Reihe mit der 12V Stromversorgung des ZedBoards geschaltet ist [7]. Bei einer gemessenen Spannung u_{J21} lässt sich die Leistung des ZedBoards mit der Formel $P = \frac{u^2}{10m\Omega}$ berechnen. Der Energieverbrauch ergibt sich durch die Integration der Leistung über die Zeit: $E = \int P\,dt$.
Die Messung kann mittels eines INA219 Spannungsmessers, welcher die gemessenen Daten per I2C an ein Raspberry Pi sendet, erfolgen. Am Raspberry Pi werden die gemessenen Daten verwendet, um die Leistung zu berechnen. Außerdem wird die Zeit gemessen, um den Gesamtverbrauch ausrechnen zu können. Das Board, auf dem der INA219 platziert ist, verfügt über einen $100m\Omega$ Widerstand zwischen Vin- und Vin+. Dieser Widerstand befindet sich parallel zu dem Widerstand zwischen den Pins des J21 Headers. Der Gesamtwiderstand zwischen des Pins beträgt damit: $R_{ges} = \frac{R_{J21} \cdot R_{100}}{R_{J21} + R_{100}}$. Für die Werte der Widerstände ergibt sich: $\frac{0{,}1\Omega * 0{,}01\Omega}{0{,}11\Omega} \approx 0{,}01\Omega$. Näherungsweise ergibt sich ein Gesamtwiderstand von $10m\Omega$. Dieser Wert lässt sich für die Berechnung der Leistungsaufnahme verwenden.

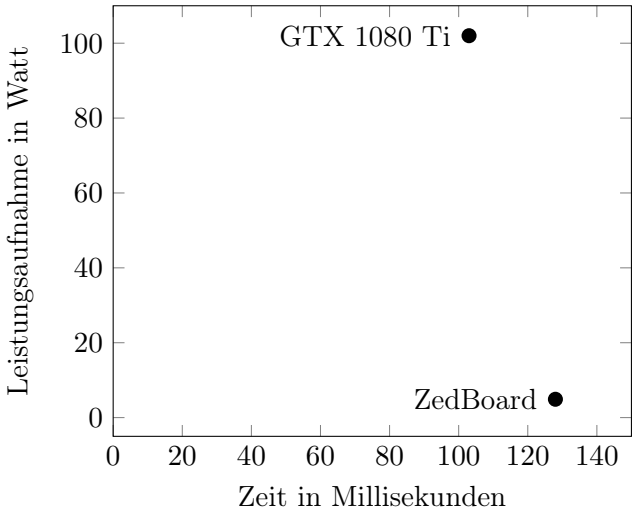
5.4 Vergleich mit anderer Hardware

Neben den Tests auf dem ZedBoard wurde die Inferenz auf einem System mit folgenden Eigenschaften getestet:

CPU	Intel Core i7-7700k mit 4x 4,0 GHz
RAM	32 GB DDR4
GPU	NVIDIA GTX 1080 Ti mit 1594 MHz
Grafikspeicher	11 GB GDDR5X
Betriebssystem	Ubuntu 18.04

Das System mit der NVIDIA Geforce GTX 1080 Ti hat im Durchschnitt für eine Inferenz 103,35 Millisekunden benötigt, während das ZedBoard 128,25 Millisekunden brauchte. Die

NVIDIA GPU war also um 24,09% schneller. Bei der Leistungsaufnahme ergab sich mit 4,9 Watt beim ZedBoard zu 102 Watt bei dem System mit GPU ein deutlich größerer Unterschied. Das GPU-System hat damit eine um den Faktor 20 größere Leistungsaufnahme als das ZedBoard.



5.5 Fazit und Ausblick

Für eine vollständige Implementierung aller Funktionen von DeepRacin für die Verwendung auf FPGAs sind umfangreiche Änderungen am Host Code notwendig. Hierzu gehört der Ersatz der glib2.0 Funktionen und die Möglichkeit vorab kompilierte Kernels in das Programm zu laden. Außerdem ist eine Alternative zu der Ausführung der OpenCL Kernels auf dem FPGA nötig, damit alle Funktionen, die nicht auf den FPGA passen, von einer CPU ausgeführt werden können.
Neben den in dieser Arbeit durchgeführten Tests, können weitere Vergleiche mit anderen Frameworks und anderer Hardware (beispielsweise einem Raspberry Pi) Aufschluss über die Eignung von DeepRacin für den Einsatz in eingebetteten Systemen geben. Hierzu ist auch eine genauere Messung des Energieverbrauchs mit dem in Kapitel 5.3.2 vorgestellten Verfahren besser geeignet, als die durchgeführte Messung.
Um die Verwendung auf anderen FPGAs als dem Zynq des ZedBoards zu ermöglichen, kann ein Makefile für die Kompilierung mit dem Xilinx xocc Compiler bereitgestellt werden. Damit können binary-Dateien erstellt werden, welche auf den entsprechenden Xilinx FPGAs zum Einsatz kommen können. Das Kapitel 3 hat gezeigt, dass es verschiedene Ansätze gibt, um DeepLearning Frameworks mit FPGA Unterstützung zu entwickeln. Ein Ansatz, welcher die Erzeugung von RTL Code für FPGAs zur Laufzeit möglich machen kann, wird in [15] vorgestellt. Dies könnte die Vorteile der Laufzeitkompilierung von OpenCL Kernels für FPGAs verfügbar machen und die Entwicklung weiterer Frameworks fördern.
Die Anpassung eines ganzen Frameworks für FPGAs ist aufwendig, da durch die notwendi-

ge Synthese eine andere Vorgehensweise als bei der Verwendung von OpenCL Kernels für CPUs oder GPUs erforderlich ist. Die Verwendung von OpenCL ist allerdings durch vorhandene Hardwaresynthesetools, die Unterstützung weiterer Hardware und die klare Trennung von parallelisierten und nicht-parallelisierten Funktionen gut für FGPA Programme geeignet. Viele Arbeiten zeigen, dass durch die Verwendung von FPGAs für DeepLearning eine hohe Geschwindigkeit bei der Inferenz und eine geringe Leistungsaufnahme erreicht werden können. Es ist daher ein hohes Leistungspotential für ein solches Framework vorhanden.

Abbildungsverzeichnis

Literaturverzeichnis

[1] AMD: *Radeon Instinct^{TM} MI25 Accelerator*. https://www.amd.com/de/products/professional-graphics/instinct-mi25. Accessed: 06-08-2018.

[2] Apple: *Informationen zur fortschrittlichen Technologie von Face ID*. https://support.apple.com/de-de/HT208108. Accessed: 05-01-2018.

[3] BERTEN: *GPU vs FPGA Performance Comparison*. http://www.bertendsp.com/pdf/whitepaper/BWP001_GPU_vs_FPGA_Performance_Comparison_v1.0.pdf, 2016.

[4] Buschjäger, Sebastian: *Vorlesungsfolien Fachprojekt: DeepLearning on FPGAs*. http://www-ai.cs.uni-dortmund.de/LEHRE/FACHPROJEKT/WS1617/, 2016.

[5] Daniel H. Noronha, Bahar Salehpour, Steven J.E. Wilton: *LeFlow: Enabling Flexible FPGA High-Level Synthesis of Tensorflow Deep Neural Networks*. To be published in FPGA for Software Programmers (FSP 2018), 2018.

[6] DiCecco, Roberto, Griffin Lacey, Jasmina Vasiljevic, Paul Chow, Graham W. Taylor und Shawki Areibi: *Caffeinated FPGAs: FPGA Framework For Convolutional Neural Networks*. CoRR, abs/1609.09671, 2016.

[7] Digilent: *ZedBoard (ZynqTM Evaluation and Development) Hardware User's Guide*, 2014.

[8] Drepper, Ulrich: *What Every Programmer Should Know About Memory*, 2007.

[9] Duarte, Javier, Song Han, Philip Harris, Sergo Jindariani, Edward Kreiner, Benjamin Kreis, Jennifer Ngadiuba, Maurizio Pierini, Ryan Rivera, Nhan Tran und Thenbin Wu: *Fast inference of deep neural networks in FPGAs for particle physics*. JINST (submitted), 2018.

[10] Fowers, Jeremy, Greg Brown, Patrick Cooke und Greg Stitt: *A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding-window Applications*. Seiten 47–56, 2012.

[11] Google: *TensorFlow*. https://www.tensorflow.org. Accessed: 25-08-2018.

[12] Hennessy, John L. und David A. Patterson: *Computer Architecture A Quantitative Approach Fifth Edition*. Morgan Kaufmann, 2012.

[13] Hofmann, Johannes, Jan Treibig, Georg Hager und Gerhard Wellein: *Comparing the Performance of Different x86 SIMD Instruction Sets for a Medical Imaging Application on Modern Multi- and Manycore Chips*. CoRR, abs/1401.7494, 2014.

[14] Intel: *Intel® FPGA SDK For OpenCL™*. `https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html`. Accessed: 25-08-2018.

[15] Jain, Abhishek Kumar, Douglas L. Maskell und Suhaib A. Fahmy: *Resource-Aware Just-in-Time OpenCL Compiler for Coarse-Grained FPGA Overlays*. CoRR, abs/1705.02730, 2017.

[16] Jennessy, John und David Patterson: *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 2011.

[17] Jonathan Tompson, Kristofer Schlachter: *An Introduction to the OpenCL Programming Model*, 2012.

[18] Jouppi, Norman P.: *In-Datacenter Performance Analysis of a Tensor Processing Unit*. CoRR, abs/1704.04760, 2017.

[19] Kruse, R., C. Borgelt, C. Braune, F. Klawonn, C. Moewes und M. Steinbrecher: *Computational Intelligence*. Springer, 2015.

[20] Lacey, Griffin, Graham W. Taylor und Shawki Areibi: *Deep Learning on FPGAs: Past, Present, and Future*. CoRR, abs/1602.04283, 2016.

[21] Lavin, Andrew und Scott Gray: *Fast Algorithms for Convolutional Neural Networks*. CoRR, abs/1509.09308, 2015.

[22] LeCun, Yann, Yoshua Bengio und Geoffrey Hinton: *Deep learning*. Nature, 521(7553):436 – 444, 2015.

[23] Lenssen, Jan Eric: *deepRacin GitHub*. `https://github.com/mrjel/deepracin`.

[24] Marwedel, Peter: *Embedded System Design*. Springer, 2011.

[25] N. Yadav, A. Yadav und M. Kumar: *An Introduction to Neural Network Mehods for Differential Equations*. Springer, 2015.

[26] Nurvitadhi, Eriko, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Gee Hock Ong, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra und Guy Boudoukh: *Can FPGAs Beat GPUs in Accelerating Next Gen Deep Neural Networks?* FPGA '17. ACM, 2017.

[27] NVIDIA: *NVIDIA TITAN V.* `https://www.nvidia.com/en-us/titan/titan-v/`. Accessed: 06-08-2018.

[28] Owenes, John D., Mike Housten, Luebke, John E. David, Stone, Simon Green und James C. Phillips: *GPU Computing.* Proceedings of the IEEE, Vol. 96, 2008.

[29] Qiu, Jiantao, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang und Huazhong Yang: *Going Deeper with Embedded FPGA Platform for Convolutional Neural Network.* Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seiten 26–35, 2016.

[30] Schmidhuber, Jürgen: *Deep Learning in Neural Networks: An Overview.* CoRR, abs/1404.7828, 2014.

[31] Shalev-Schwartz, Shai und Shai Ben-David: *Understanding Machine Learning From Theory to Algorithms.* Cambridge University Press, 2014.

[32] Teubner, Jens: *Vorlesungsfolien Computer Architecture.* `https://ls12-www.cs.tu-dortmund.de/daes/media/documents/teaching/courses/ss16/ra/ra-070809.pdf`, 2016.

[33] Venieris, Stylianos I. und Christos-Savvas Bouganis: *fpgaConvNet: A Toolflow for Mapping Diverse Convolutional Neural Networks on Embedded FPGAs.* CoRR, abs/1711.08740, 2017.

[34] Xilinx: *SDSoC Environment User Guide.* `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1027-sdsoc-user-guide.pdf`, 2018.

[35] Xilinx: *SDx Command and Utility Reference Guide.* `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1279-sdx-command-utility-reference-guide.pdf`, 2018.

[36] Yann LeCun, Corinna Cortes, Christopher J.C. Burges: *The mnist database of handwritten digits.* `http://yann.lecun.com/exdb/mnist/`. Accessed: 24-08-2018.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 27. August 2018

Andreas Bühner