HARDWARE ACCELERATION OF A CNN-BASED MODEL USING VITIS HLS ON PYNQ-Z2 FPGA

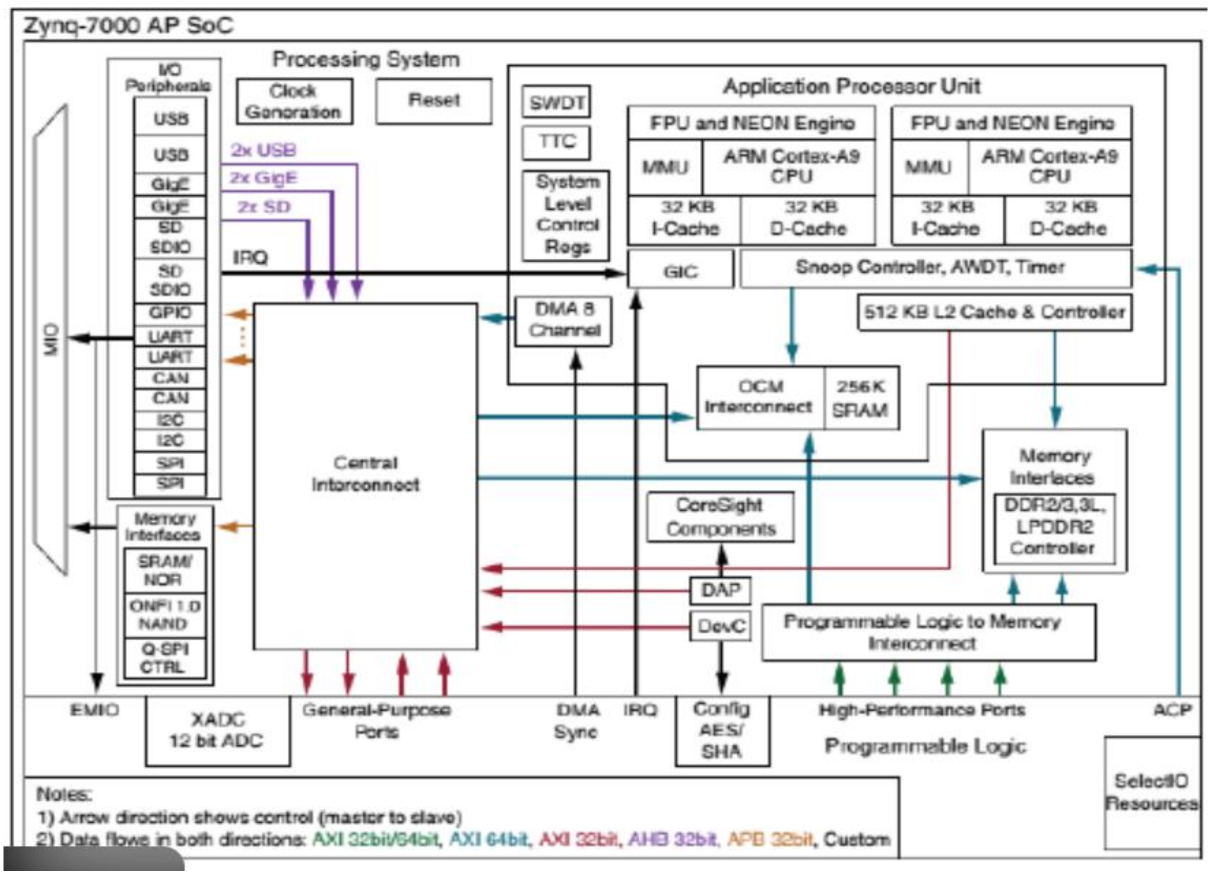
Team Name

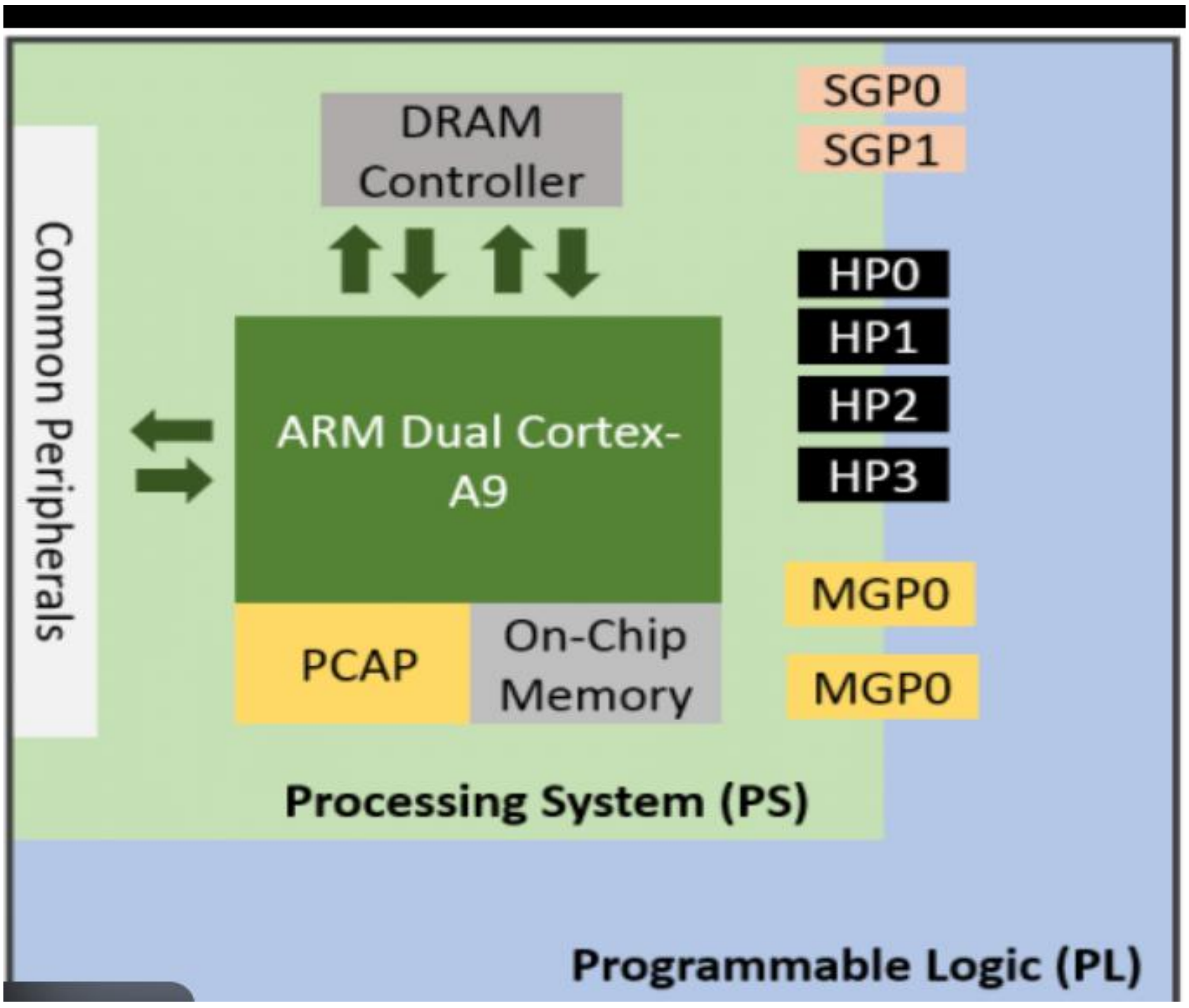
soCrafters

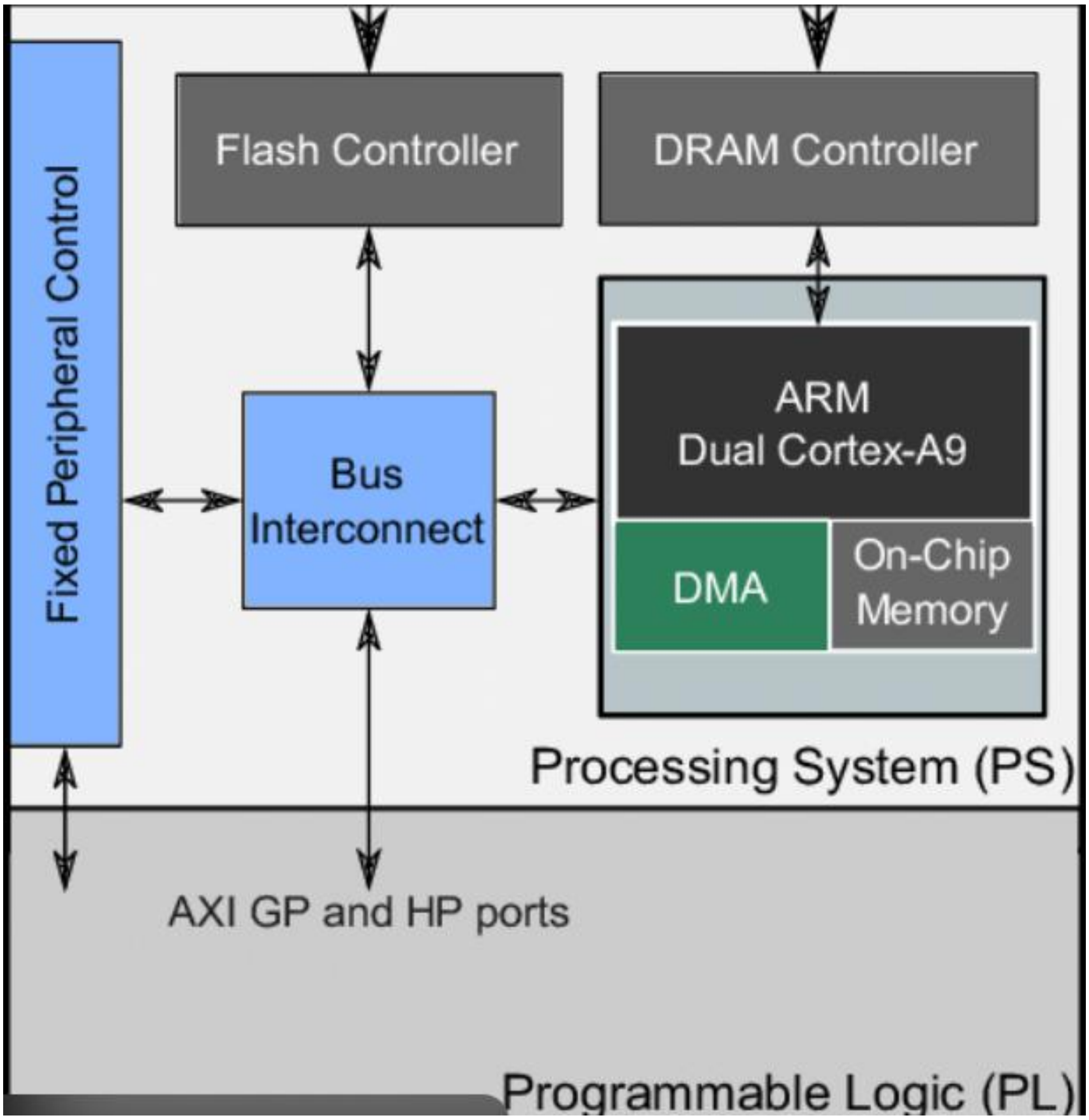
Team Members

Hartik Rai – Team Leader
Lakshya Sharma
Pushpendra Singh

Department of Electronics and Communication Engineering
National Institute of Technology, Jalandhar




DRAM Controller
SGP0
SGP1
HP0
HP1
HP2
HP3
Common Peripherals
ARM Dual Cortex-A9
MGP0
MGP0
PCAP
On-Chip Memory
Processing System (PS)
Programmable Logic (PL)



ABSTRACT

The rapid growth of artificial intelligence applications has significantly increased the computational demand for real-time image processing systems. Convolutional Neural Networks (CNNs) are widely used in computer vision tasks; however, their convolution layers require intensive multiply-accumulate operations, resulting in high latency when executed on general-purpose processors.

This project presents a complete hardware-software co-design methodology for accelerating the convolution layer of a CNN using Vitis High-Level Synthesis (HLS) and deploying it on the PYNQ-Z2 FPGA platform based on the Xilinx Zynq-7000 SoC architecture. The CNN model was trained in Python, and the

convolution layer was extracted and implemented in synthesizable C++ code. Hardware optimization techniques such as pipelining, loop unrolling, and array partitioning were applied to reduce latency and increase throughput.

The resulting RTL design was exported as an IP core, integrated into Vivado block design, and deployed onto FPGA fabric. Performance improvements were evaluated in terms of latency reduction, resource utilization, and parallel execution efficiency.

INTRODUCTION

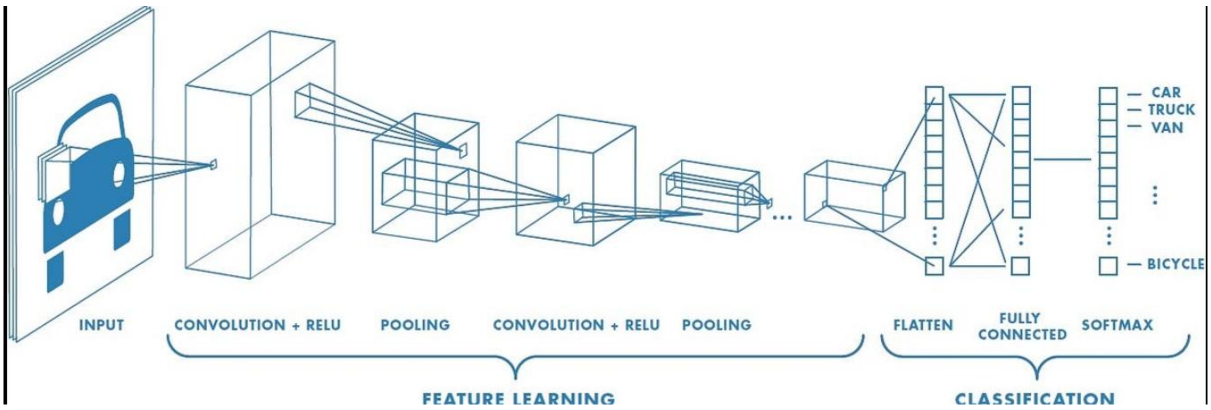
Convolutional Neural Networks (CNNs) have become the backbone of modern computer vision applications, including image classification, object detection, medical image analysis, and autonomous navigation. Despite their success, CNN inference requires large-scale matrix operations and repeated convolution computations.

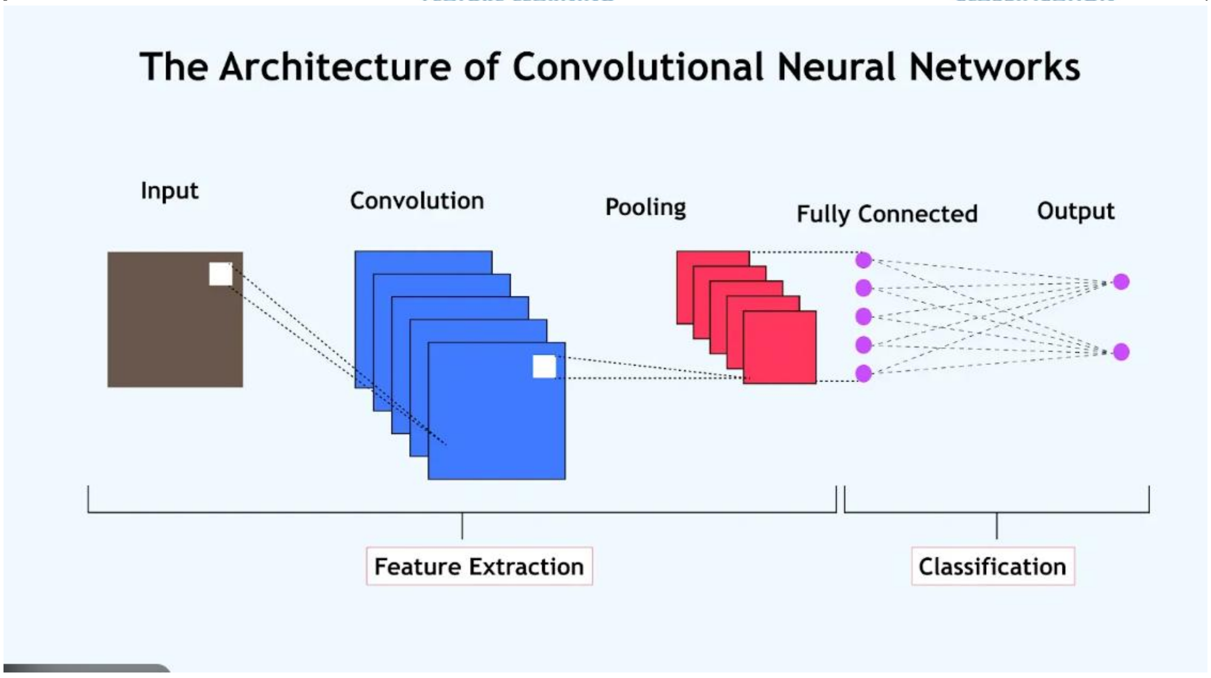
The convolution layer alone accounts for approximately 70–90% of total computation in CNN models. Traditional CPU-based execution suffers from:

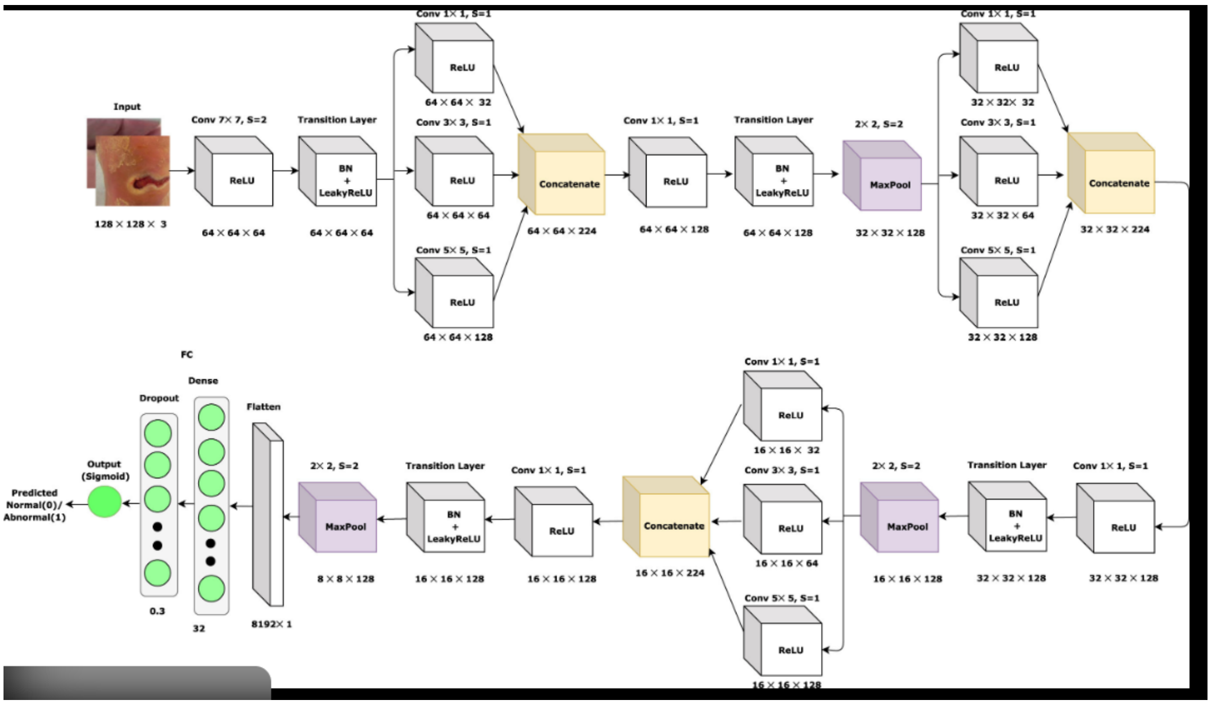
• Sequential execution
• Limited instruction-level parallelism
• High execution time
• Increased power consumption

Field Programmable Gate Arrays (FPGAs) provide a spatial computing architecture, allowing parallel hardware execution. Unlike CPUs that execute instructions sequentially, FPGAs allow the creation of custom datapaths that execute multiple arithmetic operations simultaneously.

The primary objective of this work is to demonstrate the acceleration of convolution operations using FPGA through High-Level Synthesis (HLS) and deploy the accelerator on a PYNQ-Z2 board.


CAR
TRUCK
VAN
BICYCLE
INPUT
CONVOLUTION + RELU
POOLING
CONVOLUTION + RELU
POOLING
FLATTEN
FULLY CONNECTED
SOFTMAX
FEATURE LEARNING
CLASSIFICATION


The Architecture of Convolutional Neural Networks
Input
Convolution
Pooling
Fully Connected
Output
Feature Extraction
Classification

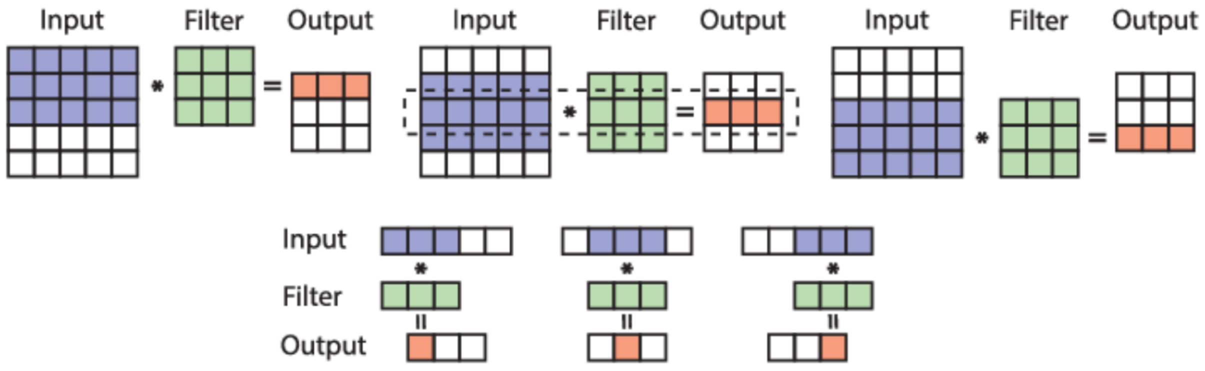

Input
128 × 128 × 3
Conv 7× 7, S=2
ReLU
64 × 64 × 64
Transition Layer
BN + LeakyReLU
64 × 64 × 64
Conv 1× 1, S=1
ReLU
64 × 64 × 32
Conv 3× 3, S=1
ReLU
64 × 64 × 64
Conv 5× 5, S=1
ReLU
64 × 64 × 128
Concatenate
64 × 64 × 224
Conv 1× 1, S=1
ReLU
64 × 64 × 128
Transition Layer
BN + LeakyReLU
64 × 64 × 128
2× 2, S=2
MaxPool
32 × 32 × 128
Conv 1× 1, S=1
ReLU
32 × 32× 32
Conv 3× 3, S=1
ReLU
32 × 32 × 64
Conv 5× 5, S=1
ReLU
32 × 32 × 128
Concatenate
32 × 32 × 224
Conv 1× 1, S=1
ReLU
32 × 32 × 128
Transition Layer
BN + LeakyReLU
32 × 32 × 128
2× 2, S=2
MaxPool
16 × 16 × 128
Conv 1× 1, S=1
ReLU
16 × 16 × 32
Conv 3× 3, S=1
ReLU
16 × 16 × 64
Conv 5× 5, S=1
ReLU
16 × 16 × 128
Concatenate
16 × 16 × 224
Conv 1× 1, S=1
ReLU
16 × 16 × 128
Transition Layer
BN + LeakyReLU
16 × 16 × 128
2× 2, S=2
MaxPool
8 × 8 × 128
Flatten
8192× 1
FC
Dense
32
Dropout
0.3
Output (Sigmoid)
Predicted Normal(0)/ Abnormal(1)

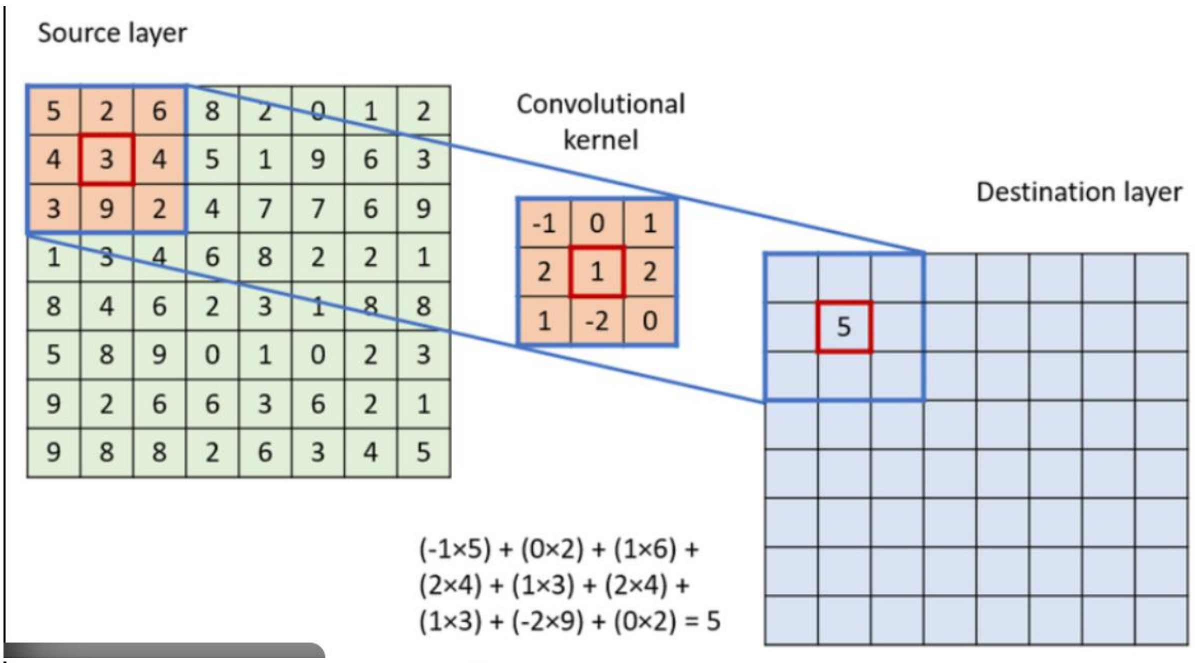
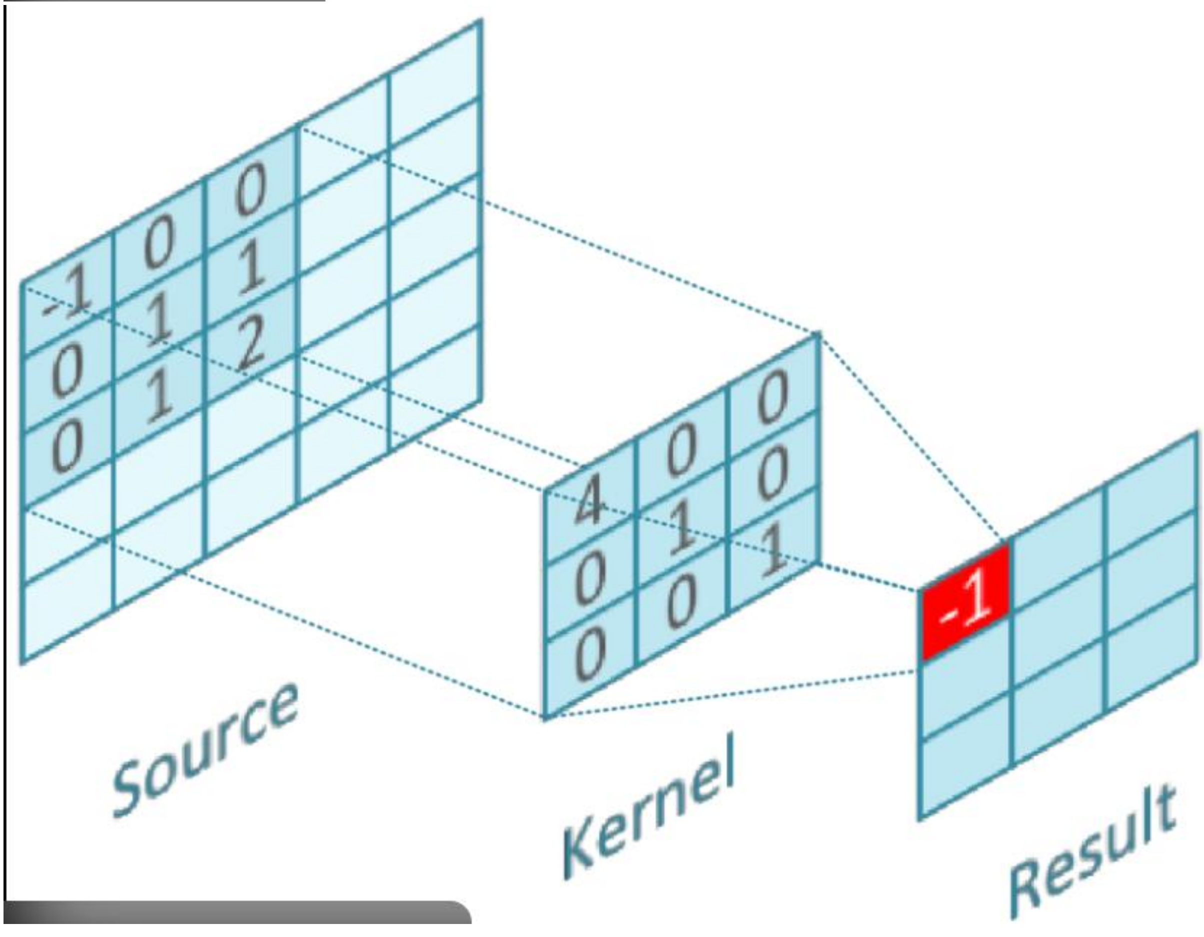
MATHEMATICAL FOUNDATION OF CONVOLUTION

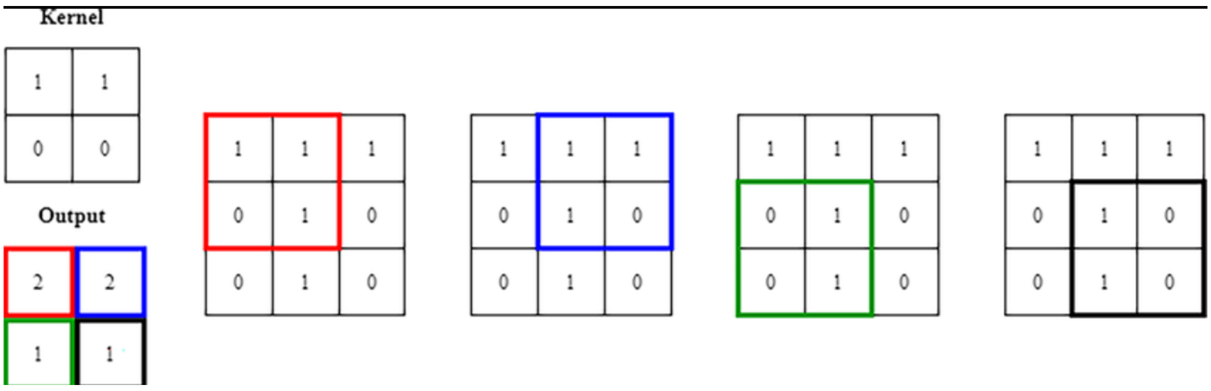
The two-dimensional discrete convolution operation is mathematically defined as:

$$y(i,j) = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} x(i+m, j+n) \cdot w(m,n)$$

Figure 1: Sliding window representation of 2D convolution using a 3×3 kernel over an input feature map.




Source layer
5 2 6 8 2 0 1 2
4 3 4 5 1 9 6 3
3 9 2 4 7 7 6 9
1 3 4 6 8 2 2 1
8 4 6 2 3 1 8 8
5 8 9 0 1 0 2 3
9 2 6 6 3 6 2 1
9 8 8 2 6 3 4 5
Convolutional kernel
-1 0 1
2 1 2
1 -2 0
Destination layer
5
(-1×5) + (0×2) + (1×6) +
(2×4) + (1×3) + (2×4) +
(1×3) + (-2×9) + (0×2) = 5

-1 0 0
0 1 1
0 1 2
4 0 0
0 1 0
0 0 1
-1
Source
Kernel
Result



Where:

x(i,j) represents the input feature map
w(m,n) represents the kernel weight
k is the kernel dimension
y(i,j) represents the output feature map

For an input of size N × N and kernel size k × k:

Total multiplications required = $N^2 \times k^2$

For example, if N = 64 and k = 3:

Multiplications = $64^2 \times 9 = 36{,}864$

This computational density explains why convolution layers are computational bottlenecks in CNN architectures.

SYSTEM ARCHITECTURE

The implemented system is based on the Xilinx Zynq-7000 SoC architecture, which integrates:

1. Processing System (PS)
    - ARM Cortex-A9 processor
    - DDR memory controller
    - Peripherals
2. Programmable Logic (PL)
    - FPGA fabric

  - DSP slices
  - BRAM blocks
  - LUTs and Flip-Flops

The Processing System handles control operations, operating system execution, and data movement. The Programmable Logic performs hardware-accelerated convolution.

Communication between PS and PL is achieved using AXI interconnect protocols.

HARDWARE IMPLEMENTATION USING VITIS HLS

The convolution layer was implemented in C++ and synthesized using Vitis HLS.

The core structure consists of nested loops iterating over image dimensions and kernel dimensions. Without optimization, each multiply-accumulate operation executes sequentially.

To improve performance, the following optimization techniques were applied:

- Loop Pipelining
- Loop Unrolling
- Array Partitioning
- Fixed-Point Arithmetic
- AXI Stream Interface

LOOP PIPELINING

Loop pipelining allows overlapping execution of loop iterations. Using:

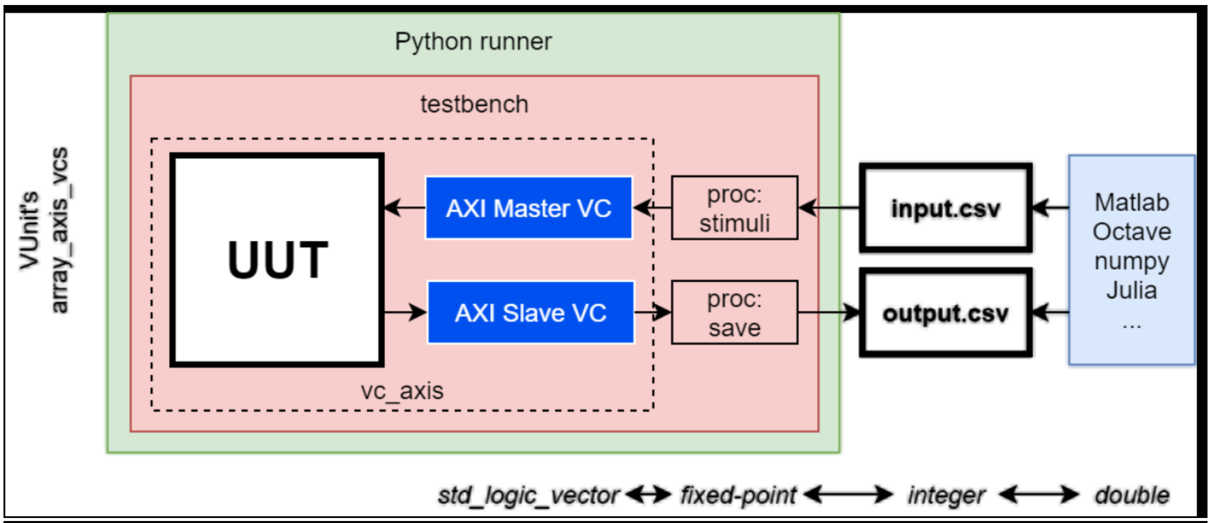
#pragma HLS PIPELINE II=1

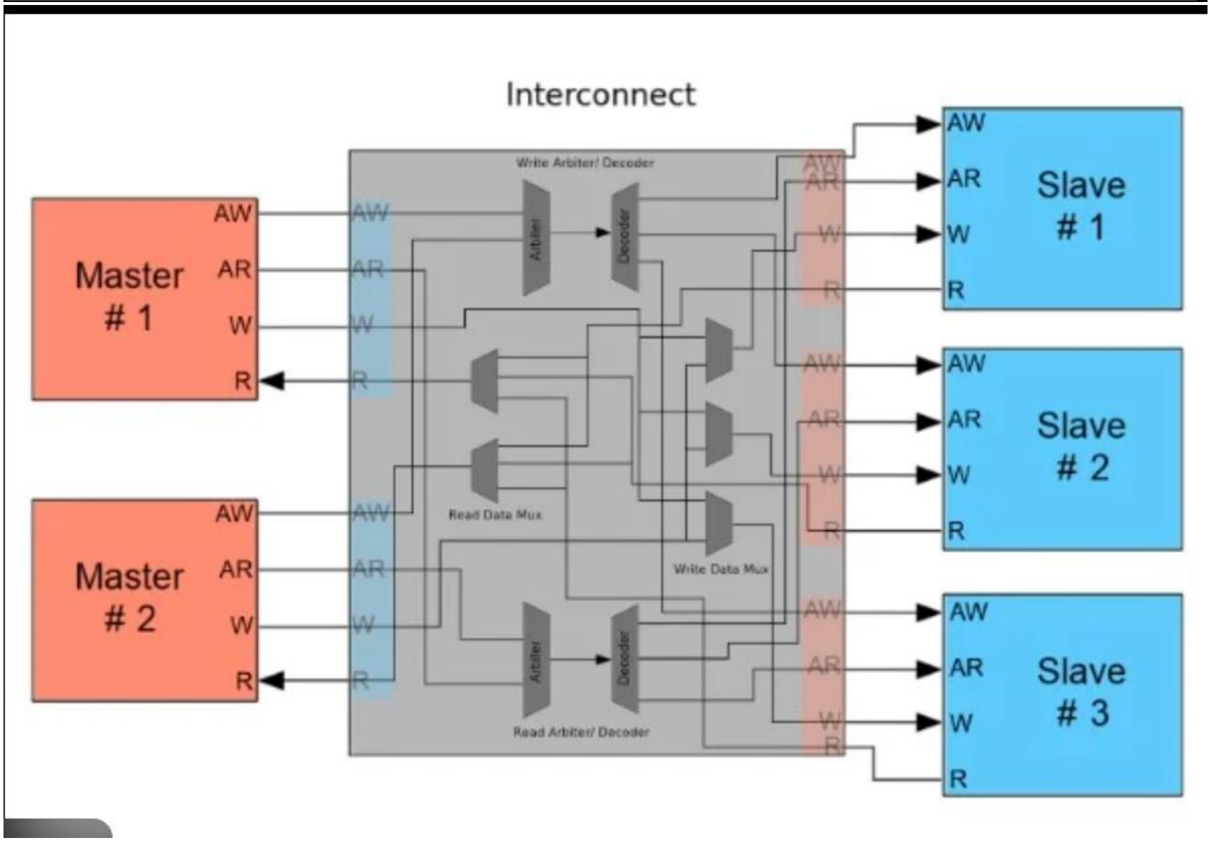
the design achieves an initiation interval (II) of 1, meaning a new pixel computation starts every clock cycle after pipeline fill.

Latency formula:

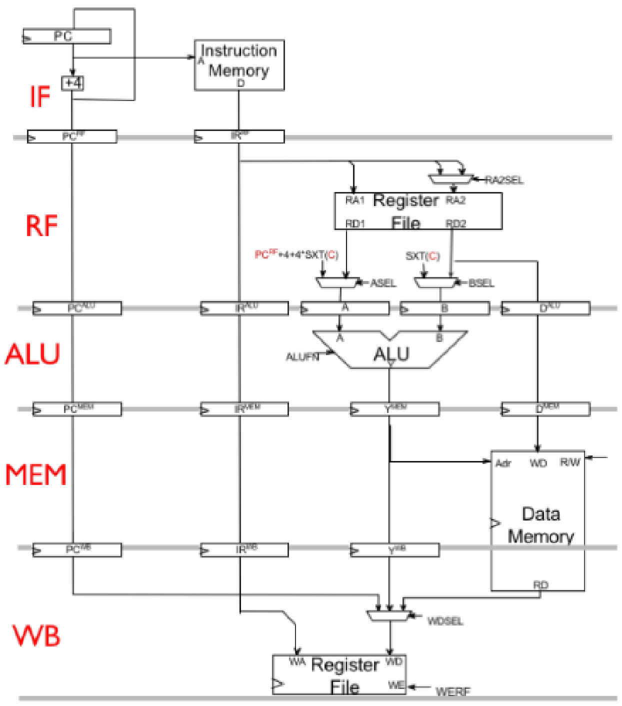
Latency = Pipeline Depth + Total Iterations − 1

Pipelining significantly reduces effective execution time.


VUnit's array_axis_vcs
Python runner
testbench
UUT
AXI Master VC
AXI Slave VC
vc_axis
proc: stimuli
proc: save
input.csv
output.csv
Matlab
Octave
numpy
Julia
...
std_logic_vector ↔ fixed-point ↔ integer ↔ double


Interconnect
Master # 1
Master # 2
AW
AR
W
R
Write Arbiter/ Decoder
Arbiter
Decoder
Read Data Mux
Write Data Mux
Read Arbiter/ Decoder
Slave # 1
Slave # 2
Slave # 3

5-Stage Pipelined Datapath



- Pipeline registers separate different stages:
  - IF – instruction fetch
  - RF – register file access
  - ALU – compute result
  - MEM – memory access
  - WB – write back to reg. file
- Each stage services one instruction per cycle
- Data memory reads are now pipelined, not combinational
  - Data read appears in RD the next cycle

LOOP UNROLLING

Loop unrolling replicates hardware resources to perform multiple operations simultaneously.

For a 3×3 kernel:

Unrolling factor = 9

This instantiates 9 multipliers (DSP slices) operating in parallel.

Speedup ≈ 9×

Tradeoff: Increased DSP usage.

MEMORY OPTIMIZATION

Memory bandwidth often limits accelerator performance. To avoid bottlenecks:

• Line buffers were used to store intermediate rows.
• Array partitioning enabled parallel memory access.
• BRAM blocks were utilized efficiently.

Without partitioning, memory conflicts limit throughput.

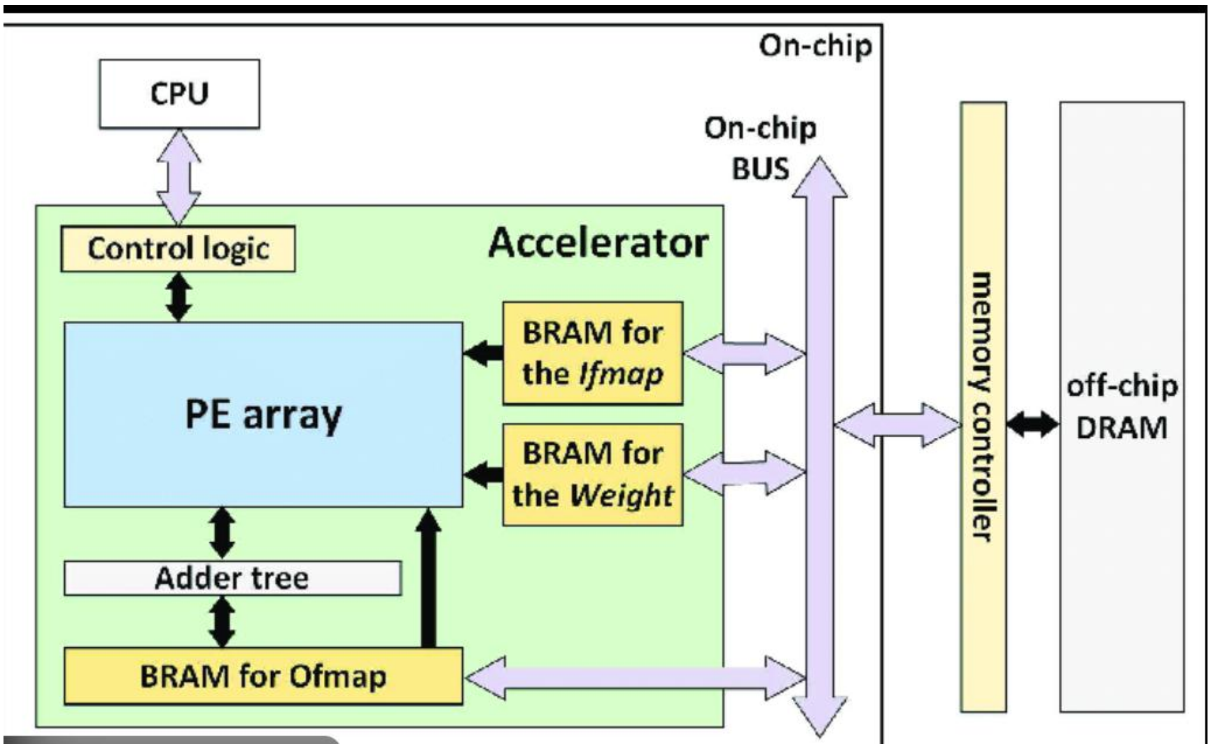
AXI COMMUNICATION PROTOCOL

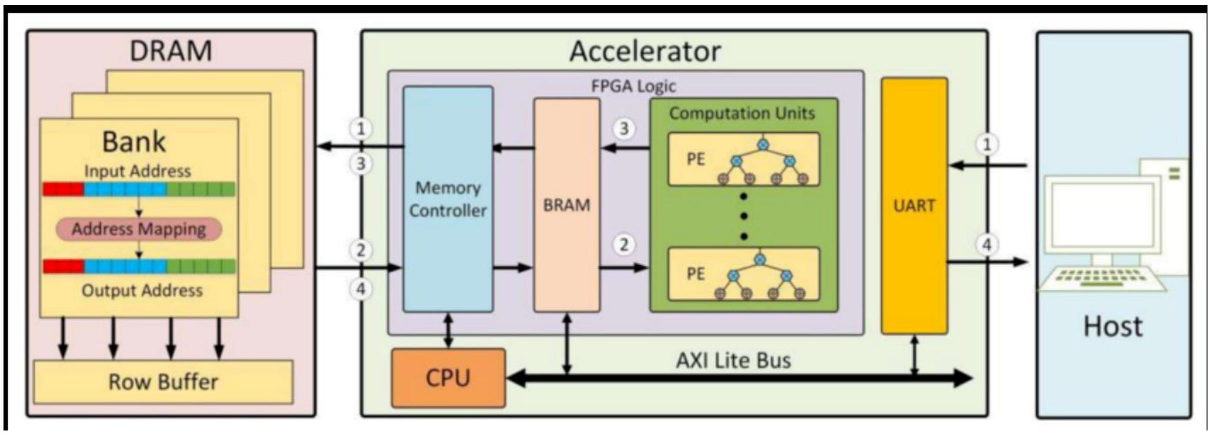
Two AXI interfaces were used:

AXI Lite:
• Used for control registers
• Low bandwidth
• Start/Stop signal

AXI Stream:
• Used for pixel data transfer
• High throughput
• Continuous data flow

Streaming architecture ensures minimal overhead and supports pipelined computation.





LATENCY AND THROUGHPUT ANALYSIS

Sequential CPU Latency:

$L\_seq = N^2 \times k^2$

Parallel FPGA Latency:

$L\_parallel = N^2$

For $k = 3$:

Speedup $\approx 9\times$

Example:

Image size = 64×64
Clock frequency = 100 MHz

Sequential time ≈ 368 µs
FPGA time ≈ 41 µs

RESOURCE UTILIZATION

Typical resource usage:

Resource	Purpose
LUT	Combinational logic
Flip-Flops	Pipeline registers
DSP slices	Multipliers
BRAM	Feature buffering

DSP slices are heavily utilized for multiply-accumulate operations.

ENERGY EFFICIENCY

Energy = Power × Time

Although FPGA may consume moderate instantaneous power, reduced execution time results in lower overall energy consumption.

This makes FPGA suitable for edge AI devices.

DEPLOYMENT PROCESS

1. CNN trained in Python
2. Convolution weights extracted
3. Fixed-point conversion performed
4. C++ implemented in Vitis HLS
5. C Simulation executed
6. C Synthesis performed

7. RTL exported as IP
8. Vivado block design created
9. Bitstream generated
10. SD card flashed with PYNQ image
11. Hardware executed

PERFORMANCE COMPARISON

Platform	Execution Style	Latency	Power
CPU	Sequential	High	Moderate
GPU	SIMD	Low	High
FPGA	Custom Parallel	Low	Low

FPGA provides optimal tradeoff for embedded AI inference.

APPLICATIONS

FPGA-based CNN acceleration can be used in:

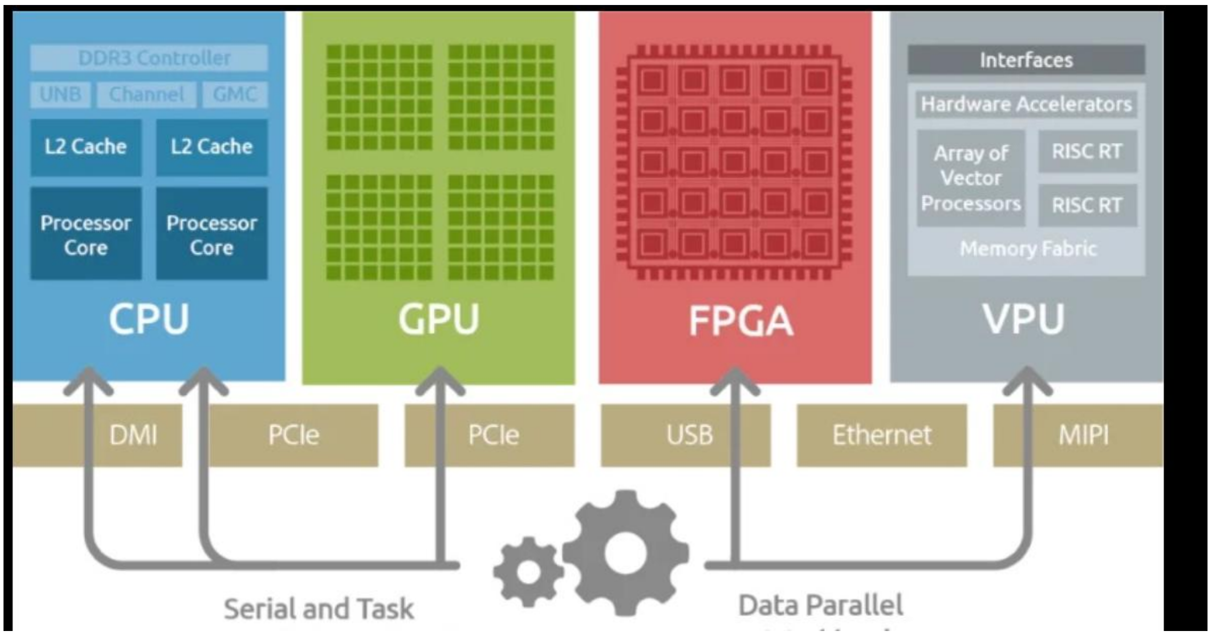
• Smart surveillance systems
• Autonomous vehicles
• Medical imaging (MRI/CT)
• Industrial defect detection
• Drone vision systems
• Edge AI IoT devices

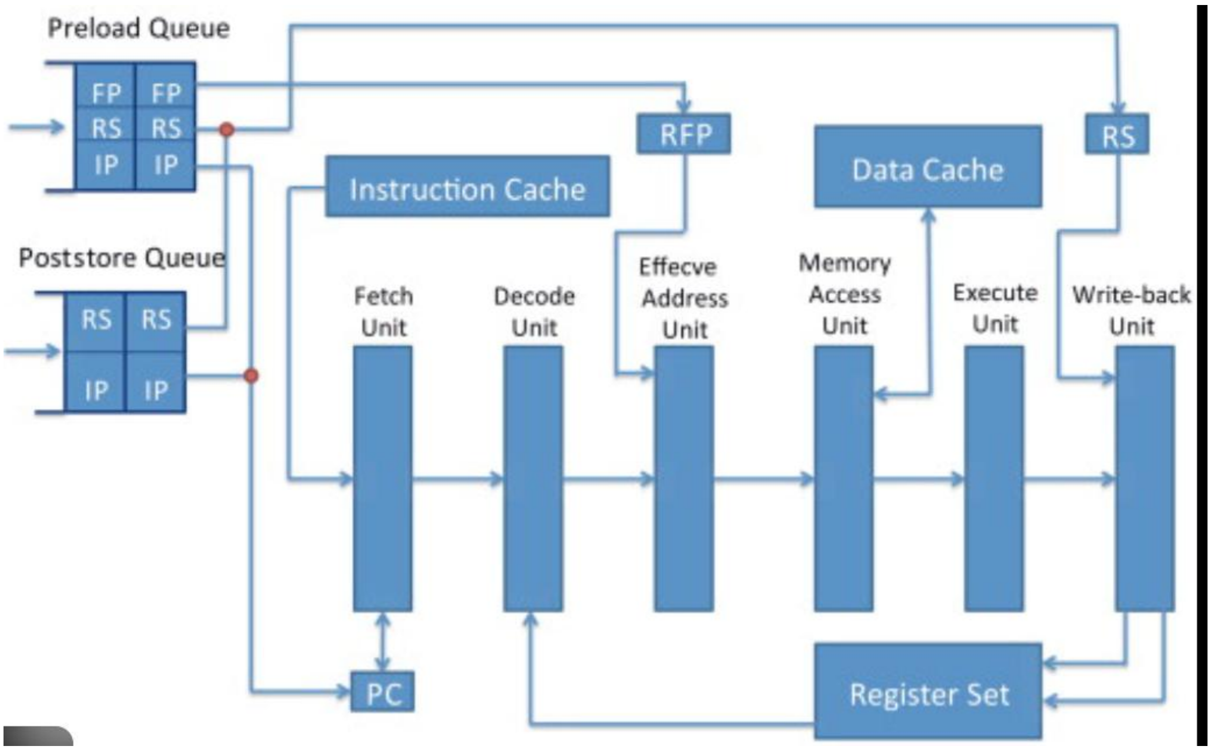
Real-time deterministic latency makes FPGA ideal for safety-critical applications.

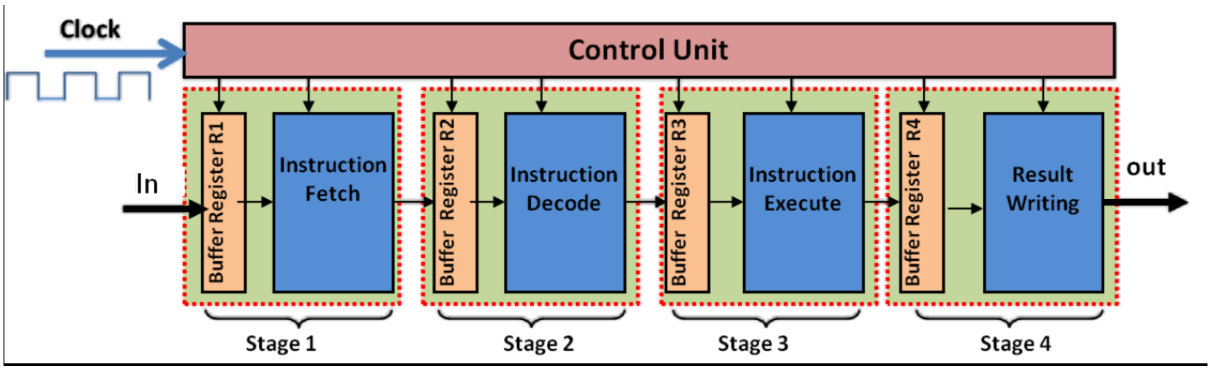
LIMITATIONS

• Only single convolution layer accelerated
• Limited BRAM for large models

- Fixed-point precision tradeoff
- Static weight embedding







FUTURE WORK

- Multi-layer CNN acceleration
- DMA-based high-speed transfer
- Multi-channel convolution
- Quantization-aware training
- Real-time camera streaming

CONCLUSION

This project successfully demonstrates the transformation of a high-level CNN convolution algorithm into a hardware-accelerated implementation using FPGA and Vitis HLS. Spatial parallelism, pipelining, and loop unrolling significantly reduce inference latency compared to CPU execution.

The work validates FPGA as an efficient and scalable solution for edge AI acceleration.

CODES:

1)train_intruder_model.py

```
import cv2

import os

import numpy as np

from sklearn.svm import LinearSVC

from sklearn.model_selection import train_test_split

from sklearn.metrics import classification_report
```

```
import joblib

def extract_hog(img):
    winSize = (64,64)
    blockSize = (16,16)
    blockStride = (8,8)
    cellSize = (8,8)
    nbins = 9

    hog = cv2.HOGDescriptor(winSize,blockSize,blockStride,cellSize,nbins)
    return hog.compute(img).flatten()

X=[]
y=[]

for label,folder in enumerate(["empty","person"]):
    path="dataset/"+folder
    for file in os.listdir(path):
        img=cv2.imread(os.path.join(path,file),0)
        feat=extract_hog(img)
        X.append(feat)
        y.append(label)

X=np.array(X)
y=np.array(y)
```

```
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_state=4
2)


model=LinearSVC()
model.fit(X_train,y_train)


pred=model.predict(X_test)
print(classification_report(y_test,pred))


joblib.dump(model,"intruder_model.pkl")
print("Model saved as intruder_model.pkl")
```

2) img_to_c.py

```
import cv2
import numpy as np
img = cv2.imread(r"C:\Users\Hartik
Rai\OneDrive\Desktop\ARM_Project\code\test.jpg",0)


if img is None:
   print("Image not found")
   exit()


img = cv2.resize(img,(64,64))


flat = img.flatten()


with open("test_img.h","w") as f:
   f.write("unsigned char input_img[4096] = {\n")
```

```
    for i,val in enumerate(flat):
        f.write(str(int(val)))
        if i!=len(flat)-1:
            f.write(",")
        if i%32==0:
            f.write("\n")
    f.write("\n};")

print("Header file created")
```

3) export_model.py

```
import joblib
import numpy as np

model = joblib.load("intruder_model.pkl")

w = model.coef_[0]
b = model.intercept_[0]

np.savetxt("weights.txt", w)
with open("bias.txt","w") as f:
    f.write(str(b))

print("Exported weights and bias")
print("Features:", len(w))
```

4) generate_c_header.py

```
import numpy as np
```

```
w = np.loadtxt("weights.txt")
b = float(open("bias.txt").read())

with open("model_params.h","w") as f:
    f.write("#ifndef MODEL_PARAMS_H\n#define MODEL_PARAMS_H\n\n")

    f.write(f"#define FEATURE_SIZE {len(w)}\n\n")

    f.write("float weights[FEATURE_SIZE] = {\n")
    for i,val in enumerate(w):
        f.write(f"{val}f")
        if i != len(w)-1:
            f.write(",")
        if i%8==0:
            f.write("\n")
    f.write("\n};\n\n")

    f.write(f"float bias = {b}f;\n")

    f.write("\n#endif\n")

print("model_params.h generated")
```

5) generate_c_header.py

```
import numpy as np
```

```
w = np.loadtxt("weights.txt")
b = float(open("bias.txt").read())

with open("model_params.h","w") as f:
   f.write("#ifndef MODEL_PARAMS_H\n#define MODEL_PARAMS_H\n\n")

   f.write(f"#define FEATURE_SIZE {len(w)}\n\n")

   f.write("float weights[FEATURE_SIZE] = {\n")
   for i,val in enumerate(w):
      f.write(f"{val}f")
      if i != len(w)-1:
         f.write(",")
      if i%8==0:
         f.write("\n")
   f.write("\n};\n\n")

   f.write(f"float bias = {b}f;\n")

   f.write("\n#endif\n")

print("model_params.h generated")
```

6)intruder_live_test.py

```
import cv2
import os

import time

import joblib

import numpy as np


# Load trained model

from intruder_pipeline import preprocess

model = joblib.load("intruder_model.pkl")


# HOG extractor (same as training)

def extract_hog(img):

    winSize = (64,64)

    blockSize = (16,16)

    blockStride = (8,8)

    cellSize = (8,8)

    nbins = 9

    hog = cv2.HOGDescriptor(winSize,blockSize,blockStride,cellSize,nbins)

    return hog.compute(img).flatten()


# Choose folder to simulate camera

STREAM_FOLDER = "dataset/person"   # change to empty to test


files = os.listdir(STREAM_FOLDER)


while True:

    for f in files:
```

```
path = os.path.join(STREAM_FOLDER,f)


frame = cv2.imread(path,0)

frame = preprocess(frame)


if frame is None:

   continue


start = time.perf_counter()


feat = extract_hog(frame).reshape(1,-1)

score = model.decision_function(feat)[0]


# safety margin

THRESHOLD = 0.8


if score > THRESHOLD:

   pred = 1

else:

   pred = 0



elapsed = time.perf_counter() - start

fps = 1/elapsed if elapsed > 0 else 0
```

```
        frame = cv2.resize(frame,(256,256))


        if pred == 1:
            text="HUMAN DETECTED"
            color=255
        else:
            text="SAFE"
            color=255


        cv2.putText(frame,text,(10,30),cv2.FONT_HERSHEY_SIMPLEX,0.8,color,2)

cv2.putText(frame,f"FPS:{fps:.2f}",(10,60),cv2.FONT_HERSHEY_SIMPLEX,0.7,col
or,2)
        print(text, " | FPS:", round(fps,2))
        time.sleep(0.2)


        # cv2.imshow("Intruder Detection",frame)


        # if cv2.waitKey(30)==27:
        #     break
```

7)intruder_pipeline.py

```
import cv2
import numpy as np


def preprocess(frame):


    frame = cv2.resize(frame,(64,64))
    blur = cv2.GaussianBlur(frame,(5,5),0)

    sobelx = cv2.Sobel(blur,cv2.CV_32F,1,0,ksize=3)
    sobely = cv2.Sobel(blur,cv2.CV_32F,0,1,ksize=3)

    mag = cv2.magnitude(sobelx,sobely)

    # normalize instead of raw edges
    mag = cv2.normalize(mag,None,0,255,cv2.NORM_MINMAX)
    mag = np.uint8(mag)

    return mag
```

8) prepare_dataset.py

```
import os
import shutil
import cv2
import numpy as np
from intruder_pipeline import preprocess
IMG_DIR = "images"
LBL_DIR = "labels"

OUT_PERSON = "dataset/person"
OUT_EMPTY = "dataset/empty"

os.makedirs(OUT_PERSON, exist_ok=True)
os.makedirs(OUT_EMPTY, exist_ok=True)

for img_name in os.listdir(IMG_DIR):

    if not img_name.lower().endswith(('.png','.jpg','.jpeg')):
        continue

    img_path = os.path.join(IMG_DIR, img_name)

    # corresponding label file
    label_name = os.path.splitext(img_name)[0] + ".txt"
    label_path = os.path.join(LBL_DIR, label_name)

    # check if label exists and has content
    has_person = False
    if os.path.exists(label_path):
        with open(label_path,'r') as f:
            content = f.read().strip()
            if content != "":
                has_person = True

    # read image
    img = cv2.imread(img_path)
    if img is None:
        continue
    # convert to grayscale 64x64 (VERY IMPORTANT)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    processed = preprocess(gray)



    # SAME preprocessing as runtime
    blur = cv2.GaussianBlur(processed,(5,5),0)
    sobelx = cv2.Sobel(blur,cv2.CV_64F,1,0,ksize=3)
    sobely = cv2.Sobel(blur,cv2.CV_64F,0,1,ksize=3)
    mag = cv2.magnitude(sobelx,sobely)
    processed = np.uint8(np.clip(mag,0,255))

    # save
    if has_person:
        cv2.imwrite(os.path.join(OUT_PERSON,img_name),processed)
    else:
        cv2.imwrite(os.path.join(OUT_EMPTY,img_name),processed)

print("Dataset prepared successfully!")
```

9)tempCodeRunnerFile.py

```
import cv2
import os
import time
import joblib
import numpy as np

# Load trained model
from intruder_pipeline import preprocess
model = joblib.load("intruder_model.pkl")

# HOG extractor (same as training)
def extract_hog(img):
    winSize = (64,64)
    blockSize = (16,16)
    blockStride = (8,8)
    cellSize = (8,8)
    nbins = 9
    hog = cv2.HOGDescriptor(winSize,blockSize,blockStride,cellSize,nbins)
    return hog.compute(img).flatten()

# Choose folder to simulate camera
STREAM_FOLDER = "dataset/empty"   # change to empty to test

files = os.listdir(STREAM_FOLDER)

while True:
    for f in files:
        path = os.path.join(STREAM_FOLDER,f)

        frame = cv2.imread(path,0)
        frame = preprocess(frame)

        if frame is None:
            continue

        start = time.perf_counter()

        feat = extract_hog(frame).reshape(1,-1)
        score = model.decision_function(feat)[0]

        # safety margin
        THRESHOLD = 0.8

        if score > THRESHOLD:
            pred = 1
        else:
            pred = 0



        elapsed = time.perf_counter() - start
        fps = 1/elapsed if elapsed > 0 else 0



        frame = cv2.resize(frame,(256,256))

        if pred == 1:
```

```
            text="HUMAN DETECTED"
            color=255
        else:
            text="SAFE"
            color=255

        cv2.putText(frame,text,(10,30),cv2.FONT_HERSHEY_SIMPLEX,0.8,color,2)

cv2.putText(frame,f"FPS:{fps:.2f}",(10,60),cv2.FONT_HERSHEY_SIMPLEX,0.7,col
or,2)
        print(text, " | FPS:", round(fps,2))
        time.sleep(0.2)

        # cv2.imshow("Intruder Detection",frame)

        # if cv2.waitKey(30)==27:
        #     break
```