Pixel-Level Multi-Channel Image Convolution verified RLT Using Python and OpenCV

The **CNN Accelerator** is a custom-designed hardware architecture that efficiently performs **convolutional neural network (CNN) operations** such as **multi-channel convolution, ReLU activation, max pooling, and memory-optimized storage**. Unlike generic processors or GPUs, this accelerator is highly **specialized for deep learning workloads**, focusing on **throughput, latency reduction, and optimized memory utilization**.

It leverages **FPGA/ASIC-friendly design principles**, such as **dual-port BRAM-based storage, pipelined multiply-accumulate (MAC) units, and efficient address generation modules**, to achieve **parallelism at multiple levels (across pixels, channels, and filters)**. The design ensures **scalability** (supporting arbitrary image sizes n×n and kernel sizes k×k) and **flexibility** (configurable number of channels ch and filters fi).

By combining **depthwise convolution flow, on-chip memory reuse, and streaming data handling**, the accelerator achieves a **balanced trade-off between area, speed, and power**, making it suitable for **real-time image processing and AI edge applications** where conventional solutions fall short.

**Key Design Features of Your CNN Accelerator**

**1. Highly Parameterized Design**

* n, k, ch, fi, MA, DEPTH all parameterized.
* Can scale image size (n\*n), kernel size (k\*k), number of channels (ch), and filters (fi) easily.
* Makes the same RTL reusable for multiple CNN layers (Conv, Pool, ReLU, Multi-Filter).
* Flexibility is a **big advantage** in hardware accelerators.

**2. Block RAM (BRAM) Based Storage**

* Using (\* ram\_style = "block" \*) ensures **efficient mapping to FPGA BRAM**, saving LUTs/FFs.
* Dual-port BRAM (bram\_dual\_port1) allows **parallel read/write**:
  + Port A → Image/Kernel fetch
  + Port B → Convolution/Pooling write-back
  + This is more efficient than distributed RAM or registers.

**3. Line Buffering for Sliding Window**

* line[0:k-1][0:n-1] register array works as a **line buffer**.
* Avoids re-fetching pixels from external memory for every kernel window → **huge memory bandwidth reduction**.
* This mimics **hardware-efficient image processing pipelines**.

**4. Shift Index Generators**

* Modules like shift\_index\_generetor, pqindex\_generetor, and pool\_pq\_index generate sliding window indices.
* They handle **dynamic indexing in convolution and pooling** without wasting cycles in nested loops.
* Saves control logic and improves parallelism.

**5. Pipelined MAC (Multiply–Accumulate) Array**

* mac9 module for **9 parallel multiplications + accumulation** in one cycle.
* Data feeding + kernel weights + pipelined MAC ensures **throughput of one convolution per cycle** after pipeline fill.
* Critical for CNN acceleration.

**6. Integrated Multi-Channel & Multi-Filter Handling**

* chh\_no and fi\_no counters support:
  + Multi-channel convolution (RGB or more).
  + Multi-filter convolution (for multiple feature maps).
* Handles **3D convolution (depthwise + pointwise)** inside one FSM.
* Saves control overhead, avoids external scheduling.

**7. ReLU + MaxPooling in Hardware**

* ReLU implemented as simple sign-check (if(dout\_a[31]==1) relu=0 else relu=dout\_a).
* max\_pool4 computes max over 2x2 block.
* Pooling indices auto-generated, values written back to BRAM.
* Accelerator performs **end-to-end CNN block (Conv → ReLU → Pool)**, not just convolution.

**8. Memory Store Indexing**

* mem\_store\_index automatically generates storage addresses for pooled results.
* Avoids address calculation overhead in main FSM.
* Makes write-back efficient.

**9. FSM Controlled Operation**

* Eight pipeline states (s0 → s7):
  + s0: Kernel loading
  + s1: Convolution (MAC)
  + s2: Channel adjust
  + s3: Address alignment
  + s4: ReLU + Pooling
  + s5: Multi-filter adjust
  + s6: Reset write address
  + s7: Completion
  + Clear separation of tasks → easier debugging, better hardware scheduling.

**10. Scalability for FPGA/ASIC**

* BRAM-based storage + FSM + parameterization → works on FPGA and scales to ASIC.
* Optimized for **low memory bandwidth + high compute reuse**, which is critical for CNN accelerators.

A blue circuit board with a white background

AI-generated content may be incorrect.

The original image has been separated into three individual channels—Channel 1 (R), Channel 2 (G), and Channel 3 (B)—each with a resolution of 256 × 256 pixels.

import cv2

import numpy as np

# Input and output paths

img\_path = r"D:\python file\standard\_convolution\amp.webp"

out\_file = r"D:\python file\standard\_convolution\pixels\_channelwise.txt"

# Step 1: Read the image

img = cv2.imread(img\_path)

# Step 2: Resize to 256x256

img\_resized = cv2.resize(img, (256, 256))

# Step 3: Split into channels (OpenCV loads as BGR)

B, G, R = cv2.split(img\_resized)

# Step 4: Write pixel values to file (channel-wise)

with open(out\_file, "w") as f:

    # Write B channel

    for row in B:

        for val in row:

            f.write(f"{val}\n")

    # Write G channel

    for row in G:

        for val in row:

            f.write(f"{val}\n")

    # Write R channel

    for row in R:

        for val in row:

            f.write(f"{val}\n")

print("Pixel file generated successfully at:", out\_file)

by using this code I have generated pixels\_channelwise.txt this file .then convert into this hex file now this hex is uploaded to my CNN Accleretor(here depthwise and standard convolution possible\_multichannel \_multifilter possible).After simmuletion it generates this file .Again this file is uploaded into this python code.



import numpy as np

import cv2

# Input pixel file (single channel convolution output)

in\_file = r"D:\memorey test file for cnn\_accleretor\bram\_dump6.txt"

# Image parameters

height = 254

width = 256

num\_pixels = height \* width  # 65024 pixels expected

# Step 1: Read pixel values

with open(in\_file, "r") as f:

    lines = f.readlines()

# Convert to integers

lines = [int(x.strip()) for x in lines]

# Step 2: Sanity check

if len(lines) != num\_pixels:

    raise ValueError(f"Pixel file has {len(lines)} values, but expected {num\_pixels} ({height}x{width})")

# Step 3: Reshape into 2D matrix

img = np.array(lines, dtype=np.float32).reshape((height, width))

# Step 4: Normalize to 0–255 (for saving as image)

img\_norm = cv2.normalize(img, None, 0, 255, cv2.NORM\_MINMAX)

img\_uint8 = img\_norm.astype(np.uint8)

# Step 5: Save as PNG image

out\_file = r"D:\python file\standard\_convolution\multicha\_convolution\_output1.png"

cv2.imwrite(out\_file, img\_uint8)

print(f"Image generated successfully at: {out\_file}")

A close-up of a circuit board

AI-generated content may be incorrect.A close-up of a circuit board

AI-generated content may be incorrect.

This is expected Getting from CNN RLT

# Step 5: Define 3x3x3 kernel (example)

kernel = np.array([

    [[0, -1, 0],

     [-1, 5, -1],

     [0, -1, 0]],   # for Red

    [[0, -1, 0],

     [-1, 5, -1],

     [0, -1, 0]],   # for Green

    [[0, -1, 0],

     [-1, 5, -1],

     [0, -1, 0]]    # for Blue

],

This 3\*3\*3 kernel I had used to my CNN\_Accleretor.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect. Hardware Utilization for 256\*256 image matrix

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.