

Implementierung und Analyse von CNN-Komponenten in VHDL

Fabian Rieger

20. November 2025

Zusammenfassung

Diese Arbeit untersucht die blockweise Implementierung von CNN-Komponenten in VHDL. Die einzelnen Funktionsblöcke – wie Faltung, Aktivierungsfunktionen und Pooling – werden separat entwickelt, analysiert und anschließend zu einer Gesamteinheit (Entity) zusammengeführt. Neben der Funktionsbeschreibung der Module werden die durch die Implementierung entstehenden Grenzwerte hinsichtlich Rechenleistung, Speicherbedarf und Datenbreite aufgezeigt. Abschließend werden mögliche Optimierungspotenziale sowie der technische Aufwand für eine vollständige Hardwareimplementierung diskutiert.

1 Einleitung

Convolutional Neural Networks (CNNs) haben sich in den letzten Jahren als leistungsfähige Methode für zahlreiche Anwendungen der Bild- und Signalverarbeitung etabliert, darunter Objekterkennung, medizinische Bildanalyse und autonome Systeme. In vielen Szenarien ist jedoch nicht nur eine hohe Erkennungsgenauigkeit, sondern auch eine schnelle und energieeffiziente Verarbeitung erforderlich – insbesondere in eingebetteten Systemen mit begrenzten Ressourcen. Eine Möglichkeit, diese Anforderungen zu erfüllen, besteht in der hardwarebasierten Implementierung von CNNs, beispielsweise auf FPGAs unter Verwendung von VHDL. Dadurch können Berechnungen parallelisiert, Latenzzeiten reduziert und der Energieverbrauch gesenkt werden.

2 Grundlagen der Faltung (Convolution)

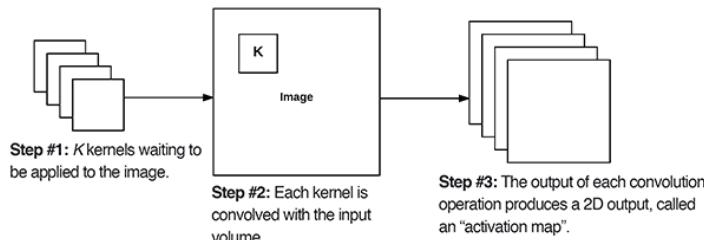


Abbildung 1: Veranschaulichung des Faltungsprozesses in einer Convolutional Neural Network (CNN)-Schicht.

Funktionsweise der Convolution: Eine Convolution (Faltung) in einem CNN ist ein lineares Filterverfahren, bei dem ein Eingabebild der Dimension (Width \times Height \times Channels) mit einem oder mehreren *Kernen* (*Kernels* oder *Filters*) verarbeitet wird. In unserem Beispiel beträgt die Kanalanzahl 3 (RGB). Jeder Kernel hat die Abmessungen ($K_{\text{size}} \times K_{\text{size}} \times \text{Channels}$), wobei hier quadratische Filter verwendet werden.

Der Kernel wird mit einer definierten Schrittweite (*stride*) über das Eingabebild verschoben. Für jede Position wird das Frobenius-Skalarprodukt zwischen Kernel und dem überdeckten Bildausschnitt berechnet. Das Ergebnis bildet einen Wert in der sogenannten *Feature Map* (Aktivierungskarte).

Zusätzlich kann eine *Padding*-Strategie eingesetzt werden: *Valid Padding* ($P = 0$, keine Auffüllung) verkleinert die Ausgabe, während *Zero Padding* ($P > 0$, Auffüllen mit Nullen) die räumliche Größe erhält.

Die allgemeine Formel für die Ausgabegröße lautet:

$$\text{Output_Width} = \left\lceil \frac{\text{Input_Width} - K_{\text{size}} + 2P}{\text{Stride}} \right\rceil + 1$$

$$\text{Output_Height} = \left\lceil \frac{\text{Input_Height} - K_{\text{size}} + 2P}{\text{Stride}} \right\rceil + 1$$

Werden K verschiedene Filter verwendet, entstehen K Aktivierungskarten, die als Stapel (Output-Volume) vorliegen und unterschiedliche Merkmalsaspekte wie Kanten, Texturen oder komplexere Muster erfassen.

3 Gesamtarchitektur der Inferenz-Pipeline

Abbildung 2 zeigt die gesamte VHDL-Architektur auf Systemebene, die als Inferenz-Pipeline für ein vollständiges CNN konzipiert ist. Der Datenfluss ist seriell und modular aufgebaut, wobei jede Sektion eine spezifische Operation des Neuronalen Netzes implementiert.

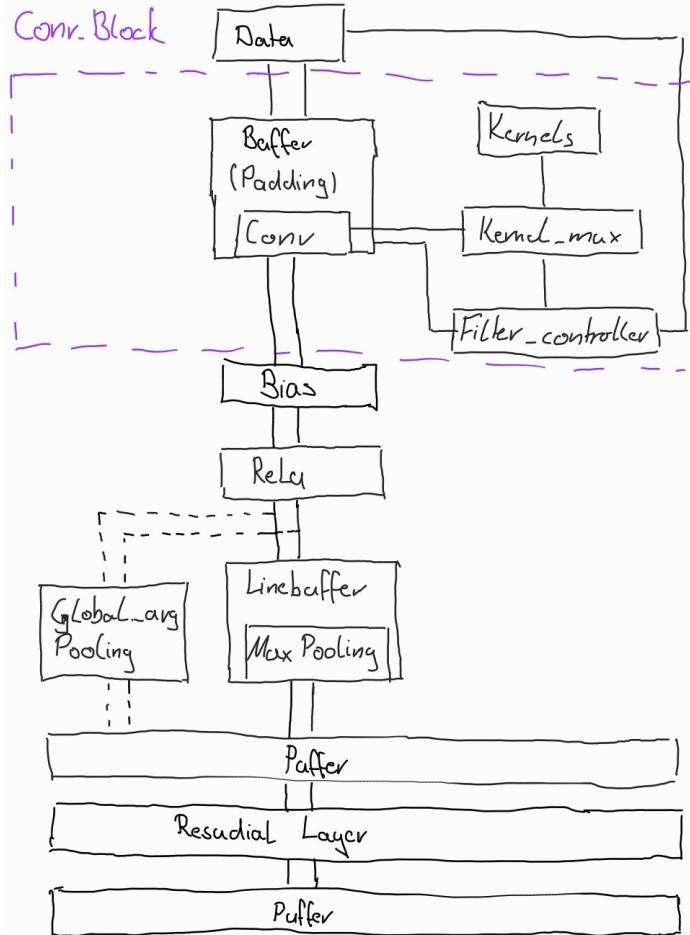


Abbildung 2: Blockschaltbild der gesamten CNN-Inferenz-Pipeline.

Die Pipeline verarbeitet die Daten in der folgenden Reihenfolge:

- 1. Convolution Block:** Die Eingangsdaten (Data) werden zunächst in einen Buffer geladen, der auch das 'Padding' implementiert. Der Conv-Kern verarbeitet die gepufferten Daten. Ein

Filter_controller steuert, welcher Kernel aus dem **Kernels-Speicher** (via **Kernel_max**) an den Conv-Kern übergeben wird. Dies ist aufgrund der vielen Multiplikationen der rechenintensivste Teil der Pipeline.

2. **Bias-Addition:** Das Ergebnis des Faltungsblocks wird an ein Modul weitergeleitet, das einen Bias-Wert aufaddiert.
3. **Aktivierungsfunktion (ReLU):** Das Ergebnis der Bias-Addition durchläuft eine 'Rectified Linear Unit' (ReLU)-Funktion. Diese nicht-lineare Operation setzt alle negativen Werte auf null, was für das Lernen von komplexen Mustern essentiell ist.
4. **Pooling-Schicht:** Nach der Aktivierung durchläuft die Feature Map eine Downsampling-Operation. Die Architektur ist flexibel ausgelegt, um entweder **Global_avg_Pooling** (Reduktion der gesamten Map auf einen Wert) oder **Max_Pooling** (das ein $n \times n$ -Fenster über einen Linebuffer erfordert) zu verwenden.
5. **Residual Layer (optional):** Die Pipeline unterstützt Residual Connections". Ein Puffer (**Puffer**) hält die Daten vor, um sie mit dem Ausgang einer tieferliegenden Schicht zu addieren (z.B. in einem ResNet-Block).

Jede dieser Komponenten wird als separate, streaming-fähige VHDL-Entität implementiert, um einen kontinuierlichen Datenfluss und hohen Durchsatz zu gewährleisten.

4 Gesamtarchitektur des Convolution Blocks

Das vorliegende Blockschaltbild zeigt die Hardware-Architektur eines in VHDL implementierten Convolution-Blocks (bezeichnet als Conv-Layer). Diese Architektur ist darauf ausgelegt, Faltungsoperationen effizient durchzuführen.

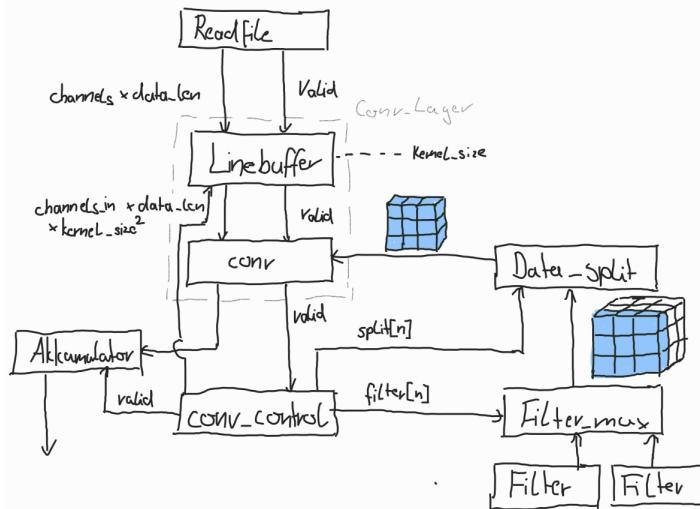


Abbildung 3: Veranschaulichung des Prozesses im Convolution Block.

Die zentralen Komponenten dieser Implementierung sind:

- Ein **Read File**-Modul für das Einlesen der Eingangsdaten.
- Ein **Linebuffer** zur Zwischenspeicherung der Daten, um ein Fenster passender **Kernel_size** für die Faltung bereitzustellen.
- Die eigentliche Faltungseinheit (**conv**), welche die Multiplikations- und Additionsoperationen durchführt.
- Eine Steuereinheit (**conv_control**), welche den Datenfluss und die Synchronisation mittels **Valid**-Signalen sowie die Filterauswahl (**filter[n]**) und Datenaufteilung (**split[n]**) managt.

- Ein **Akkumulator**, der die Ergebnisse der **conv**-Einheit aufsummiert.
- Module zur Filter- und Datenverwaltung (**Data_Split**, **Filter_max** und **Filter**).

4.1 Signalfluss

Als primäres Eingangssignal empfängt das **Read File**-Modul die Daten (definiert als `channels × data_len`) und leitet diese, gesteuert durch ein **Valid**-Signal, an den **Linebuffer** weiter. Der Buffer stellt der **conv**-Einheit die Daten in der korrekten Fenstergröße (`channels_in × kernel_size2`) zur Verfügung.

Die **conv**-Einheit verarbeitet diese Daten und sendet ihre Teilergebnisse an den **Akkumulator**. Nachdem der **Akkumulator** alle notwendigen Werte für ein Ausgabeelement aufsummiert hat – gesteuert durch die **conv_control**-Logik – wird ein finaler Ausgabewert bereitgestellt. Dieser wird durch den nach unten führenden Pfeil am **Akkumulator** symbolisiert. Die Daten könnten von dort an durch einen weiteren Layer wie Max Pooling laufen um dann wieder in einen Puffer landen.

5 Modulare Komponenten der Architektur

Die folgenden Abschnitte beschreiben die Implementierungsdetails der zentralen VHDL-Module, aus denen sich die Gesamtarchitektur zusammensetzt.

5.1 Aufbau und Schnittstelle (Entity Conv-Layer)

Der Convolution Layer (im Folgenden als **Conv-Layer** bezeichnet) ist die Top-Level-Entität der Faltungsarchitektur und instanziert die primären Verarbeitungsböcke:

- **Puffer:** Ein spezialisierter Puffer, der die seriell eintreffenden Eingangsdaten (Pixel) zwischen-speichert. Seine Hauptfunktion ist es, ein dreidimensionales Datenfenster (Window) in der Größe des Faltungskernels (Kernel) für die parallele Verarbeitung bereitzustellen.
- **conv_nxn (Convolution Core):** Diese Einheit empfängt das Datenfenster vom Linebuffer sowie den zugehörigen Kernel und führt die eigentliche Faltungsoperation (Multiplikation und Akkumulation) durch.

5.1.1 Konfigurationsparameter (Generics)

Der **Conv-Layer** ist durch die folgenden Parameter generisch konfigurierbar, um ihn an unterschiedliche Datenformate und Architekturanforderungen anzupassen.

Dimensionen und Struktur:

- **WIDTH:** Breite des Eingangsbildes/der Feature-Map.
- **HEIGHT:** Höhe des Eingangsbildes/der Feature-Map.
- **KERNEL_SIZE:** Kantenlänge des (quadratischen) Faltungskernels.
- **STRIDE:** Schrittweite (Stride) der Faltungsoperation.
- **CHANNELS:** Anzahl der Eingangskanäle (z.B. 3 für RGB).

Datenformat Eingangsdaten (Fixed-Point):

- **TOTAL_BITS_IN:** Gesamtanzahl der Bits für Eingangsdaten.
- **INT_BITS_IN:** Anzahl der Vorkommastellen (Integer-Anteil).
- **FRAC_BITS_IN:** Anzahl der Nachkommastellen (Fractional-Anteil).

Datenformat Filtergewichte (Fixed-Point):

- TOTAL_BITS_WEIGHT: Gesamtanzahl der Bits für Kernel-Gewichte.
- INT_BITS_WEIGHT: Anzahl der Vorkommastellen.
- FRAC_BITS_WEIGHT: Anzahl der Nachkommastellen.

5.1.2 Externe Schnittstelle (Ports)

Die Schnittstelle des Conv-Layer-Moduls ist wie folgt definiert:

Tabelle 1: Eingangssignale des Conv-Layers

Signal	Format	Beschreibung
clk	1 Bit	Systemtakt für alle synchronen Operationen.
rst	1 Bit	Globales Reset-Signal (typischerweise aktiv-high).
pixel_in	CHANNELS * TOTAL_BITS_IN	Paralleler Eingangsdaten-Port (Pixelwerte aller Kanäle).
valid_in_kernel	1 Bit	Steuersignal; zeigt an, dass ein gültiger Kernel am kernel_in-Port anliegt.
kernel_in	(KS ² * CH) * TOT_BITS_WEIGHT	Der Faltungskernel, als flacher Vektor serialisiert.
ctrl_in	1 Bit	Trigger-Signal, um die Fenster-Weitergabe vom Linebuffer an den Core zu starten.
row_ptr	(dynamisch)	Adresspointer (Zeile) für die linke obere Ecke des Fensters.
col_ptr	(dynamisch)	Adresspointer (Spalte) für die linke obere Ecke des Fensters.

Tabelle 2: Ausgangssignale des Conv-Layers

Signal	Format	Beschreibung
pixel_out	(Berechnet)*	Das finale berechnete Ausgangspixel (Ergebnis der Akkumulation).
valid_linebuffer	1 Bit	Signalisiert, dass der Ausgang des Linebuffers (window_out) gültig ist.

* Format von pixel_out: $\text{ceil}(\log_2(\text{real}(2^{*(\text{INT_BITS_WEIGHT}*2)} * \text{KERNEL_SIZE}^2 * \text{CHANNELS}))) + \text{FRAC_BITS_WEIGHT} + 1$

5.1.3 Interne Signale (Verbindungen)

Die folgende Tabelle listet die wesentlichen internen Signale auf, welche die Hauptkomponenten (Linebuffer und Convolution Core) verbinden.

Tabelle 3: Interne Signale des Conv-Layers

Signal	Format	Beschreibung
window_out	(KS ² * CH) * TOTAL_BITS_IN	Internes Signal; das vom Linebuffer bereitgestellte, vollständige Datenfenster. Dient als Eingang für den conv_nxn-Block.
valid_window	1 Bit	Gültigkeitssignal für das window_out. (Entspricht funktional dem valid_linebuffer-Ausgang).

5.2 Umsetzung der Convolution (conv_nxn)

Der `conv_nxn` Block ist die Hardware-Implementierung einer $n \times n$ Convolution. Als Eingabe erhält der Block:

- Einen Kernel der Größe `kernel_size × kernel_size`
- Einen entsprechende Bildausschnitte gleicher Größe (`Channels`)

Die Architektur berechnet automatisch die erforderlichen Registergrößen anhand der Parameter. Die Parameter werden von der `Conv-Layer`-Entität geerbt.

5.2.1 Architektur

In Abbildung 4 ist der Signalfluss und die Berechnungsschritte des `conv_nxn` Blocks dargestellt. Die Darstellung beinhaltet die Festkommaformate in der Form:

Vorzeichen.Vorkommastellen.Nachkommastellen
SIGN_BITS.INT_BITS.FRAC_BITS

5.2.2 Berechnung der Bitbreiten

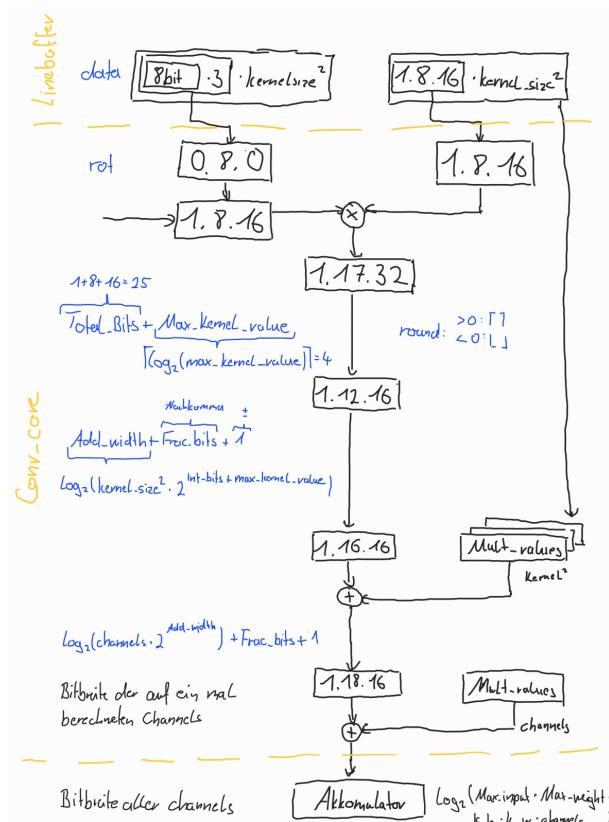


Abbildung 4: Veranschaulichung der Umsetzung auf Hardware

fenster dar. Sobald ein gültiges Fenster vorliegt, wird dieses in RGB Werte getrennt, für diese werden

Die Bitbreiten ergeben sich aus:

$$\text{TOTAL_BITS} = 1 + \text{INT_BITS} + \text{FRAC_BITS}$$

Zusätzlich muss berücksichtigt werden, dass nach der Multiplikation mit einem Kernel Wert die benötigte Bitbreite steigt (auf `MULT_WIDTH`), diese kann wie folgt berechnet werden:

$$\text{MULT_WIDTH} = \text{TOTAL_BITS} + \lceil \log_2(\text{MAX_KERNEL_VALUE}) \rceil$$

und für die Aufsummation der `Kernel_size * Kernel_size` Multiplikationswerte, muss die Bitbreite (`ADDITION_WIDTH`) wie folgt erweitert werden:

$$\begin{aligned} & \log_2(\text{kernel_size}^2 \cdot \\ & 2^{\text{INT_BITS} + \text{MAX_KERNEL_VALUE}}) \\ & + \text{FRAC_BITS} + 1 \end{aligned}$$

Durch dieses Design lässt sich der Ressourcenverbrauch einstellen. Da man durch die Variable 'Channels' X Channels gleichzeitig verarbeiten kann. Durch den danch geschaltenen Akkumulator lassen sich dann die gesamten Channelblocks zu einem Endergebnis addieren.

5.2.3 Funktionaler Ablauf

Abbildung 5 zeigt den zeitlichen Ablauf der Signale innerhalb des `conv_nxn` Blocks. Das Signal `Window_in` stellt das Eingabedaten-

die zugehörigen Multiplikationswerte (`Mult_value`) berechnet und in die nächste Verarbeitungsstufe weitergegeben. Daraufhin erfolgt die Akkumulation der Ergebnisse, die im Signal `Sum_out` sichtbar ist. Am Ende des Verarbeitungzyklus signalisiert `valid_out`, dass das Ergebnis vollständig und gültig vorliegt. Die versetzte Abfolge der Signale verdeutlicht die Latenz innerhalb der Pipeline sowie die Überlappung der Verarbeitungsschritte, wodurch eine kontinuierliche Verarbeitung mehrerer Fenster ermöglicht wird.

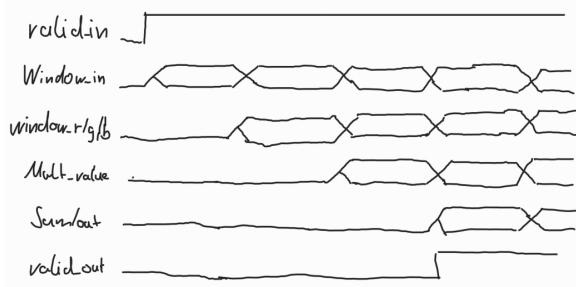


Abbildung 5: Zeitlicher Ablauf der internen Signale im `conv_nxn` Block

5.3 Umsetzung des Puffers

Der **Puffer** ist eine Hardware-Implementierung zur Zwischenspeicherung von Eingangsdaten, um fortlaufend lokale Bildausschnitte für nachfolgende Verarbeitungsschritte bereitzustellen.

Als Eingabe erhält der Block:

- fortlaufende Eingangs Werte des Eingangssignals im eingestellten Format.
- synchronisierte Steuersignale (`valid_in`, `vailid_out` und `ctrl_in`)
- Positionsauswahl (`row_ptr` und `col_ptr`)

Als Ausgabe liefert der **Puffer** einen vollständigen Bildausschnitt der Größe `kernel_size2 * Channels`, bestehend aus allen Pixelwerten, die für eine $n \times n$ Convolution benötigt werden. Die Architektur ist so ausgelegt, dass der Ausgabebereich bei jedem Takt um ein Stride verschoben wird, sodass die nachfolgende Verarbeitung (z. B. durch den `conv_nxn`-Block) kontinuierlich mit den aktuellen Bildausschnitten versorgt wird. Der **Puffer** erbt seine Parameter ebenfalls aus dem Convolutional Layer.

5.3.1 Architektur

Der vorgestellte Aufbau realisiert die Extraktion eines `KERNEL_SIZE × KERNEL_SIZE × Channels`-Pixelfensters aus einem vollständigen Bildspeicher unter Anwendung von 'Same Padding'. Eingehende Bilddaten werden Pixel für Pixel über einen Eingabestrom (`pixel_in`, `valid_in`) empfangen und in einem vollständigen Framebuffer, der typischerweise als Block-RAM (BRAM) implementiert wird, zwischengespeichert. Diese Architektur ermöglicht einen wahlfreien Zugriff auf beliebige Bildausschnitte, anstatt einen festen gleitenden Strom zu erzeugen. Über externe Steuersignale (`ctrl_in`, `row_ptr`, `col_ptr`) wird der Mittelpunkt des gewünschten Fensters spezifiziert. Das Modul berechnet die absoluten Leseadressen für alle Pixel des Fensters relativ zu diesem Mittelpunkt. Eine integrierte Logik prüft, ob die berechneten Koordinaten innerhalb der Bildgrenzen (`IMAGE_WIDTH`, `IMAGE_HEIGHT`) liegen; andernfalls wird der entsprechende Pixelwert im Fenster mit Nullen aufgefüllt ('Zero-Padding'). Das vollständige Fenster wird als flacher Vektor (`window_out`) zusammen mit einem Gültigkeitssignal (`valid_out`) ausgegeben. Diese Struktur eignet sich besonders für Faltungsoperationen (Convolution), die einen flexiblen oder nicht-sequenziellen Zugriff auf Bilddaten benötigen. Es wurde zusätzlich über die Implementierung eines Linebuffers nachgedacht um geringeren BRAM speicher zu benötigen. Es

wurde sich dagegen entschieden, da man bei dieser Variante nicht gleichzeitig für die nächste Convolution auffüllen kann, während man eine andere Ausgibt. Dieser würde jedes mal zu einer Eingangslatenz führen. Jedoch kann man für die Zukunft durch geschickte Implementierung den Speicherbedarf an dieser Stelle reduzieren.

5.3.2 Performance

Zu Beginn entsteht eine Anlaufverzögerung, weil zunächst $(k - 1)$ komplette Zeilen plus $(k - 1)$ weitere Pixel eingelesen werden müssen, bevor das erste vollständige Fenster vorliegt. Die Startlatenz in Takten (clk cycles) lautet daher.

$$L_{\text{init}} = (k - 1) \cdot w + (k - 1) + 1$$

Die plus 1 ist dabei die Latenz für die Verarbeitung im Layer: Für einen Kernel der Größe $k = 3$ und eine Bildbreite von $w = 640$ ergibt sich:

$$L_{\text{init}} = (3 - 1) \cdot 640 + (3 - 1) + 1 = \mathbf{1283} \text{ Takte}$$

Beim Zeilenwechsel fällt zusätzlich eine Latenz an, bis das Fenster horizontal wieder vollständig ist, wenn man **kein** Same Padding verwendet:

$$L_{\text{row}} = k - 1$$

Für den 3×3 -Kernel gilt damit:

$$L_{\text{row}} = 3 - 1 = \mathbf{2} \text{ Takte}$$

Nach Ablauf der Initialisierung arbeitet die Pipeline mit einem Durchsatz von **1** Pixel pro Takt, sofern das Eingangssignal valid_{in} kontinuierlich anliegt.

Die gesamte Latenz zur Verarbeitung eines kompletten Bildes beträgt im Prinzip 1 Takt, es wird zwar später mit dem Einlesen begonnen, da aber nicht so die selbe Anzahl an Pixeln ausgegeben werden muss, ist nur der eine Verarbeitungstakt relevant. Somit ist der Output 1 Takt nach dem Einlesen des letzten Bits des Bildes fertig.

5.4 Validierung und Performance des Layers

Genauigkeit Während viele andere VHDL-basierte Convolution-Implementierungen mit auf 8 Bit quantisierten Gewichten (Kernels) arbeiten und dadurch Genauigkeitsverluste in Kauf nehmen, verwendet die hier vorgestellte Architektur Festkommaarithmetik mit höherer numerischer Präzision.

Zur Validierung der Ergebnisse wurde ein Testbild sowohl durch den in VHDL implementierten Convolution-Layer als auch durch ein äquivalentes Modell in Keras verarbeitet. Anschließend wurden die beiden Ausgabebilder pixelweise miteinander verglichen, und die Differenzen wurden in einem Plot visualisiert, um Abweichungen sowohl quantitativ als auch qualitativ zu bewerten.

Als Testbild kam ein für die Bildverarbeitung typisches Bild, „ILENA“, zum Einsatz. Als Kernel wurde ein 3×3 -Kantenerkennungskernel verwendet, dessen Werte eine Genauigkeit von zwei Nachkommastellen besitzen:

$$K = \begin{bmatrix} -1.25 & -0.90 & -1.10 \\ -0.95 & 8.30 & -0.85 \\ -1.15 & -0.95 & -1.20 \end{bmatrix}$$

Für die RGB-Kanäle wurden als Maximalwerte der Abweichung [1,1,1] und als MSE, sowie MAE 0,0013 ermittelt. Aus diesen Werten sowie der Analyse des Differenzbildes lässt sich erkennen, dass Abweichungen ausschließlich in positiver Bitrichtung auftreten. Dies ist vermutlich auf unterschiedliche Rundungsverfahren zurückzuführen.

Performance/- Latenzbetrachtung Die Gesamtlatenz der Architektur ergibt sich aus der Summe der Latenzen des Linebuffers und der Convolutionseinheit:

$$L_{\text{ges}} = L_{\text{Linebuffer}} + L_{\text{Convolution}} = 1 + 3 = 4$$

Das bedeutet, dass das Ergebnis *vier Takte* nach dem Einlesen des letzten Bildpixels vorliegt.



Abbildung 6: Visualisierung der pixelweisen Differenzen zwischen Keras-Referenz und VHDL-Implementierung

Da das Bild pixelweise eingelesen wird, entsteht zunächst eine Latenz vom Beginn bis zum Ende des Einlesevorgangs:

$$L_{\text{read}} = \text{Width} \times \text{Height}$$

Für ein Bild mit einer Breite von 640 Pixeln und einer Höhe von 480 Pixeln gilt somit:

$$L_{\text{read}} = 640 \times 480 = 307,200$$

Die Gesamtdauer vom Start des Einlesens bis zur vollständigen Ausgabe des Ergebnisses berechnet sich daher zu:

$$L = L_{\text{read}} + L_{\text{ges}} = 307200 + 4 = 307204$$

5.5 Akkumulator (accumulator_pingpong)

5.5.1 Aufbau und Schnittstelle

Der **Akkumulator** ist eine Kernkomponente zur Aufsummierung von Teilergebnissen, insbesondere bei Architekturen mit begrenzten DSP-Ressourcen, bei denen die Kanalverarbeitung serialisiert wird. Er ist dafür ausgelegt, einen Block von Eingangswerten (**tdata**), die als gültig markiert sind (**tvalid**), zu einem Gesamtwert zu akkumulieren. Das Ende jedes Blocks wird durch **tlast** signalisiert.

Um eine lückenlose "Back-to-Back"Verarbeitung von Datenblöcken ohne Latenzzyklen (Stalls) zu ermöglichen, basiert die Architektur auf einem **Ping-Pong-Prinzip**.

- **Zwei Akkumulator-Register (acc0, acc1):** Zwei separate Register, die abwechselnd zur Aufsummierung eines Datenblocks verwendet werden.
- **Steuerlogik (use_acc0):** Ein Signal, das umschaltet, welcher der beiden Akkumulatoren gerade aktiv summiert und welcher inaktiv ist.
- **Ausgangsregister (out_reg, out_v_reg):** Pufferregister, die das fertige Ergebnis und dessen Gültigkeit für einen Taktzyklus halten.

5.5.2 Konfigurationsparameter (Generics)

Der **Akkumulator** ist durch die folgenden Parameter generisch konfigurierbar:

- **WIDTH:** Die Bitbreite des **signed** Eingangssignals **tdata**.

- **ACC_WIDTH:** Die Bitbreite der internen Akkumulatoren. Muss ausreichend groß gewählt werden, um einen Überlauf zu verhindern ($\text{typ. WIDTH} + \lceil \log_2(\text{Max_Blocklänge}) \rceil$).

5.5.3 Externe Schnittstelle (Ports)

Tabelle 4: Eingangssignale des Akkumulators

Signal	Format	Beschreibung
clk	1 Bit	Systemtakt.
rst	1 Bit	Globales Reset-Signal (typischerweise aktiv-high).
tvalid	1 Bit	Steuersignal (Stream); zeigt an, dass an tdata ein gültiger Wert anliegt.
tlast	1 Bit	Steuersignal (Stream); markiert das letzte gültige Datenelement eines Verarbeitungsblocks.
tdata	WIDTH Bits	Der signed Eingangsvektor, der zum Block-Ergebnis aufaddiert werden soll.

Tabelle 5: Ausgangssignale des Akkumulators

Signal	Format	Beschreibung
out_valid	1 Bit	Pulst für exakt einen Taktzyklus hoch, wenn out_data das gültige Endergebnis enthält.
out_data	ACC_WIDTH Bits	Das signed Endergebnis der Akkumulation.

Latenzbetrachtung Obwohl der Akkumulator selbst eine inhärente Latenz von nur einem Taktzyklus besitzt, muss die Verarbeitungsstrategie der Kanäle berücksichtigt werden. Da nicht alle Kanäle parallel, sondern nacheinander in N Blöcken verarbeitet werden, entsteht durch diese Serialisierung eine zusätzliche Latenz von N Takten. Die effektive Gesamtlatenz bis zum finalen Ergebnis beträgt somit $N + 1$ Taktzyklen.

5.6 Kernel Multiplexer (kernel_block_mux)

5.6.1 Aufbau und Schnittstelle

Der **kernel_block_mux** ist eine Hardware-Komponente, die als spezialisierter Multiplexer dient. Seine Hauptaufgabe ist es, aus einem großen, flachen Speichervektor (**kernel_pool**), der die Gewichte aller Filter enthält, einen spezifischen Kernel-Block auszuwählen.

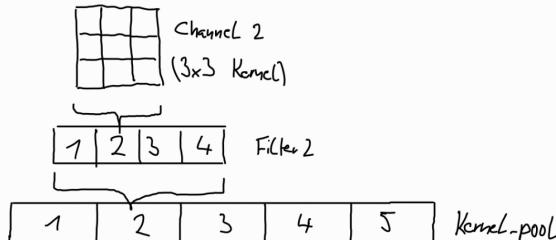


Abbildung 7: Visualisierung der Speicherung von Kernels

Diese Komponente ist notwendig, wenn die Convolution nicht alle Kanäle gleichzeitig verarbeiten kann (Limitierung durch CHANNELS_PER_BLOCK). Der Multiplexer wählt basierend auf dem **filter_sel** und **channel_block** den korrekten Datensatz aus.

Die Logik besteht im Kern aus:

- **Index-Berechnungslogik:** Ermittelt die Startadresse des Datenblocks im `kernel_pool`.
- **Slicing-Logik:** Ein als Vektor-Slice implementierter Multiplexer.
- **Ausgangsregister:** `kernel_out` und `valid_out` sind registriert (1 Takt Latenz).

5.6.2 Konfigurationsparameter (Generics)

Der `kernel_block_mux` wird durch die folgenden Parameter konfiguriert:

- `DATA_WIDTH`: Die Bitbreite eines einzelnen Kernel-Gewichts.
- `KERNEL_SIZE`: Die Kantenlänge des (quadratischen) Faltungskernels.
- `CHANNELS_TOTAL`: Die Gesamtanzahl der Kanäle, die ein kompletter Filter abdeckt.
- `CHANNELS_PER_BLOCK`: Die Anzahl der Kanäle, die der `conv_nxn`-Block *gleichzeitig* verarbeiten kann.
- `FILTERS_TOTAL`: Die Gesamtanzahl der verschiedenen Filter, die gespeichert sind.

5.6.3 Externe Schnittstelle (Ports)

Tabelle 6: Eingangssignale des Kernel Multiplexers

Signal	Format	Beschreibung
<code>clk</code>	1 Bit	Systemtakt.
<code>rst</code>	1 Bit	Globales Reset-Signal.
<code>filter_sel</code>	$\log_2(\text{FILTERS_TOTAL})^*$	Index, welcher Filter (0 bis N-1) verwendet wird.
<code>channel_block</code>	$\log_2(\text{CH_TOT} / \text{CH_PER_BL})^*$	Index, welcher Kanalblock innerhalb des Filters geladen wird.
<code>kernel_pool</code>	(Berechnet)**	Großer Vektor, der die Gewichte aller Filter und Kanäle enthält.

Tabelle 7: Ausgangssignale des Kernel Multiplexers

Signal	Format	Beschreibung
<code>valid_out</code>	1 Bit	Wird für einen Takt '1', um anzugeben, dass <code>kernel_out</code> gültig ist (Latenz: 1 Takt).
<code>kernel_out</code>	(Berechnet)***	Der ausgewählte Kernel-Block.

* Bitbreiten gerundet als `integer(ceil(log2(real(...))))` `downto 0`

** Breite `kernel_pool`: $\text{FILTERS_TOTAL} \times \text{CH_TOTAL} \times \text{KS}^2 \times \text{DATA_WIDTH}$

*** Breite `kernel_out`: $\text{CH_PER_BLOCK} \times \text{KS}^2 \times \text{DATA_WIDTH}$

5.7 Steuereinheit (`conv_control`)

Die `conv_control`-Entität bildet das **Steuerwerk** (*Control Path*) des gesamten Layers. Ihre Funktion wird durch eine **Finite State Machine** (**FSM**) realisiert, welche die **Steuersignale** generiert, um die **sequenzielle Koordination** der Datenpfad-Komponenten zu gewährleisten.

5.7.1 Konzeptionelle Analyse der `conv_control`-Einheit

Die Kernkomplexität der Steuerung liegt in der Synchronisation des eingehenden Pixelstroms mit dem ausgehenden Fenster-Leseprozess.

Dual-Pointer-Synchronisation (Write/Read)

- **Write Pointer** (`w_row, w_col`): Interne Zähler, getaktet von `valid_in_write`, verfolgen den Füllstand des Linebuffers.
- **Read Pointer** (`r_row, r_col`): Interne Zähler, gesteuert von der FSM, inkrementieren basierend auf STRIDE und geben die Lese-Adressen (`row_ptr, col_ptr`) aus.

Pipeline-Stall / Synchronisationslogik Die kritischste Komponente ist die 'Start-Bedingung' (`if ((r_row + PAD-1 < w_row) ...)`), die einen **Read-Before-Write-Hazard** verhindert. Die FSM darf den Rechen-Trigger (`ctrl_out_conv`) nur auslösen, wenn der Write Pointer dem Read Pointer ausreichend voraus ist.

Iteration über Filter und Kanalblöcke Sobald die Synchronisationsbedingung erfüllt ist, betritt die FSM eine innere Schleife (gesteuert durch `filter` und `channel`), um alle MAC-Operationen für diesen einen Ausgabe-Pixel zu triggern.

Akkumulator-Steuerung (`t_last`) Das Signal `t_last` wird gesetzt, wenn der *letzte Kanalblock* eines Filters verarbeitet wird. Dies signalisiert dem Akkumulator, die Summe zu finalisieren und auszugeben.

Overall-Handshake (`conv_abgeschlossen`) Ein Flag `conv_abgeschlossen` implementiert einen Handshake auf Bild-Ebene, um dem übergeordneten System zu signalisieren, dass die Verarbeitung des gesamten Bildes beendet ist und ein neues Bild angenommen werden kann.

6 Implementierung der Pooling-Layer

Pooling-Layer sind eine fundamentale Komponente in CNNs, die primär zur Dimensionsreduktion (Downsampling) der Feature Maps eingesetzt werden. Sie reduzieren die räumliche Größe (Breite und Höhe) der Eingangsdaten, wodurch die Anzahl der Parameter und der Rechenaufwand in nachfolgenden Schichten verringert wird. Dies trägt zur Invarianz gegenüber Translationen bei und hilft Overfitting zu kontrollieren. In dieser Arbeit wurden die zwei gängigsten Pooling-Varianten implementiert: Max Pooling und Global Average Pooling.

6.1 Max Pooling (Max_Pooling_Layer)

6.1.1 Aufbau und Schnittstelle

Max Pooling ist eine Downsampling-Operation, die den Maximalwert aus einer definierten Region (einem $n \times n$ -Fenster) der Eingangs-Feature-Map auswählt. Die hier implementierte `Max_Pooling_Layer`-Entität ist ein Streaming-Wrapper, der, ähnlich dem Convolution Layer, einen `linebuffer` zur Fenstererzeugung und einen Kern-Verarbeitungsblock (`MaxPooling`) instanziert.

- **linebuffer:** Empfängt den seriellen Pixelstrom (`pixel_in, valid_in`) und puffert die Daten, um ein paralleles $n \times n$ -Fenster zu generieren.
- **MaxPooling (Kern):** Diese Komponente empfängt das $n \times n$ -Fenster (`window_in`). Die Logik de-interleaved die Kanäle (R, G, B), findet den Maximalwert für jeden Kanal *unabhängig* voneinander und setzt das Ausgangspixel aus diesen Maxima zusammen.

6.1.2 Konfigurationsparameter (Generics)

Die `Max_Pooling_Layer`-Entität wird durch die folgenden Parameter konfiguriert:

- **WIDTH:** Breite der Eingangs-Feature-Map.
- **HEIGHT:** Höhe der Eingangs-Feature-Map.
- **kernel_size:** Die Kantenlänge des (quadratischen) Pooling-Fensters (z.B. 2 für 2x2).

- **pixeldepth**: Die Bitbreite eines einzelnen Farbkanals (z.B. 8 Bit).
- **Stride**: Die Schrittweite, mit der das Fenster über die Feature Map bewegt wird.

6.1.3 Externe Schnittstelle (Ports)

Tabelle 8: Eingangssignale des Max Pooling Layers

Signal	Format	Beschreibung
clk	1 Bit	Systemtakt.
rst	1 Bit	Globales Reset-Signal.
pixel_in	3*pixeldepth-1..0	Eintreffender Pixelstrom (R, G, B).
valid_in	1 Bit	Signalisiert, dass pixel_in gültig ist.

Tabelle 9: Ausgangssignale des Max Pooling Layers

Signal	Format	Beschreibung
pixel_out	3*pixeldepth-1..0	Das resultierende Pixel (R_max, G_max, B_max).
valid_out	1 Bit	Signalisiert, dass pixel_out gültig ist.

6.1.4 Funktion des Linebuffers im Pooling-Kontext

Die `Max_Pooling_Layer`-Entität nutzt eine `linebuffer`-Komponente, die auch im Faltungs-Layer (Convolution Layer) zum Einsatz kommt. Ihre Funktion ist hier strukturell identisch: Sie dient als Streaming-zu-Parallel-Wandler.

Der Linebuffer empfängt den seriellen `pixel_in`-Strom und speichert `kernel_size - 1` Zeilen in internen BRAM- oder Register-basierten Puffern. Für jeden Takt, in dem ein gültiges Eingangspixel ankommt, assembliert der Linebuffer das vollständige, parallele $n \times n$ -Fenster (`window_out`), indem er auf die gepufferten Zeilen und das aktuelle Pixel zugreift. Dieses Fenster wird direkt an den `MaxPooling`-Kern weitergeleitet.

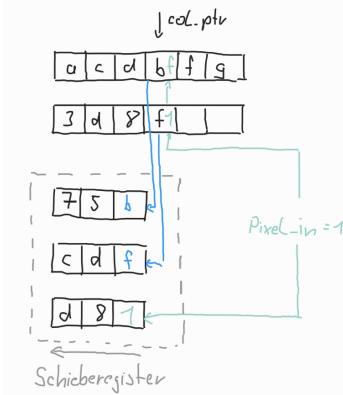


Abbildung 8: Visualisierung der Speicherung im Linebuffer

Die Steuerung der Schrittweite (`Stride`) wird ebenfalls an den Linebuffer delegiert. Die `Max_Pooling_Layer`-Entität reicht ihre `Stride`-Generic direkt an die `linebuffer`-Instanz durch. Der Linebuffer nutzt diesen Wert, um seine internen Adresszeiger (die `row_ptr` und `col_ptr` entsprechen) korrekt zu inkrementieren und so das Downsampling gemäß der definierten Schrittweite zu realisieren.

6.1.5 Validierung und Genauigkeit

Im Gegensatz zur Faltungsoperation, die aufgrund von Festkomma-Arithmetik und Rundungsdifferenzen leichte Abweichungen erwarten lässt, ist Max Pooling eine deterministische Integer-Vergleichsoperation. Die VHDL-Implementierung des MaxPooling-Moduls wurde, analog zur Convolution, gegen ein Keras-Referenzmodell validiert.

Wie in Abbildung 9 dargestellt, wurden die Ausgaben beider Implementierungen (Keras und VHDL) pixelweise verglichen. Die absoluten Differenz-Plots für alle drei Farbkanäle (Rot, Grün, Blau) zeigen keinerlei Abweichung (ein Wert von 0.0 über das gesamte Bild). Dies bestätigt, dass die VHDL-Implementierung bit-identische Ergebnisse zur Software-Referenz liefert.

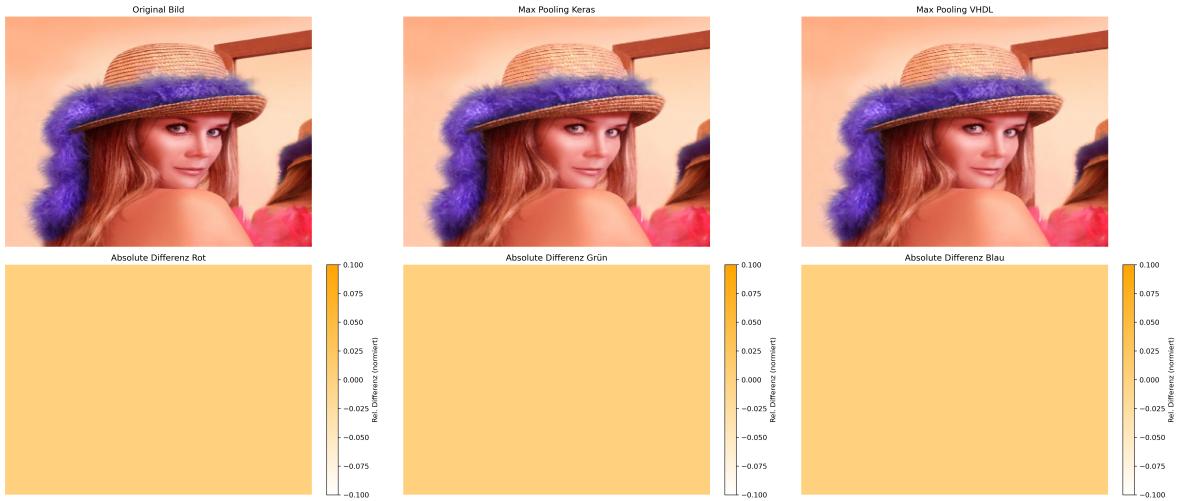


Abbildung 9: Validierungs-Plot für Max Pooling: VHDL-Ausgabe vs. Keras-Referenz. Die Differenz-Plots (unten) sind durchgehend null.

6.2 Global Average Pooling (Global_Avg_Pooling)

6.2.1 Aufbau und Schnittstelle

Global Average Pooling (GAP) ist eine drastische Form des Downsamplings, die typischerweise am Ende eines CNNs vor den finalen Klassifikations-Layern eingesetzt wird. Es reduziert die gesamte räumliche Dimension einer Feature Map (z.B. 640×480) auf einen einzelnen Wert pro Kanal (ein 1×1 -Pixel).

Die hier implementierte `Global_Avg_Pooling`-Entität arbeitet als Streaming-Komponente:

- **Akkumulation:** Sie empfängt einen seriellen Pixelstrom (`pixel_in`, `pixel_valid`). Für jedes gültige Pixel werden die Kanalwerte (R, G, B) getrennt auf 32-Bit breite Summenregister (`sum_r`, `sum_g`, `sum_b`) addiert.
- **Pixel-Zählung:** Ein Zähler (`count`) zählt die Anzahl der verarbeiteten Pixel.
- **Division:** Sobald das letzte Pixel der Feature Map (definiert durch `WIDTH * HEIGHT`) verarbeitet wurde, wird die Division (z.B. `sum_r / NUM_PIXELS`) durchgeführt, um den Durchschnittswert für jeden Kanal zu ermitteln.
- **Ausgabe:** Die Komponente gibt einen einzelnen Pixel (`avg_out`) und einen einzelnen Gültigkeitspuls (`avg_valid`) für den gesamten Frame aus.

6.2.2 Konfigurationsparameter (Generics)

Der `Global_Avg_Pooling`-Block wird durch die Dimensionen der Feature Map konfiguriert:

- **WIDTH:** Breite der Eingangs-Feature-Map.
- **HEIGHT:** Höhe der Eingangs-Feature-Map.

6.2.3 Externe Schnittstelle (Ports)

Tabelle 10: Eingangssignale des Global Average Pooling

Signal	Format	Beschreibung
clk	1 Bit	Systemtakt.
rst	1 Bit	Globales Reset-Signal.
pixel_in	24 Bits (8+8+8)	Eintreffender Pixelstrom (R, G, B).
pixel_valid	1 Bit	Signaliert, dass pixel_in gültig ist.

Tabelle 11: Ausgangssignale des Global Average Pooling

Signal	Format	Beschreibung
avg_out	24 Bits (8+8+8)	Der berechnete Durchschnitts-Pixel (Avg_R, Avg_G, Avg_B) für den gesamten Frame.
avg_valid	1 Bit	Pulst für einen Takt, wenn avg_out gültig ist.

7 Zusammenfassung

Diese Arbeit detailliert die VHDL-Implementierung und Analyse von fundamentalen Komponenten eines Convolutional Neural Network (CNN), die für den Einsatz auf FPGAs optimiert sind. Die Architektur ist modular aufgebaut und als generische Streaming-Pipeline konzipiert.

Im Mittelpunkt steht der Faltungs-Layer, bestehend aus einer zentralen Steuereinheit (`conv_control`), einem `conv_nxn`-Rechenkern und einem Puffer, der 'Same Padding' mittels eines Framebuffers realisiert. Ein `kernel_block_mux` verwaltet die Filtergewichte, während ein `accumulator_pingpong` eine lückenlose Aufsummierung serialisierter Kanalergebnisse sicherstellt.

Darüber hinaus wurden die beiden gängigsten Pooling-Operationen implementiert: `Max_Pooling_Layer`, das ebenfalls einen Linebuffer zur Fenstererzeugung nutzt, und `Global_Avg_Pooling` zur Reduktion ganzer Feature-Maps auf einen Einzelwert.

Ein wesentliches Merkmal ist die hohe numerische Präzision. Die Validierung des Faltungs-Layers gegen ein Keras-Modell zeigte unter Verwendung von Festkommaarithmetik nur minimale Abweichungen ($MSE \approx 0,0013$). Die Max-Pooling-Implementierung lieferte sogar bit-identische Ergebnisse zur Software-Referenz. Die Pipeline-Architektur ermöglicht einen hohen Datendurchsatz bei minimaler Latenz.

8 Fazit

Die Arbeit demonstriert die erfolgreiche, modulare und generische Implementierung von Kernbausteinen eines CNNs – namentlich Convolution, Max Pooling und Global Average Pooling – in VHDL. Die entwickelten Komponenten zeichnen sich durch ihre Flexibilität, hohe numerische Genauigkeit und eine performante Streaming-Fähigkeit aus.

Die Validierungsergebnisse bestätigen, dass die Hardware-Implementierung eine präzise Alternative zu Software-Modellen darstellt. Damit ist eine solide Grundlage für den Entwurf einer vollständigen CNN-Inferenz-Pipeline (siehe Abbildung 2) auf FPGAs geschaffen.

Das primäre Optimierungspotenzial liegt in der Speicherarchitektur des Faltungs-Puffers. Die aktuelle Implementierung nutzt einen BRAM-intensiven Framebuffer für flexible Fensterzugriffe.

Die logischen nächsten Schritte sind die Integration der hier validierten Blöcke sowie die Ergänzung fehlender Komponenten, wie der Bias-Addition und der ReLU-Aktivierungsfunktion, um die vollständige, in der Gesamtarchitektur gezeigte Inferenz-Pipeline zu realisieren. Zudem sind die Pooling Layers noch auf variable Eingänge anzupassen.

9 Quellen

- ILena.jpg, Wikimedia Commons, Lizenz: CC BY-SA 2.5 Brasilien. :contentReference[oaicite:0]index=0
- Lena (Testbild), Wikipedia. :contentReference[oaicite:1]index=1

Abkürzungsverzeichnis

- **ALU:** Arithmetic Logic Unit (Arithmetisch-logische Einheit)
- **CHANNELS (CH):** Anzahl der Eingangskanäle (z. B. RGB)
- **Conv:** Convolution (Faltung)
- **FSM:** Finite State Machine (Endlicher Automat / Zustandsmaschine)
- **FPGA:** Field-Programmable Gate Array
- **KS:** Kernel Size (Kantenlänge des Faltungskerns)
- **TOTAL_BITS_IN:** Gesamt-Bitbreite des Eingangspixels

Verwendete Parameter

Diese Begriffe werden oft als Akronyme in Formeln oder Code verwendet:

- **FRAC_BITS:** Fractional Bits (Anzahl der Nachkommastellen)
- **INT_BITS:** Integer Bits (Anzahl der Vorkommastellen)
- **KS²:** Kernel Size squared (Anzahl Kernelemente)