FSK CNN MODEM

This document presents a complete implementation journey of a Convolutional Neural Network (CNN) based Frequency Shift Keying (FSK) demodulator, from initial concept through hardware deployment on embedded FPGA platforms. The work follows the methodology described in the IEEE article available at https://ieeexplore.ieee.org/document/10334792, which demonstrates that deep learning techniques can achieve competitive performance with traditional signal processing approaches while maintaining minimal computational complexity.

Traditional FSK demodulation relies on established digital signal processing techniques such as matched filtering, coherent detection, or non-coherent energy detection. While these methods are well-understood and mathematically rigorous, they often require significant computational resources and careful tuning of parameters such as carrier recovery loops and timing synchronization circuits.

Recent advances in machine learning have shown that neural networks can learn optimal detection strategies directly from data, potentially simplifying receiver architectures while maintaining or even improving performance. The challenge lies in creating networks compact enough for resource-constrained embedded systems while achieving bit error rates comparable to classical approaches.

This project aims to:

- Implement and train an ultra-compact CNN architecture with only 12 trainable parameters
- Achieve bit error rate performance below 10^{-6} at 15dB signal-to-noise ratio
- Deploy the trained model on FPGA hardware using multiple implementation approaches
- Validate end-to-end functionality from Python training through hardware execution
- Demonstrate the practical viability of neural network-based demodulation in embedded systems

A critical aspect of this design is the use of intentional undersampling. Rather than sampling FSK signals at rates well above the Nyquist criterion, the system deliberately undersamples 19-21 MHz FSK signals, causing them to alias down to baseband. This approach dramatically reduces the required sampling rate and subsequent processing complexity, while the CNN learns to correctly classify the aliased in-phase (I) and quadrature (Q) components.

DOCUMENT LAYOUT

This documentation follows the complete implementation workflow:

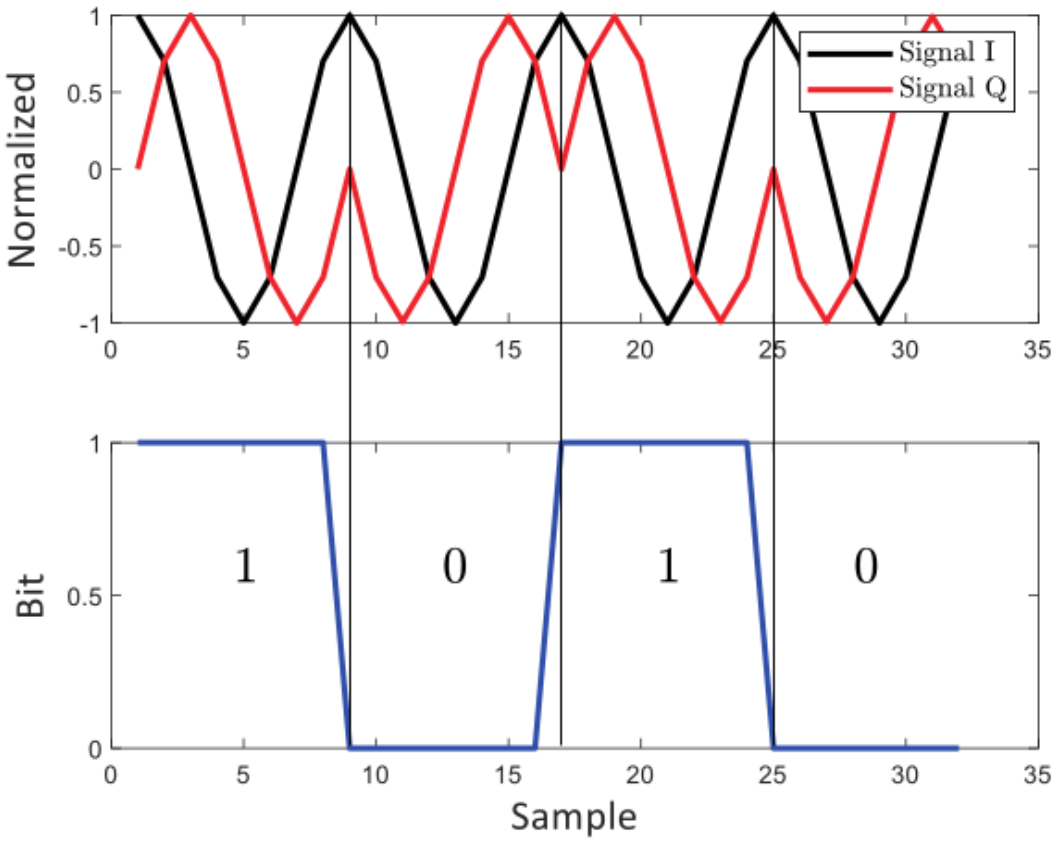
1. **Python Training**: Development of synthetic dataset generation, neural network architecture definition, training procedure, and parameter quantization
2. **Python Evaluation**: Validation scripts demonstrating demodulation performance with both floating-point and quantized models
3. **HLS Implementation**: Translation of the trained CNN into synthesizable C++ using Xilinx Vitis HLS with AXI-Stream interfaces
4. **Vivado Integration**: Creation of comprehensive testbenches using AXI VIP components for hardware verification
5. **Vitis AI Deployment**: Quantization and compilation for deployment on the Deep Learning Processing Unit (DPU) of a Zynq UltraScale+ platform

Each section provides detailed technical information, code implementations, and verification results, creating a complete reference for practitioners interested in deploying neural networks on FPGA hardware.

FSK CNN MODEM

In this modem, we'll see how to implement a CNN based FSK demodulator, as the one described in the IEE article:

https://ieeexplore.ieee.org/document/10334792



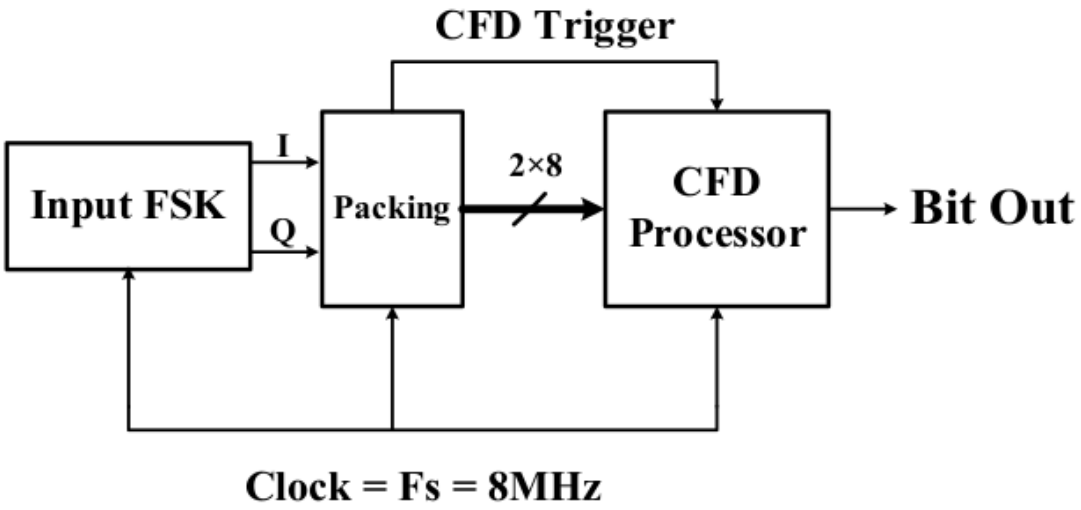


Fig. 4. CNN-Based FSK demodulator architecture

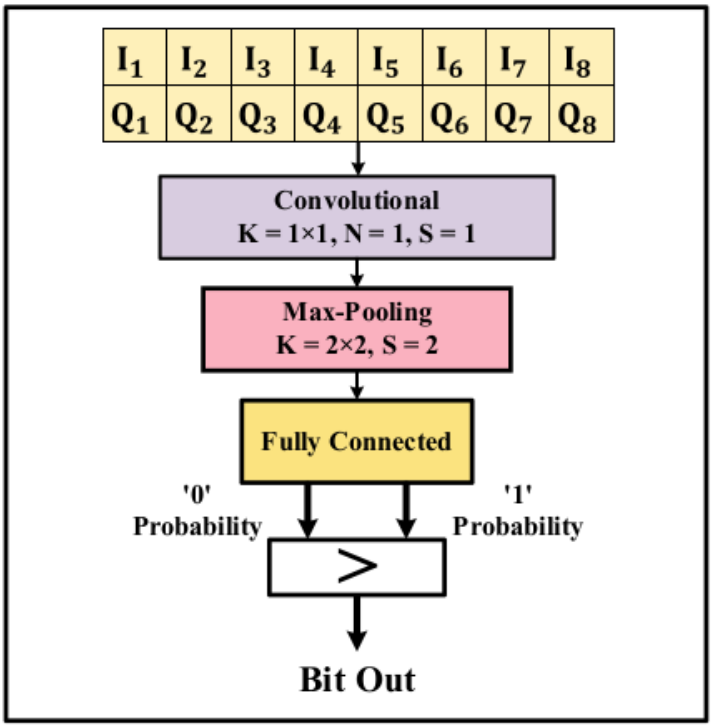


Fig. 5. The proposed convolutional neural network for data classification.

This paper presents an **ultra-compact CNN-based FSK demodulator** with only 12 trainable parameters that achieves comparable performance to ideal demodulators (BER < 10^{-6} at 15dB) while using minimal FPGA resources (970 LUTs, 751 FFs). The system uses intentional undersampling to alias 19-21 MHz FSK signals down to baseband, where a simple 3-layer CNN classifies each symbol as bit 0 or 1.

PYTHON TRAINING SCRIPT

A python script has been created to create the CNN described in the article, to create a dataset and to train and save weights as well:

Code Blame 337 lines (253 loc) · 11.2 KB

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
import numpy as np

# Set random seeds for reproducibility
torch.manual_seed(42)
np.random.seed(42)

# FSK Signal Parameters
F_CARRIER = 20e6  # 20 MHz
F_BITRATE = 1e6   # 1 Mbps
F_DEVIATION = 1e6  # 1 MHz
F_SAMPLING = 8e6  # 8 MHz
SAMPLES_PER_SYMBOL = int(F_SAMPLING / F_BITRATE)  # 8 samples

# Training Parameters
TRAIN_SAMPLES = 200000  # 2e5
VAL_SAMPLES = 50000     # 5e4
BATCH_SIZE = 256
EPOCHS = 50

class FSKDataset(Dataset):
    """Generate FSK symbols with AWGN noise"""

    def __init__(self, num_samples, eb_n0_range=(0, 25)):
        self.num_samples = num_samples
        self.eb_n0_min, self.eb_n0_max = eb_n0_range

    def __len__(self):
        return self.num_samples

    def generate_fsk_symbol(self, bit, eb_n0_db):
        """Generate one FSK symbol (I/Q components)"""
        # Frequency for bit 0: 19MHz, bit 1: 21MHz
        freq = F_CARRIER + (F_DEVIATION if bit == 1 else -F_DEVIATION)

        # Time vector for one symbol
        t = np.arange(SAMPLES_PER_SYMBOL) / F_SAMPLING

        # Generate I and Q components (baseband after aliasing)
        # Due to undersampling, the signal appears at baseband
        baseband_freq = freq % F_SAMPLING
        if baseband_freq > F_SAMPLING / 2:
            baseband_freq = F_SAMPLING - baseband_freq

        # Adjust for the actual aliased frequencies
        if bit == 0:
            phase_shift = 2 * np.pi * (-F_DEVIATION) / F_SAMPLING
        else:
            phase_shift = 2 * np.pi * F_DEVIATION / F_SAMPLING

        I = np.cos(phase_shift * np.arange(SAMPLES_PER_SYMBOL))
        Q = np.sin(phase_shift * np.arange(SAMPLES_PER_SYMBOL))

        # Add AWGN noise
        eb_n0_linear = 10 ** (eb_n0_db / 10)
        noise_power = 1 / (2 * eb_n0_linear)
        noise_std = np.sqrt(noise_power)

        I += np.random.normal(0, noise_std, SAMPLES_PER_SYMBOL)
        Q += np.random.normal(0, noise_std, SAMPLES_PER_SYMBOL)

        # Stack I and Q into 2x8 array
        symbol = np.stack([I, Q], axis=0)

```

This script implements the complete training, validation, and parameter export flow for a compact convolutional neural network designed to demodulate binary Frequency Shift Keying signals in the presence of additive white Gaussian noise. The objective is to train a minimal-complexity neural network that can reliably classify FSK symbols while being suitable for deployment on resource-constrained hardware such as an FPGA.

The script begins by defining the signal parameters used to synthesize the FSK waveform, including carrier frequency, bit rate, frequency deviation, sampling frequency, and the resulting number of samples per symbol. These parameters model an undersampled high-frequency FSK signal that aliases to baseband, producing in-phase and quadrature components that are later used as neural network inputs. Training parameters such as dataset size, batch size, and number of epochs are also defined to control the learning process.

A custom PyTorch dataset class is implemented to generate synthetic FSK symbols on the fly. For each sample, a random bit value is selected and mapped to one of two frequency deviations around the carrier. The corresponding aliased baseband I/Q signal is generated over a single symbol period and corrupted with additive white Gaussian noise based on a randomly chosen Eb/N0 value within a specified range. Each dataset sample therefore consists of a two-channel time-domain representation of one noisy FSK symbol and its associated bit label, allowing the network to learn robustly across varying signal-to-noise conditions.

The neural network architecture is deliberately minimal and consists of a single one-dimensional convolutional layer, followed by max pooling and a fully connected classification layer. The convolution operates across the I/Q channels with a kernel size of one, effectively learning a linear combination of the two components. After temporal downsampling via max pooling, the resulting features are flattened and mapped to two output classes corresponding to binary symbol decisions. The total number of trainable parameters is extremely small, making the model well suited for hardware implementation.

Separate functions are provided to perform training and validation over one epoch. During training, the model processes batches of synthetic symbols, computes the cross-entropy loss, and updates the parameters using the NAdam optimizer. Validation is performed without gradient updates to measure classification accuracy and loss on an independent dataset. The script tracks the best validation accuracy achieved during training and saves the corresponding model weights to disk.

After training completes, the script reloads the best performing model and extracts its parameters for deployment. The floating-point weights and biases are saved both in NumPy binary format and in a human-readable text file to facilitate inspection and documentation. A quantization routine is then applied to convert the trained parameters into a fixed-point representation with a configurable number of bits. This process computes per-parameter scaling based on the observed dynamic range and produces quantized integer values suitable for FPGA implementation.

Finally, the script saves the quantized parameters in both machine-readable and text formats and prints a summary of the parameter ranges before and after quantization. The complete workflow therefore covers signal generation, neural network training, model selection, and parameter export, providing a direct path from algorithm development in Python to deployment in fixed-point digital hardware.

This script has been used in a virtual environment.

For this purpose, the following bash script has been written:

FSK_CNN_MODEM / PYTHON / setup_env.sh

caccolillo Added a venv creation bash script and a training script.

Code Blame 37 lines (29 loc) · 1.12 KB

```bash
#!/bin/bash

echo "=== FSK CNN Training Environment Setup ==="
echo ""

# Create virtual environment
echo "Creating virtual environment..."
python3 -m venv fsk_cnn_env

# Activate virtual environment
echo "Activating virtual environment..."
source fsk_cnn_env/bin/activate

# Upgrade pip
echo "Upgrading pip..."
pip install --upgrade pip

# Install PyTorch 1.11 with CUDA 11.3
echo "Installing PyTorch 1.11 with CUDA 11.3..."
pip install torch==1.11.0+cu113 torchvision==0.12.0+cu113 torchaudio==0.11.0 --extra-index-url https://download.pytorch.org/whl/cu113

# Install numpy
echo "Installing numpy..."
pip install numpy

# Verify installation
echo ""
echo "=== Verifying Installation ==="
python -c "import torch; print(f'PyTorch: {torch.__version__}'); print(f'CUDA available: {torch.cuda.is_available()}'); print(f'CUDA device: {torch.cuda.get_device_name(0) if torch.cuda.is_available() else \"N/A\"}')"

echo ""
echo "=== Setup Complete! ==="
echo "Virtual environment is activated."
echo "You can now run: python train_fsk_cnn.py"
echo ""
echo "To deactivate later, run: deactivate"
echo "To reactivate, run: source fsk_cnn_env/bin/activate"
```

A Python virtual environment, commonly created using venv, is an isolated workspace that contains its own Python interpreter and its own set of installed packages. Its purpose is to decouple a project's dependencies from the system-wide Python installation and from other projects on the same machine. This avoids version conflicts, makes experiments reproducible, and ensures that a specific combination of libraries can be used without affecting or being affected by other software. In practice, a virtual environment behaves like a lightweight container for Python packages: once activated, all python and pip commands operate only within that environment.

The provided bash script automates the creation and configuration of such an isolated environment specifically for training the FSK CNN model. The script starts by printing informative messages to make it clear what stage of the setup process is currently running. It then creates a new virtual environment named fsk_cnn_env using the system's Python 3 interpreter. This step generates a local directory containing a private Python executable and package manager.

After the environment is created, the script activates it. Activation modifies the current shell session so that the python and pip commands point to the virtual environment

rather than the system installation. From this point onward, any Python packages installed are confined to fsk_cnn_env and do not interfere with other projects.

Once the environment is active, the script upgrades pip to the latest version available. This ensures compatibility with modern Python packages and avoids installation issues caused by outdated tooling. The script then installs a specific version of PyTorch together with its associated vision and audio libraries, explicitly targeting CUDA 11.3. This guarantees that the neural network training code runs with a known and tested PyTorch release and can take advantage of GPU acceleration if a compatible NVIDIA GPU is present. By pinning exact versions, the script ensures reproducibility across different machines and over time.

The script also installs NumPy, which is required for numerical operations and signal generation in the training code. After installing all dependencies, a short verification step is executed. This step runs a small Python command that imports PyTorch, prints its version, checks whether CUDA is available, and, if so, reports the name of the detected GPU device. This provides immediate confirmation that the software stack is correctly installed and that GPU acceleration is functioning as expected.

Finally, the script prints a summary indicating that the setup is complete and that the virtual environment is active. It reminds the user how to run the training script within this environment and explains how to deactivate and later reactivate the virtual environment. Overall, this script establishes a clean, reproducible Python environment tailored to the FSK CNN training workflow and ensures that all required dependencies are installed and verified before execution.

For training, we are going to use a CUDA GPU:

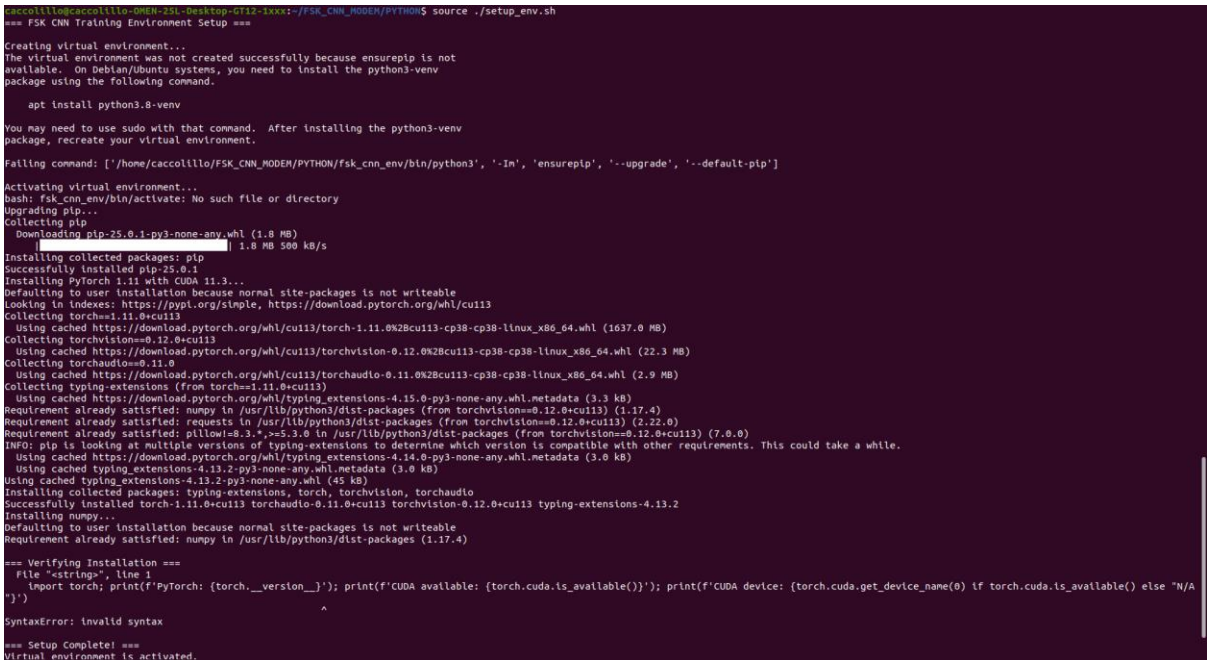
```
caccolillo@caccolillo-OMEN-25L-Desktop-GT12-1xxx:~$ nvidia-smi
Mon Jan  5 17:55:38 2026
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.183.01             Driver Version: 535.183.01   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA GeForce RTX 3060        Off | 00000000:01:00.0  On |                  N/A |
| 35%   25C    P8              12W / 170W |    306MiB / 12288MiB |      5%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|    0   N/A  N/A      1323      G   /usr/lib/xorg/Xorg                           35MiB |
|    0   N/A  N/A      2364      G   /usr/lib/xorg/Xorg                           89MiB |
|    0   N/A  N/A      2491      G   /usr/bin/gnome-shell                         78MiB |
|    0   N/A  N/A      4418      G   ...cess-track-uuid=3190708988185955192       85MiB |
+---------------------------------------------------------------------------------------+
caccolillo@caccolillo-OMEN-25L-Desktop-GT12-1xxx:~$
```

nvidia-smi is a command-line tool provided by NVIDIA to monitor and manage NVIDIA GPUs. It communicates directly with the GPU driver, so it works even if the CUDA toolkit is not installed. It is commonly used to verify that the NVIDIA driver is correctly installed and that the GPU is available.

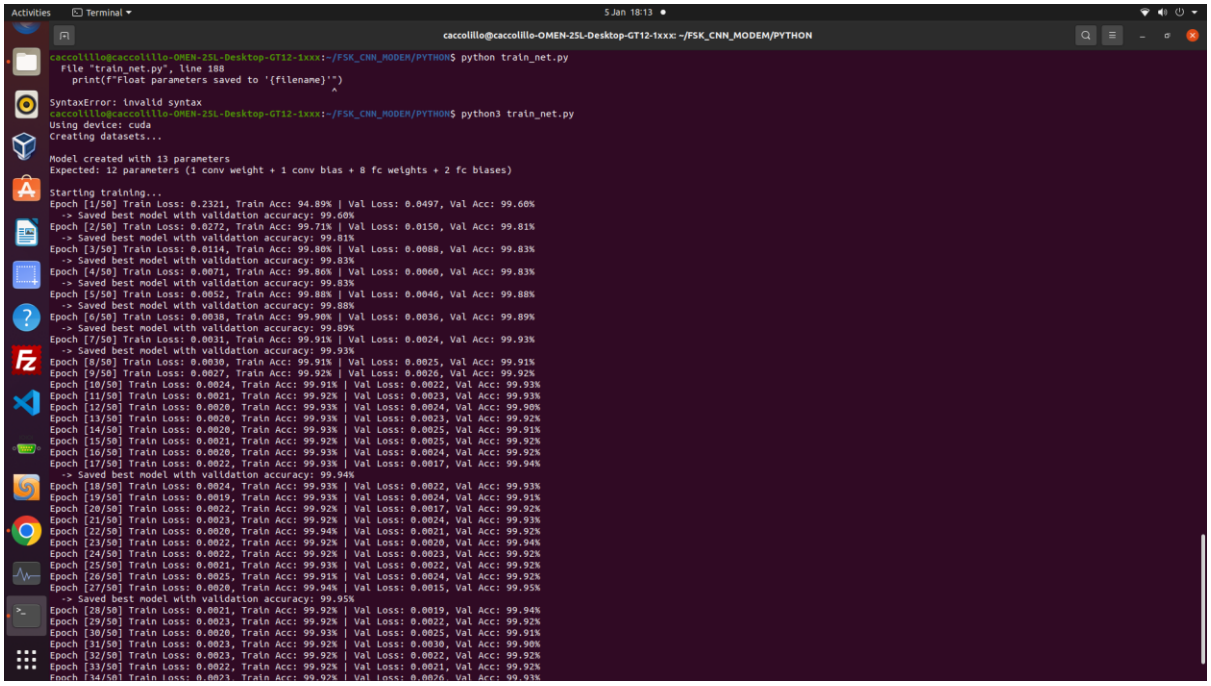
The command displays key information such as the GPU model, driver version, and the maximum CUDA version supported by the driver. It also shows real-time data including temperature, power usage, memory usage, and GPU utilization, which is useful when running CUDA or deep learning workloads.

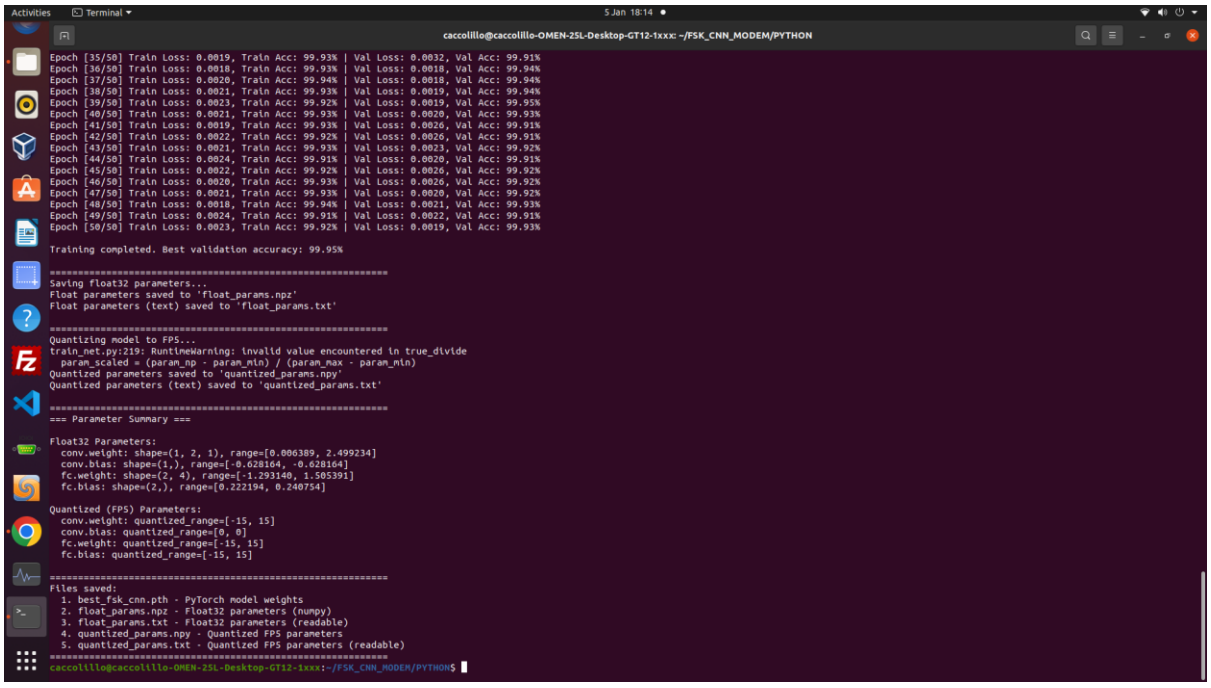
nvidia-smi can list the processes currently using the GPU and how much memory they consume, making it helpful for debugging resource conflicts or memory issues. Overall, it is a primary diagnostic and monitoring tool for CUDA-enabled systems.

By launching the bash script:

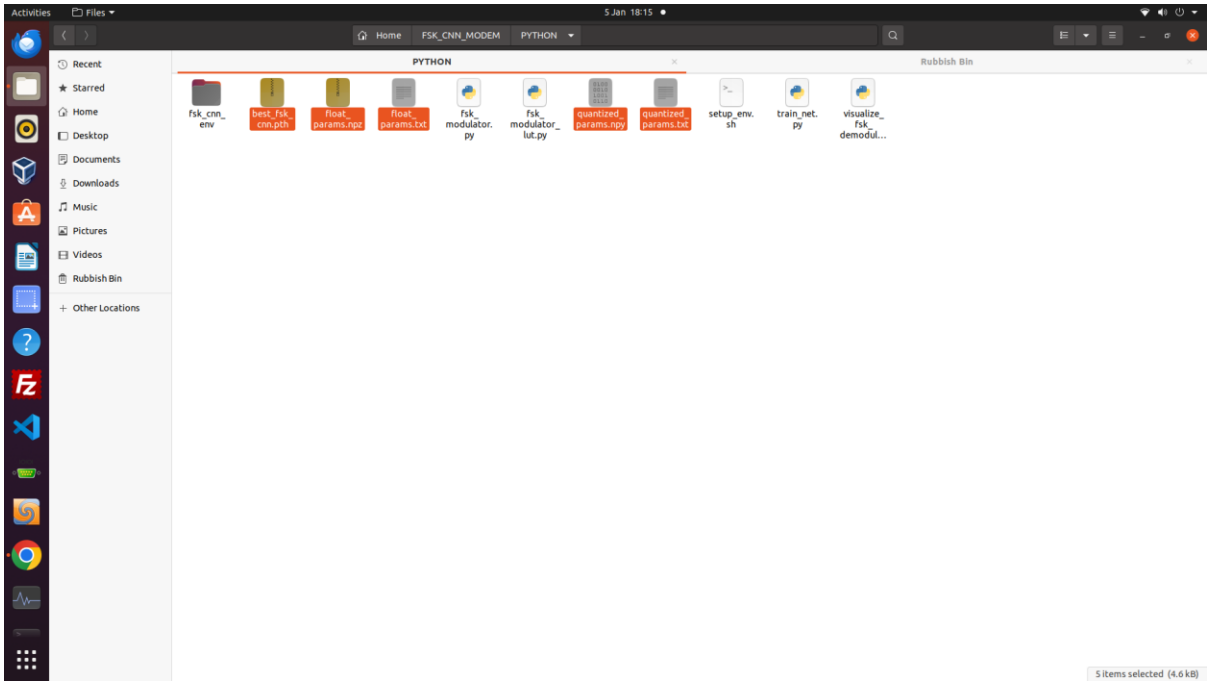


Once the virtual environment is ready, now we can launch the training script:





At the end of wich, we get the network parameters we are after:



Open

float_params.txt
~/FSK_CNN_MODEM/PYTHON

```
=== Float32 Parameters ===

conv.weight:
  Shape: (1, 2, 1)
  Values: [[[0.00638873]
  [2.4992344 ]]]

conv.bias:
  Shape: (1,)
  Values: [-0.62816364]

fc.weight:
  Shape: (2, 4)
  Values: [[-0.10768079 -1.1741734   0.3632769   1.4843116 ]
 [ 0.86062443  1.5053908  -0.62919515 -1.29314   ]]

fc.bias:
  Shape: (2,)
  Values: [0.22219384 0.24075383]

```

PYTHON CNN EVALUATION SCRIPT

An ad-hoc python script has been written to evaluate the trained CNN network:

FSK_CNN_MODEM / PYTHON / **visualize_fsk_demodulation.py**

caccolillo Added a script to load the training parameters and evaluate the model...

Code | Blame | 286 lines (225 loc) · 9.97 KB

```python
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt

# FSK Signal Parameters
F_CARRIER = 20e6  # 20 MHz
F_BITRATE = 1e6   # 1 Mbps
F_DEVIATION = 1e6  # 1 MHz
F_SAMPLING = 8e6  # 8 MHz
SAMPLES_PER_SYMBOL = int(F_SAMPLING / F_BITRATE)  # 8 samples

class CNNFSK(nn.Module):
    """Compact CNN for FSK demodulation - only 12 parameters"""

    def __init__(self):
        super(CNNFSK, self).__init__()

        # Convolutional layer: 1x1 kernel, 1 filter, stride=1
        self.conv = nn.Conv1d(in_channels=2, out_channels=1,
                              kernel_size=1, stride=1, bias=True)

        # Max pooling: 2x2 window, stride=2
        self.maxpool = nn.MaxPool1d(kernel_size=2, stride=2)

        # Fully connected layer: 4 inputs, 2 outputs (bit 0 or 1)
        self.fc = nn.Linear(4, 2)

    def forward(self, x):
        # x shape: (batch, 2, 8)
        x = self.conv(x)  # (batch, 1, 8)
        x = self.maxpool(x)  # (batch, 1, 4)
        x = x.view(x.size(0), -1)  # (batch, 4)
        x = self.fc(x)  # (batch, 2)
        return x


def generate_fsk_symbol(bit, eb_n0_db):
    """Generate one FSK symbol (I/Q components) with AWGN noise"""
    # Frequency for bit 0: 19MHz, bit 1: 21MHz
    freq = F_CARRIER + (F_DEVIATION if bit == 1 else -F_DEVIATION)

    # Adjust for the actual aliased frequencies at baseband
    if bit == 0:
        phase_shift = 2 * np.pi * (-F_DEVIATION) / F_SAMPLING
    else:
        phase_shift = 2 * np.pi * F_DEVIATION / F_SAMPLING

    # Generate I and Q components
    I = np.cos(phase_shift * np.arange(SAMPLES_PER_SYMBOL))
```

This script implements a complete end-to-end demonstration of a convolutional neural network–based Frequency Shift Keying demodulator, focusing on signal generation, inference, performance inspection, and visualization. It is intended to validate and illustrate the behavior of a previously trained CNN model when applied to noisy FSK signals under controlled signal-to-noise conditions.

The script begins by defining the fundamental FSK signal parameters, including carrier frequency, bit rate, frequency deviation, sampling frequency, and the resulting number of samples per symbol. These parameters model a high-frequency FSK signal that is intentionally undersampled so that it aliases to baseband, allowing the in-phase and quadrature components to be directly processed by the neural network.

A compact convolutional neural network architecture is defined to perform binary FSK demodulation. The model consists of a single one-dimensional convolutional layer that

combines the I and Q components, followed by max pooling to reduce temporal resolution and a fully connected layer that produces class scores for bit values zero and one. The architecture is deliberately minimal, containing only a small number of trainable parameters, and mirrors the model used during training to ensure compatibility with previously saved weights.

The script includes a signal generation function that synthesizes individual FSK symbols based on a randomly selected bit value. Each symbol is generated by applying a phase increment corresponding to the selected frequency deviation and constructing cosine and sine waveforms for the I and Q components. Additive white Gaussian noise is then applied according to a specified Eb/N0 value, allowing realistic channel conditions to be simulated. These noisy I/Q samples are stacked into a tensor format suitable for neural network inference.

To simulate a digital transmission, a sequence generation function produces a random bit stream and generates a corresponding sequence of FSK symbols. In addition to the symbol tensors used by the CNN, the function also concatenates the I and Q samples into continuous time-domain sequences. This enables time-based visualization of the modulated signal and facilitates direct comparison between the original and demodulated bit streams.

The demodulation process is performed by passing the generated symbols through the trained CNN in evaluation mode. The model outputs class logits for each symbol, which are converted into probabilities using a softmax function. The predicted bit values are extracted by selecting the class with the highest probability, and both the hard decisions and confidence levels are returned for further analysis.

A dedicated visualization function generates a comprehensive set of plots that illustrate the entire demodulation process. These plots include the original digital bit sequence, the in-phase and quadrature waveforms over time, the I/Q constellation diagram, and the demodulated bit sequence overlaid with confidence information. This visual representation provides insight into the relationship between the time-domain signal, the constellation structure, and the neural network's decision process. The resulting figure is saved to disk and displayed for interactive inspection.

To complement the graphical output, the script also prints a detailed bit-by-bit comparison between the transmitted and demodulated sequences. For each symbol, the original bit, predicted bit, class probabilities, and correctness are displayed, followed by a summary of the total error count and overall demodulation accuracy. This textual output enables precise performance evaluation and debugging.

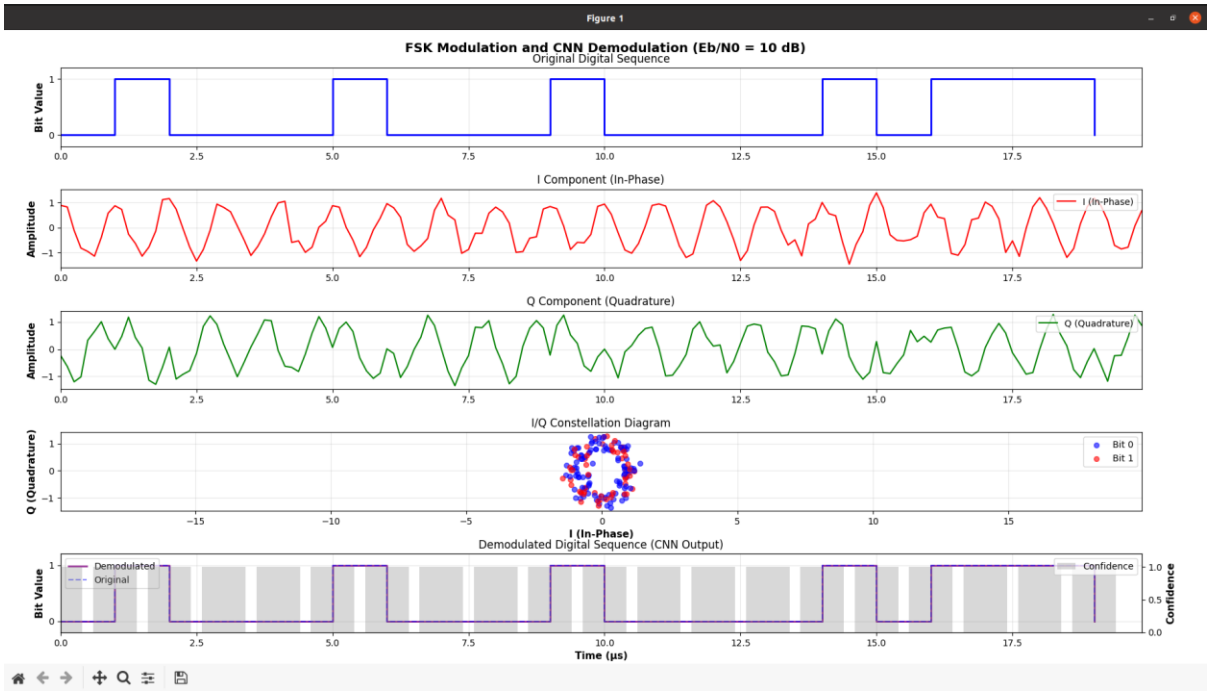
The main execution flow configures the simulation parameters, selects the computation device, and loads the trained model weights from disk. If the trained model file is not found, the script exits gracefully and informs the user that training must be performed first. Once the model is loaded, a random bit sequence is generated, demodulated using the CNN, and analyzed using both numerical and visual methods.

Overall, this script serves as a verification and demonstration tool for the CNN-based FSK demodulator. It bridges the gap between abstract model performance metrics and physical signal behavior by combining realistic signal generation, neural network inference, detailed accuracy reporting, and intuitive visualizations, making it particularly useful for algorithm validation, presentations, and hardware-oriented design workflows.

```
caccolillo@caccolillo-OMEN-25L-Desktop-GT12-1xxx:~/FSK_CNN_MODEM/PYTHON$ python3 visualize_fsk_demodulation.py
======================================================================
FSK CNN DEMODULATOR - VISUALIZATION SCRIPT
======================================================================

Using device: cuda

Loading trained model...
✓ Model loaded successfully from 'best_fsk_cnn.pth'

Model has 13 parameters

Generating 20 random bits with FSK modulation...
Eb/N0 = 10 dB
Carrier Frequency = 20.0 MHz
Bit 0 Frequency = 19.0 MHz
Bit 1 Frequency = 21.0 MHz
Sample Rate = 8.0 MHz
Samples per Symbol = 8

Original bit sequence:
01000100010000101110

Demodulating using CNN...

Demodulated bit sequence:
01000100010000101110

======================================================================
BIT-BY-BIT COMPARISON
======================================================================
Bit #    Original     Demodulated     Prob[0]      Prob[1]      Match
----------------------------------------------------------------------
0        0            0               1.0000       0.0000       ✓
1        1            1               0.0000       1.0000       ✓
2        0            0               1.0000       0.0000       ✓
3        0            0               1.0000       0.0000       ✓
4        0            0               1.0000       0.0000       ✓
5        1            1               0.0000       1.0000       ✓
6        0            0               1.0000       0.0000       ✓
7        0            0               1.0000       0.0000       ✓
8        0            0               1.0000       0.0000       ✓
9        1            1               0.0000       1.0000       ✓
10       0            0               0.9995       0.0005       ✓
11       0            0               1.0000       0.0000       ✓
12       0            0               1.0000       0.0000       ✓
13       0            0               1.0000       0.0000       ✓
14       1            1               0.0000       1.0000       ✓
15       0            0               0.9996       0.0004       ✓
16       1            1               0.0000       1.0000       ✓
17       1            1               0.0000       1.0000       ✓
18       1            1               0.0000       1.0000       ✓
19       0            0               1.0000       0.0000       ✓
----------------------------------------------------------------------
Total Bits: 20
Errors: 0
```

```
------------------------------------------------------------------------
Total Bits: 20
Errors: 0
Accuracy: 100.00%
========================================================================

Generating visualization...
Visualization saved as 'fsk_demodulation_visualization.png'
```

The following summary picture gets generated at the end of the script:



This proves the correct functioning of the CNN, even in the presence of noise, as claimed in the article.

Another python script has been prepared to test the effectiveness of the quantized version too:

FSK_CNN_MODEM / PYTHON / **visualize_fsk_demodulation_quantized.py**

caccolillo Added python inference script using quantized coefficients.

Code | Blame | 87 lines (72 loc) · 2.94 KB

```python
import numpy as np

# === 1. Load quantized parameters ===
quantized_params = np.load('quantized_params.npy', allow_pickle=True).item()
print("Loaded quantized parameters from 'quantized_params.npy'")

# === 2. Tiny FP5 CNN inference function ===
def cnn_fsk_fp5(symbol, quantized_params):
    conv_w = quantized_params['conv.weight']['quantized']  # (1,2,1)
    conv_b = quantized_params['conv.bias']['quantized']    # (1,)
    scale_w = quantized_params['conv.weight']['scale']
    scale_b = quantized_params['conv.bias']['scale']

    conv_out = np.zeros(8, dtype=np.float32)
    for t in range(8):
        conv_sum = 0
        for i in range(2):
            conv_sum += symbol[i, t] * conv_w[0, i, 0] * scale_w
        conv_sum += conv_b[0] * scale_b
        conv_out[t] = conv_sum

    # MaxPool1d
    pool_out = np.array([max(conv_out[0], conv_out[1]),
                         max(conv_out[2], conv_out[3]),
                         max(conv_out[4], conv_out[5]),
                         max(conv_out[6], conv_out[7])])

    # Fully connected
    fc_w = quantized_params['fc.weight']['quantized']  # (2,4)
    fc_b = quantized_params['fc.bias']['quantized']    # (2,)
    scale_w_fc = quantized_params['fc.weight']['scale']
    scale_b_fc = quantized_params['fc.bias']['scale']

    fc_out = np.zeros(2, dtype=np.float32)
    for j in range(2):
        s = 0
        for i in range(4):
            s += pool_out[i] * fc_w[j, i] * scale_w_fc
        s += fc_b[j] * scale_b_fc
        fc_out[j] = s

    # Softmax
    exp_out = np.exp(fc_out - np.max(fc_out))
    probs = exp_out / np.sum(exp_out)
    pred_bit = np.argmax(probs)

    return pred_bit, probs

# === 3. FSK Signal Parameters ===
SAMPLES_PER_SYMBOL = 8
```

It uses the quantized coefficients discussed so far:

Open quantized_params.txt ~/FSK_CNN_MODEM/PYTHON

```
=== Quantized Parameters (FP5) for FPGA Implementation ===

conv.weight:
  Quantized values (int8): [[[-15]
  [ 15]]]
  Min: 0.006389
  Max: 2.499234
  Scale: 0.083095

conv.bias:
  Quantized values (int8): [0]
  Min: -0.628164
  Max: -0.628164
  Scale: 0.000000

fc.weight:
  Quantized values (int8): [[ -2 -14   3  15]
 [  8  15  -8 -15]]
  Min: -1.293140
  Max: 1.505391
  Scale: 0.093284

fc.bias:
  Quantized values (int8): [-15  15]
  Min: 0.222194
  Max: 0.240754
  Scale: 0.000619

```

By running it, we can ensure there is no loss of performance:

caccolillo@caccolillo-OMEN-25L-Desktop-GT12-1xxx: ~/FSK_...

```
caccolillo@caccolillo-OMEN-25L-Desktop-GT12-1xxx:~/FSK_CNN_MODEM/PYTHON$ python3
 ./visualize_fsk_demodulation_quantized.py
Loaded quantized parameters from 'quantized_params.npy'

=== FP5 Inference Results ===
Symbol # Original Predicted  Correct  Probabilities
------------------------------------------------------------------------
1        0        0          ✓        [0.9984079  0.00159215]
2        1        1          ✓        [9.1444032e-04 9.9908555e-01]
3        0        0          ✓        [0.9984079  0.00159215]
4        1        1          ✓        [9.1444032e-04 9.9908555e-01]
5        0        0          ✓        [0.9984079  0.00159215]
6        0        0          ✓        [0.9984079  0.00159215]
7        1        1          ✓        [9.1444032e-04 9.9908555e-01]
8        1        1          ✓        [9.1444032e-04 9.9908555e-01]
9        0        0          ✓        [0.9984079  0.00159215]
10       0        0          ✓        [0.9984079  0.00159215]

=== Overall Statistics ===
Total symbols: 10
Correct predictions: 10
Errors: 0
Accuracy: 100.00%
caccolillo@caccolillo-OMEN-25L-Desktop-GT12-1xxx:~/FSK_CNN_MODEM/PYTHON$
```

Despite the quantization.

HLS CNN IMPLEMENTATION

The CNN, once trained, can be implemented in HLS:

Synthesis Summary(solution1) fsk_cnn_demod.cpp

```cpp
#include "fsk_cnn_demod.hpp"
#include <hls_math.h>

// Optimized helper functions for dataflow

static void read_iq_parallel(
    hls::stream<axis_data> &input_I,
    hls::stream<axis_data> &input_Q,
    fixed_t I_buffer[SAMPLES_PER_SYMBOL],
    fixed_t Q_buffer[SAMPLES_PER_SYMBOL]
) {
#pragma HLS INLINE off
    // Read I and Q samples in parallel with II=1
    READ_IQ: for (int i = 0; i < SAMPLES_PER_SYMBOL; i++) {
#pragma HLS PIPELINE II=1
        axis_data i_data = input_I.read();
        axis_data q_data = input_Q.read();
        I_buffer[i] = i_data.data;
        Q_buffer[i] = q_data.data;
    }
}

static void compute_convolution(
    fixed_t I_buffer[SAMPLES_PER_SYMBOL],
    fixed_t Q_buffer[SAMPLES_PER_SYMBOL],
    acc_t conv_out[SAMPLES_PER_SYMBOL]
) {
#pragma HLS INLINE off

    // Precompute scaled weights (constant folding optimization)
    const acc_t w_i = (acc_t)CONV_WEIGHT_QUANT[0][0] * CONV_WEIGHT_SCALE;
    const acc_t w_q = (acc_t)CONV_WEIGHT_QUANT[0][1] * CONV_WEIGHT_SCALE;
    const acc_t bias = (acc_t)CONV_BIAS_QUANT[0] * CONV_BIAS_SCALE;

    CONV: for (int t = 0; t < SAMPLES_PER_SYMBOL; t++) {
#pragma HLS PIPELINE II=1
#pragma HLS UNROLL factor=2
        acc_t sum = bias;
        sum += (acc_t)I_buffer[t] * w_i;
        sum += (acc_t)Q_buffer[t] * w_q;
        conv_out[t] = sum;
    }
}

static void compute_pooling(
    acc_t conv_out[SAMPLES_PER_SYMBOL],
    fixed_t pooled[POOLED_SIZE]
) {
#pragma HLS INLINE off

    POOL: for (int i = 0; i < POOLED_SIZE; i++) {
#pragma HLS PIPELINE II=1
#pragma HLS UNROLL factor=2
        int idx = i << 1;  // i * 2
        pooled[i] = (conv_out[idx] > conv_out[idx + 1]) ?
                    (fixed_t)conv_out[idx] :
                    (fixed_t)conv_out[idx + 1];
    }
}

static void compute_fc(
    fixed_t pooled[POOLED_SIZE],
    acc_t fc_out[FC_OUT]
) {
#pragma HLS INLINE off

    // Precompute scaled biases
    const acc_t bias_0 = (acc_t)FC_BIAS_QUANT[0] * FC_BIAS_SCALE;
    const acc_t bias_1 = (acc_t)FC_BIAS_QUANT[1] * FC_BIAS_SCALE;

    // Fully unrolled FC computation (only 2 outputs, 4 inputs)
    acc_t acc_0 = bias_0;
    acc_t acc_1 = bias_1;

    FC_COMPUTE: for (int i = 0; i < FC_IN; i++) {
#pragma HLS UNROLL
        acc_t weight_0 = (acc_t)FC_WEIGHT_QUANT[0][i] * FC_WEIGHT_SCALE;
        acc_t weight_1 = (acc_t)FC_WEIGHT_QUANT[1][i] * FC_WEIGHT_SCALE;

        acc_0 += (acc_t)pooled[i] * weight_0;
        acc_1 += (acc_t)pooled[i] * weight_1;
    }

    fc_out[0] = acc_0;
    fc_out[1] = acc_1;
}

static void output_decision(
    acc_t fc_out[FC_OUT],
    hls::stream<axis_output> &predicted_bit
) {
#pragma HLS INLINE off
```

```
    axis_output out;
    out.data = (ap_uint<1>)((fc_out[1] > fc_out[0]) ? 1 : 0);
    out.last = 1;
    predicted_bit.write(out);
}

void fsk_cnn_demod(
    hls::stream<axis_data> &input_I,
    hls::stream<axis_data> &input_Q,
    hls::stream<axis_output> &predicted_bit
) {
#pragma HLS INTERFACE axis port=input_I
#pragma HLS INTERFACE axis port=input_Q
#pragma HLS INTERFACE axis port=predicted_bit
#pragma HLS INTERFACE ap_ctrl_none port=return

// Enable task-level pipelining - allows new input while processing previous
#pragma HLS DATAFLOW

    // Intermediate buffers
    fixed_t I_buffer[SAMPLES_PER_SYMBOL];
    fixed_t Q_buffer[SAMPLES_PER_SYMBOL];
    acc_t conv_out[SAMPLES_PER_SYMBOL];
    fixed_t pooled[POOLED_SIZE];
    acc_t fc_out[FC_OUT];

#pragma HLS ARRAY_PARTITION variable=I_buffer complete
#pragma HLS ARRAY_PARTITION variable=Q_buffer complete
#pragma HLS ARRAY_PARTITION variable=conv_out complete
#pragma HLS ARRAY_PARTITION variable=pooled complete
#pragma HLS ARRAY_PARTITION variable=fc_out complete

    // Dataflow pipeline stages
    read_iq_parallel(input_I, input_Q, I_buffer, Q_buffer);
    compute_convolution(I_buffer, Q_buffer, conv_out);
    compute_pooling(conv_out, pooled);
    compute_fc(pooled, fc_out);
    output_decision(fc_out, predicted_bit);
}
```

This module implements a fully pipelined CNN-based FSK demodulator in Vitis HLS using AXI-Stream interfaces. I and Q samples are read in parallel and processed through a single-tap convolution layer with fixed-point quantized weights and bias, followed by max-pooling to reduce the temporal dimension. The pooled features are then passed to a fully connected layer with two outputs representing the symbol classes. A final decision stage compares the two logits and outputs the predicted bit on an AXI-Stream channel. The design uses HLS DATAFLOW, loop pipelining, and array partitioning to maximize throughput and allow continuous symbol processing with an initiation interval of one sample.

This, has been validate through the following testbench:

Synthesis Summary(solution1) | fsk_cnn_demod.cpp | Dataflow(solution1) | Function Call Graph | fsk_cnn_demod_tb.cpp

```cpp
#ifndef __SYNTHESIS__
#include <iostream>
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <iomanip>

#include "fsk_cnn_demod.hpp"

void generate_fsk_symbol(int bit, float input_I[8], float input_Q[8]) {
    // Match Python FSK generation exactly:
    // phase_shift = 2 * np.pi * (1 if bit == 1 else -1) / SAMPLES_PER_SYMBOL
    // I = np.cos(phase_shift * np.arange(SAMPLES_PER_SYMBOL))
    // Q = np.sin(phase_shift * np.arange(SAMPLES_PER_SYMBOL))

    float phase_shift;
    if (bit == 1) {
        phase_shift = 2.0 * M_PI * 1.0 / 8.0;  // 2*pi/8 for bit 1
    } else {
        phase_shift = 2.0 * M_PI * (-1.0) / 8.0;  // -2*pi/8 for bit 0
    }

    for (int i = 0; i < 8; i++) {
        input_I[i] = cos(phase_shift * i);
        input_Q[i] = sin(phase_shift * i);
    }
}

int main() {
    // Use fixed seed for reproducibility during debug
    srand(42);

    std::cout << "================================================\n";
    std::cout << "FSK CNN DEMODULATOR - DEBUG TESTBENCH\n";
    std::cout << "Testing with exact Python FSK generation\n";
    std::cout << "================================================\n\n";

    int correct_count = 0;
    int total_bits = 10;

    // Test with 10 random bits
    for (int test_num = 0; test_num < total_bits; test_num++) {
        // Generate random bit (0 or 1)
        int test_bit = rand() % 2;

        std::cout << "Test #" << (test_num + 1) << " - Bit = " << test_bit << "\n";
        std::cout << "------------------------------------------------\n";
```

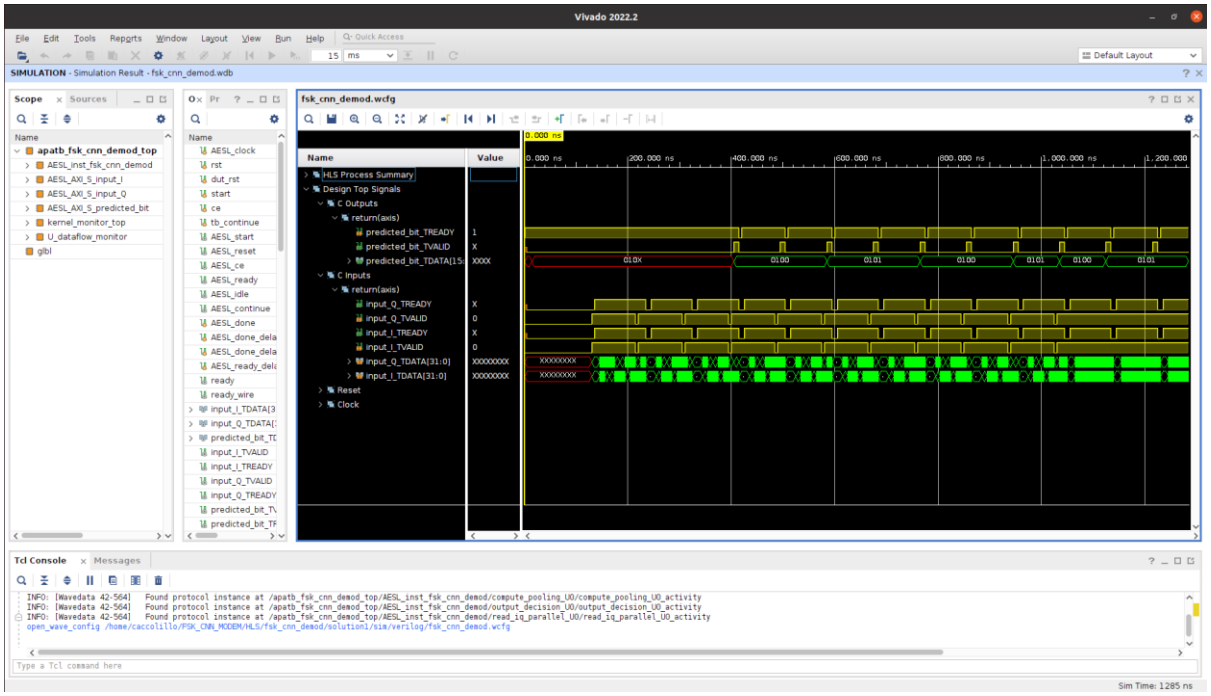
Synthesis Summary(solution1) | fsk_cnn_demod.cpp | Dataflow(solution1) | Function Call Graph | fsk_cnn_demod_tb.cpp

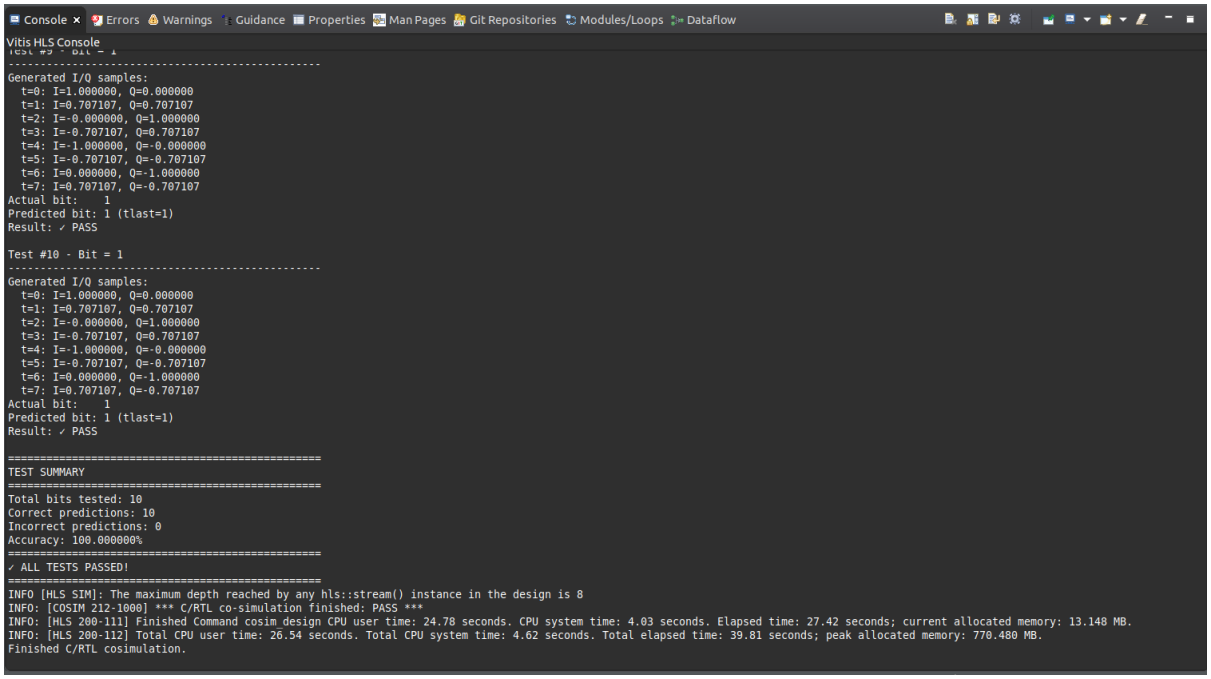
```cpp
        // Generate FSK symbol
        float I_float[8], Q_float[8];
        generate_fsk_symbol(test_bit, I_float, Q_float);

        // Print generated symbols for debugging
        std::cout << "Generated I/Q samples:\n";
        std::cout << std::fixed << std::setprecision(6);
        for (int i = 0; i < 8; i++) {
            std::cout << "  t=" << i << ": I=" << I_float[i]
                      << ", Q=" << Q_float[i] << "\n";
        }

        // Create AXI streams
        hls::stream<axis_data> input_I_stream;
        hls::stream<axis_data> input_Q_stream;
        hls::stream<axis_output> predicted_bit_stream;

        // Convert to fixed-point and write to streams
        for (int i = 0; i < 8; i++) {
            axis_data I_val, Q_val;
            I_val.data = fixed_t(I_float[i]);
            I_val.last = (i == 7) ? 1 : 0;
            Q_val.data = fixed_t(Q_float[i]);
            Q_val.last = (i == 7) ? 1 : 0;
            input_I_stream.write(I_val);
            input_Q_stream.write(Q_val);
        }

        // Run inference
        fsk_cnn_demod(input_I_stream, input_Q_stream, predicted_bit_stream);

        // Read predicted bit from output stream
        axis_output predicted_output = predicted_bit_stream.read();
        int predicted_bit = predicted_output.data;

        // Check if prediction is correct
        bool is_correct = (predicted_bit == test_bit);
        if (is_correct) {
            correct_count++;
        }

        // Print results
        std::cout << "Actual bit:    " << test_bit << "\n";
        std::cout << "Predicted bit: " << predicted_bit
```

```
                  << " (tlast=" << predicted_output.last << ")\n";
        std::cout << "Result: " << (is_correct ? "✓ PASS" : "✗ FAIL") << "\n\n";
    }

    // Print summary statistics
    std::cout << "================================================\n";
    std::cout << "TEST SUMMARY\n";
    std::cout << "================================================\n";
    std::cout << "Total bits tested: " << total_bits << "\n";
    std::cout << "Correct predictions: " << correct_count << "\n";
    std::cout << "Incorrect predictions: " << (total_bits - correct_count) << "\n";
    std::cout << "Accuracy: " << (100.0 * correct_count / total_bits) << "%\n";
    std::cout << "================================================\n";

    if (correct_count == total_bits) {
        std::cout << "✓ ALL TESTS PASSED!\n";
    } else {
        std::cout << "✗ SOME TESTS FAILED\n";
    }
    std::cout << "================================================\n";

    return (correct_count == total_bits) ? 0 : 1;
}
#endif
```

This file implements a C++ debug testbench for the FSK CNN demodulator. It generates ideal FSK symbols that exactly match the Python training data, converts them to fixed-point, and feeds them into the demodulator via AXI-Stream interfaces. The testbench runs inference on a set of random bits, compares predicted outputs against the ground truth, and reports per-test results and overall accuracy. This environment is intended for functional validation and bit-accurate debugging prior to HLS synthesis and hardware deployment.

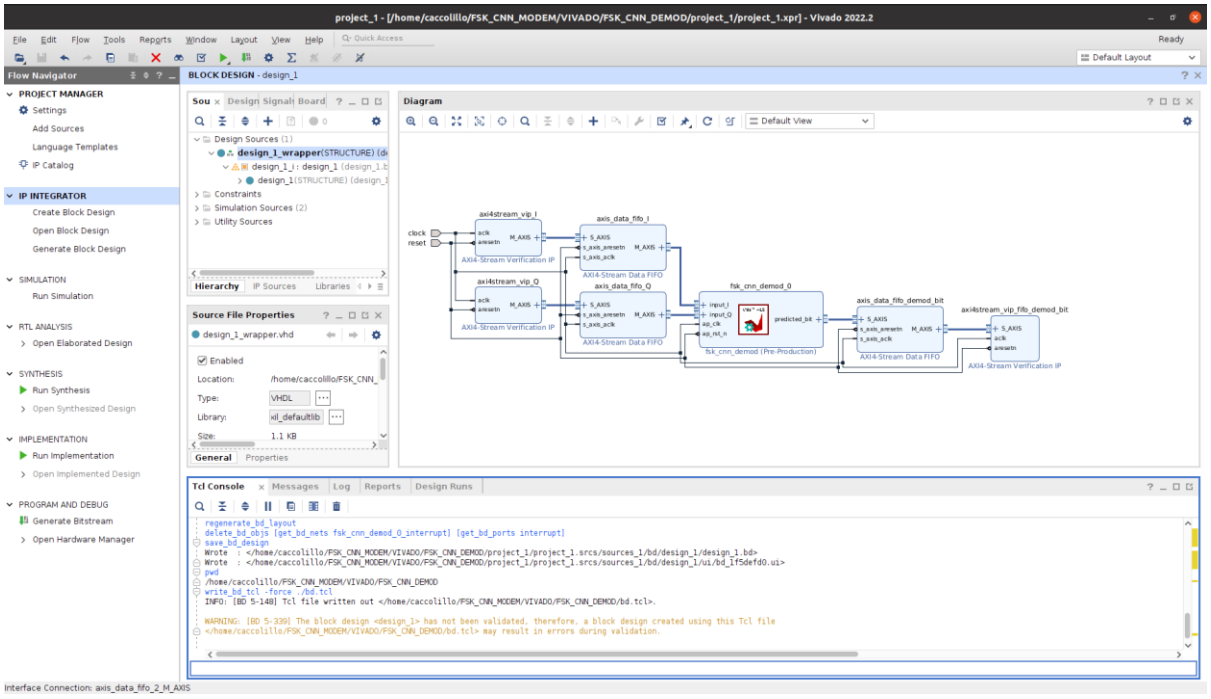
The test passes both in C simulation and in C/RTL cosimulation:



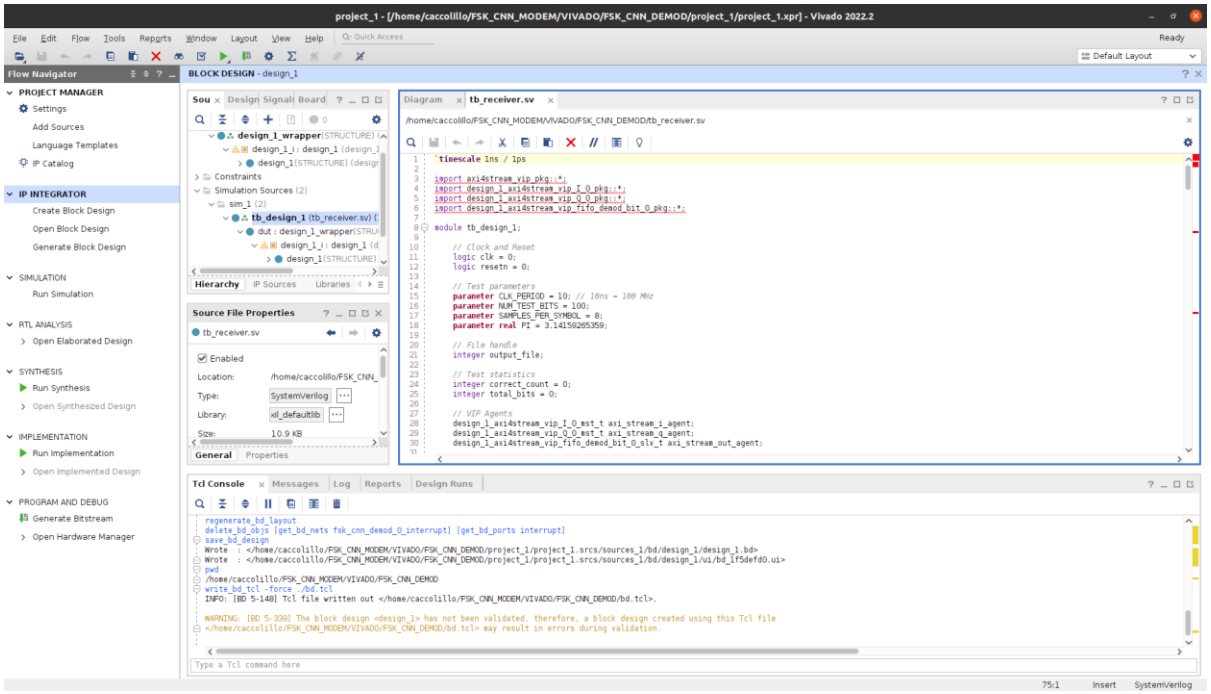


HLS CNN VIVADO TESTBENCH

A Vivado design leveraging upon AXI stream VIPs has been created:

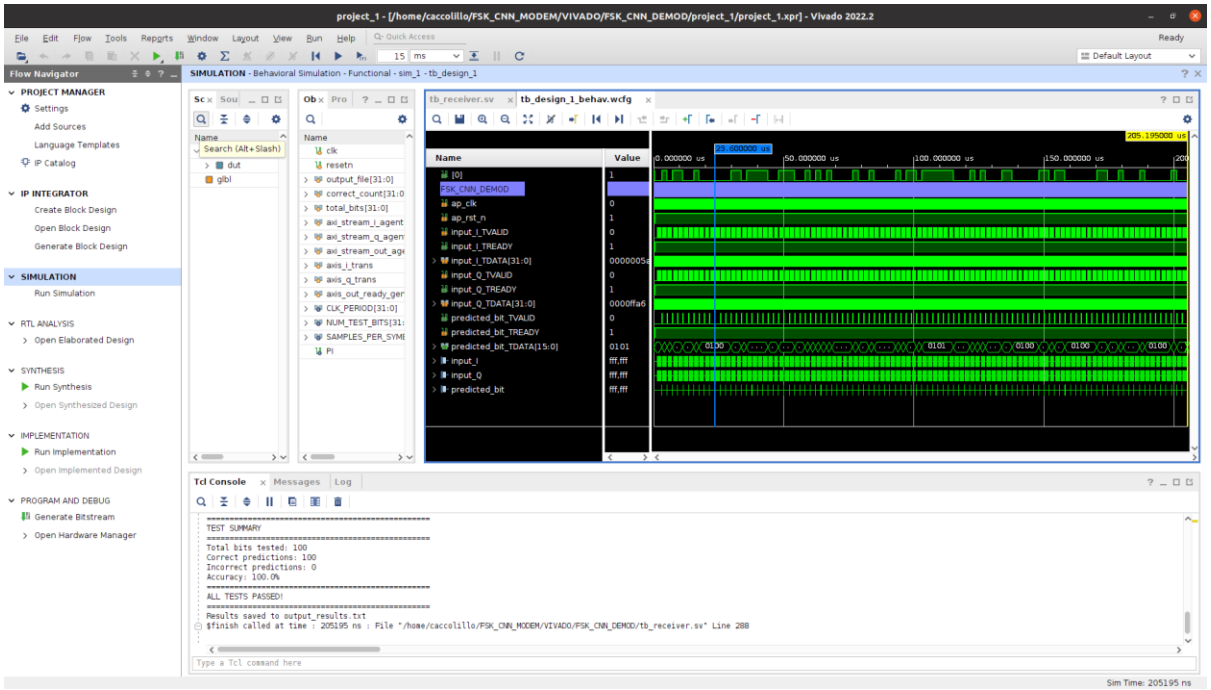


A system verilog testbench mimics the Vitis HLS one:



This SystemVerilog testbench provides a comprehensive functional verification environment for the CNN-based FSK demodulator integrated in the Zynq/Vivado block design. AXI4-Stream VIP master agents are used to inject synchronized I and Q sample streams into the DUT, while a slave VIP agent captures the predicted demodulated bit output. For each test, the bench generates ideal FSK symbols using a phase-increment model that exactly matches the Python training data and the C++ HLS reference implementation, then converts the samples to fixed-point format consistent with the hardware interface.

The test sequence applies multiple randomized symbols, enforces proper AXI handshaking and TLAST signaling, and validates the demodulator's output against the known transmitted bit. Detailed console output and file-based logging are used to record per-symbol results, summary statistics, and overall accuracy. Reset sequencing, ready/valid behavior, timeout protection, and controlled stimulus timing are included to ensure robust verification prior to hardware deployment.



The ip core as is, is conceived to be used in a CPU-less environment, so the usage of a DMA hasn't been evaluated.

Anyway, as when using a Zynq 7000 or Zynq US+ it has to be used, the IP core can be reworked with an AXI stream subset converter:

https://community.element14.com/technologies/fpga-group/b/blog/posts/use-the-zynq-xadc-with-dma-part-1-bare-metal

https://discuss.pynq.io/t/transferring-xadc-packets-of-samples-through-dma/3248

PORTING THE CNN TO VITIS AI

As the tool version in use is 2022.2, the compatible Vitis AI version to use is the 3.0:

https://xilinx.github.io/Vitis-AI/3.0/html/docs/workflow-system-integration.html#workflow-dpu

Now to aim is to deploy the trained CNN on a DPU hosted on an Ultra96 v2 board, and we're going it through the so called "Vivado flow".

The Zynq US+ supports the folowing DPU:

https://docs.amd.com/r/en-US/pg338-dpu?tocId=3xsG16y_QFTWvAJKHbisEw

Which is available as an IP-core at:

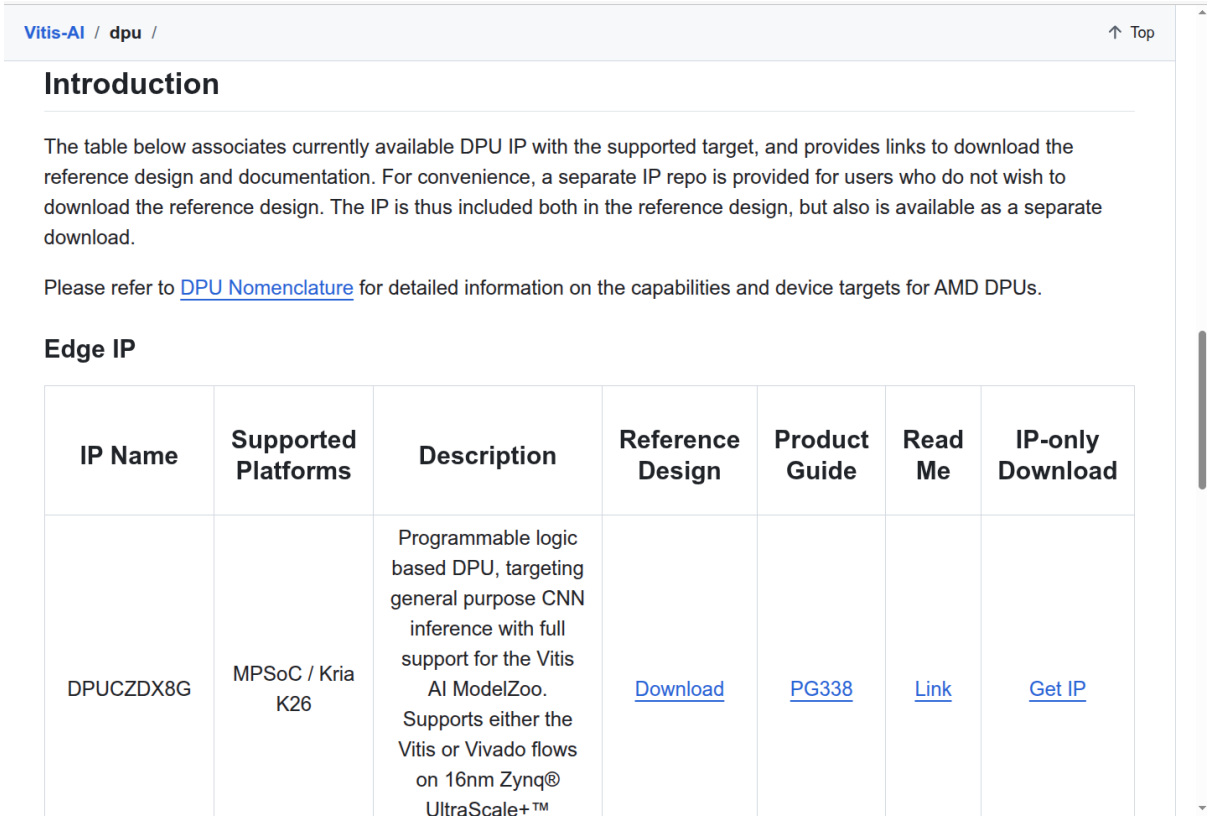
https://github.com/Xilinx/Vitis-AI/tree/3.0/dpu

For the 3.0 version of the Vitis AI tool we're going to use.

According to the documentation:

https://github.com/Xilinx/Vitis-AI/tree/3.0/dpu

A reference Vivado design is availabe for download:



Vitis-AI / dpu / ↑ Top

Introduction

The table below associates currently available DPU IP with the supported target, and provides links to download the reference design and documentation. For convenience, a separate IP repo is provided for users who do not wish to download the reference design. The IP is thus included both in the reference design, but also is available as a separate download.

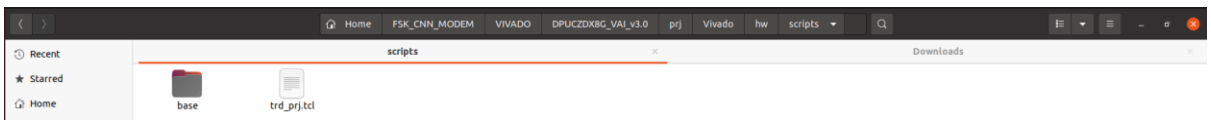
Please refer to DPU Nomenclature for detailed information on the capabilities and device targets for AMD DPUs.

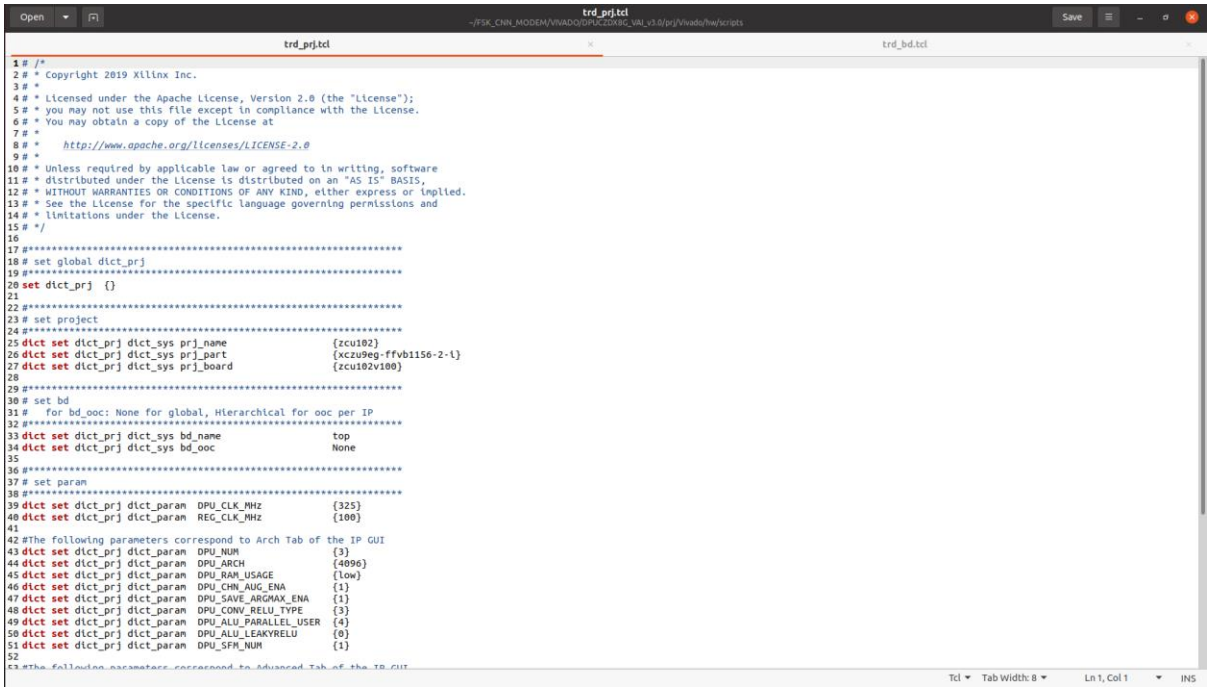
Edge IP

IP Name	Supported Platforms	Description	Reference Design	Product Guide	Read Me	IP-only Download
DPUCZDX8G	MPSoC / Kria K26	Programmable logic based DPU, targeting general purpose CNN inference with full support for the Vitis AI ModelZoo. Supports either the Vitis or Vivado flows on 16nm Zynq® UltraScale+™	Download	PG338	Link	Get IP

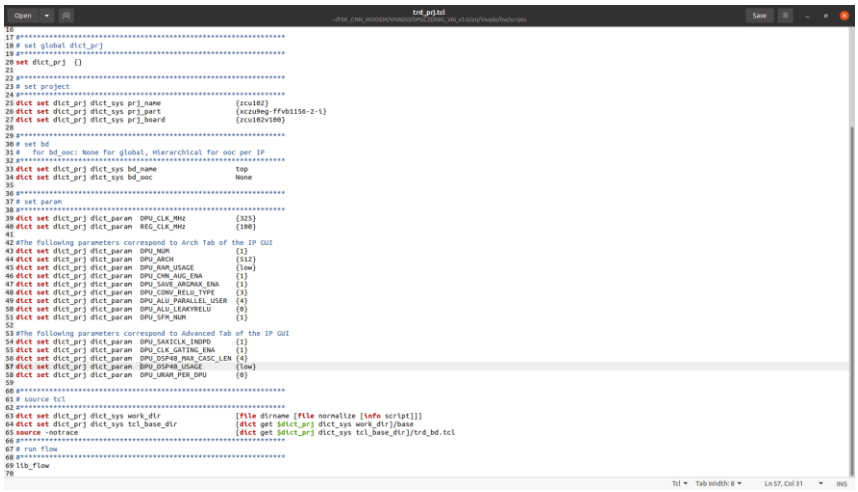
https://www.xilinx.com/bin/public/openDownload?filename=DPUCZDX8G_VAI_v3.0.tar.gz

Once downloaded, we can open the script "trd_prj.tcl" in the prj/Vivado/hs/scripts subfolder:

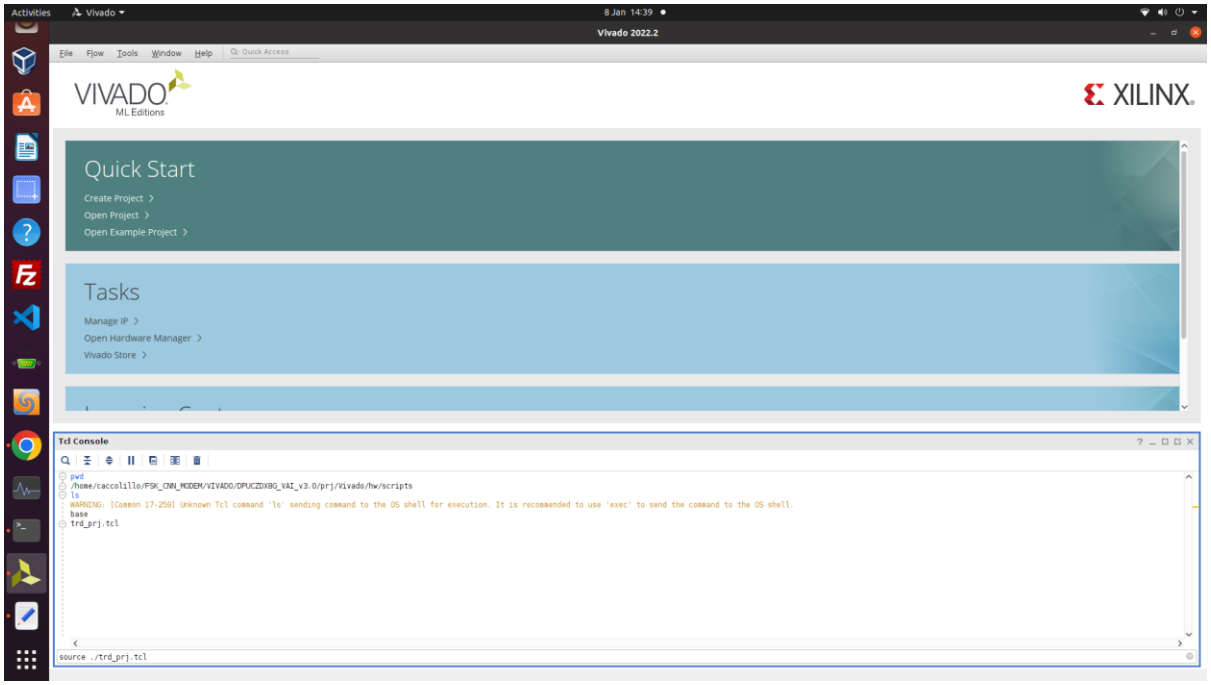




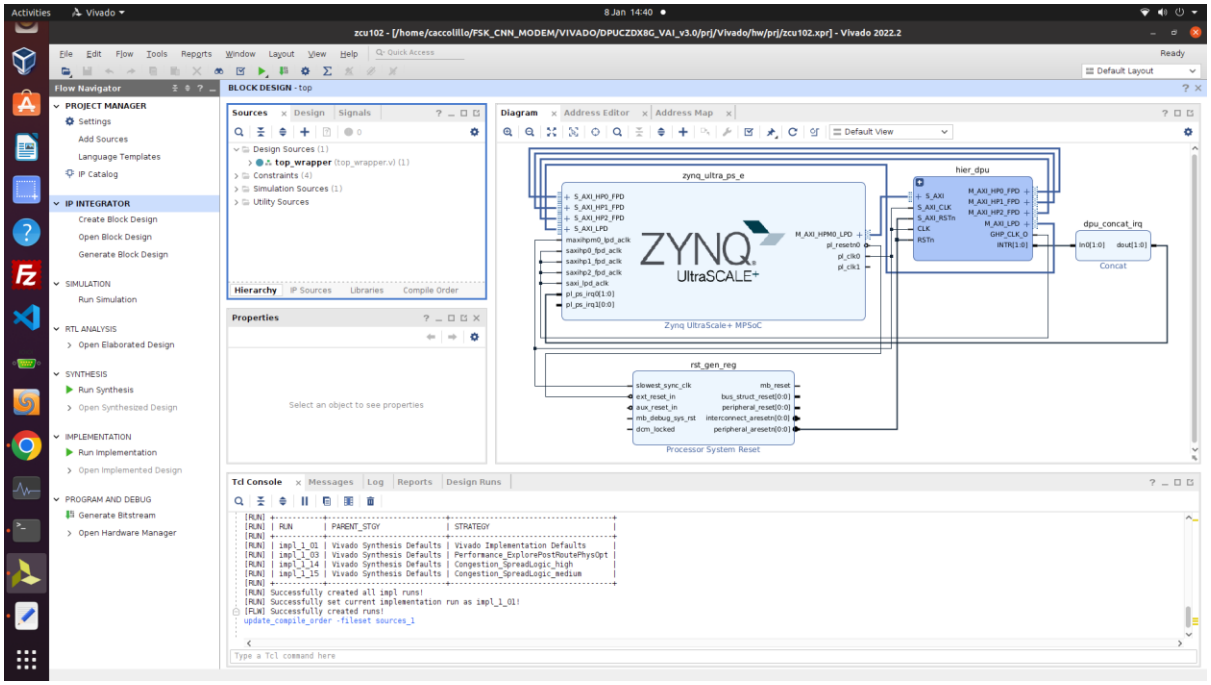
And change it so to have DPU_NUM = 1, DPU_ARCH = 2304, DPU_DSP48_USAGE = low:



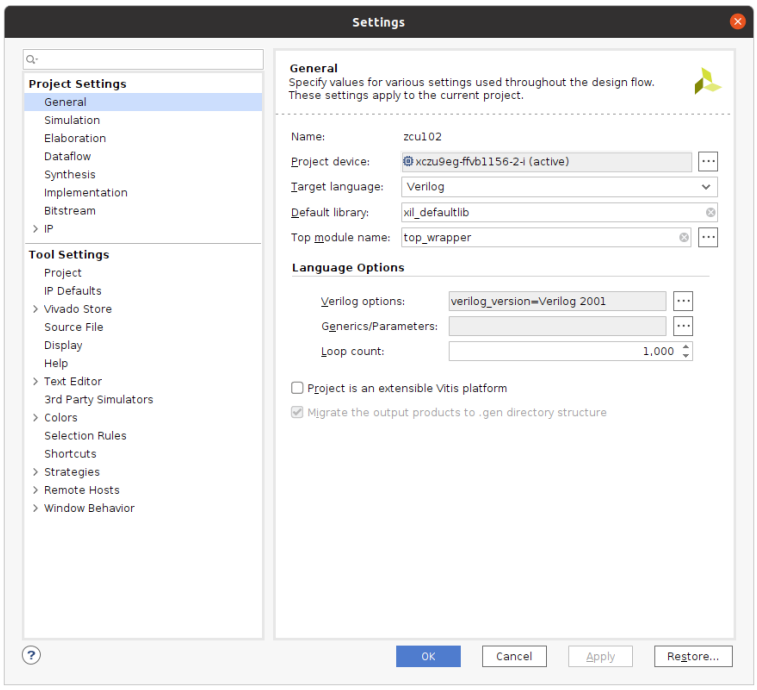
Then we run the script in Vivado getting the design, but for the zcu102 board.



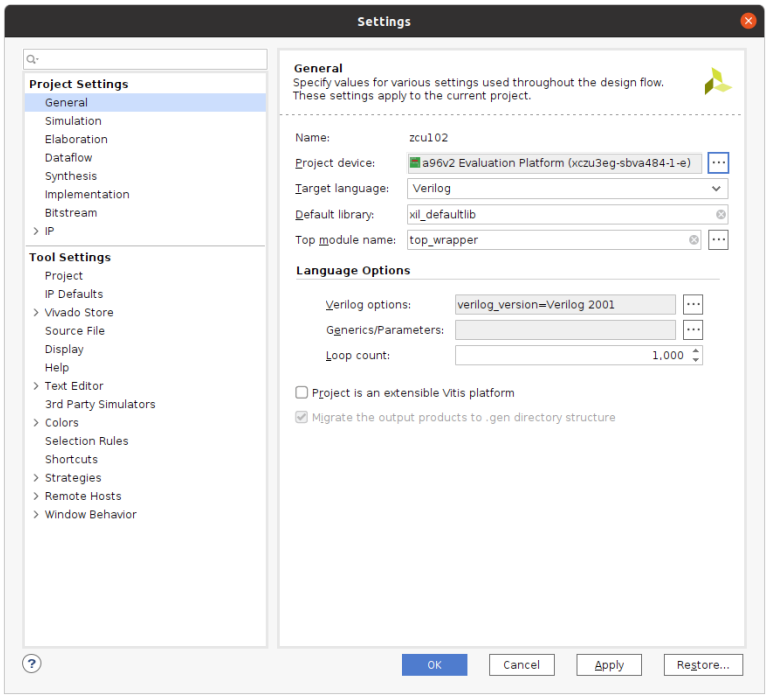
Once executed it:



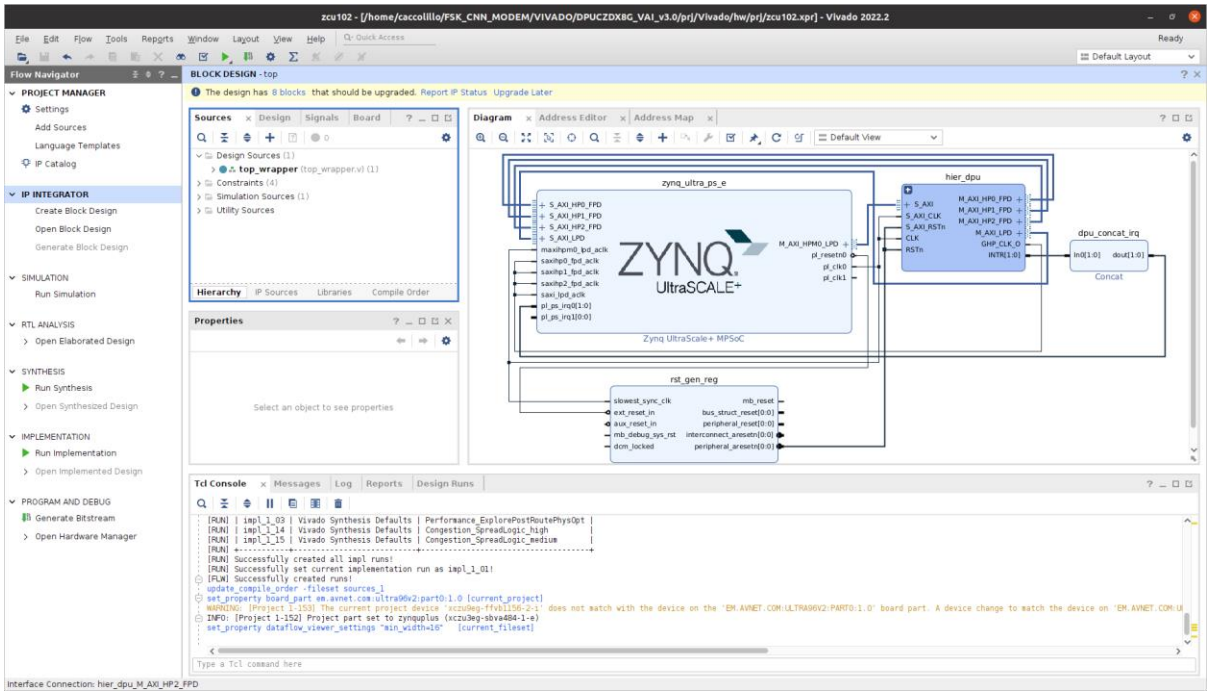
we can change the board in use:

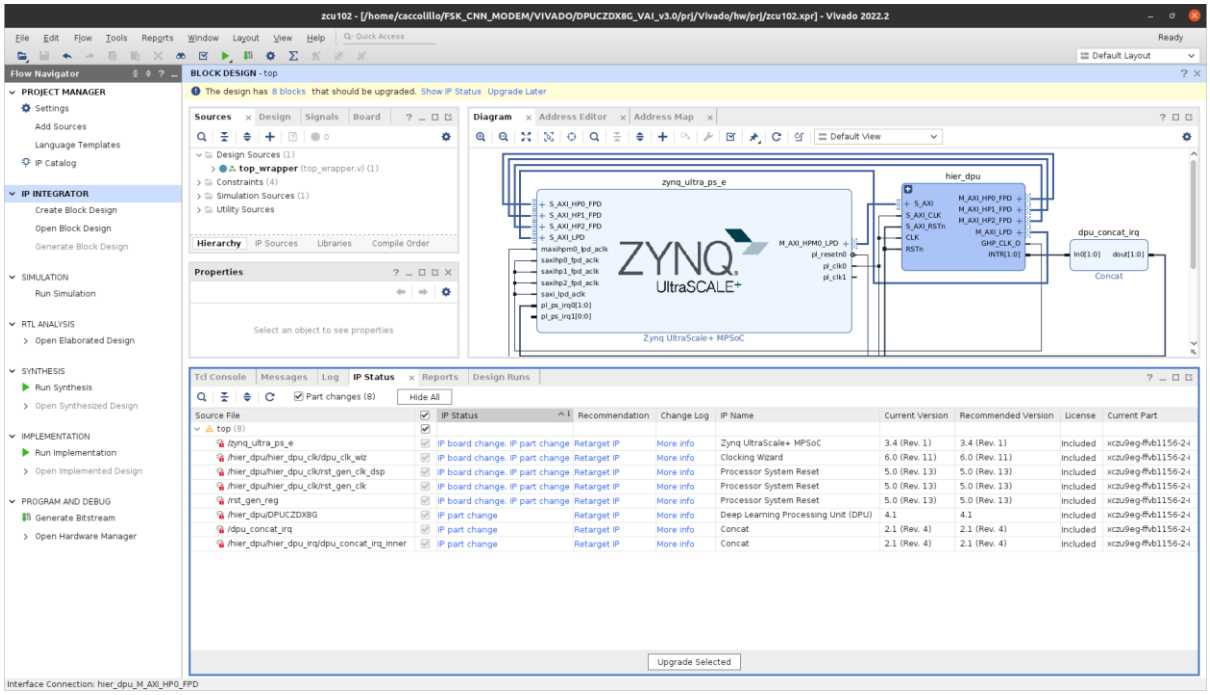


to the Ultra96 v2:

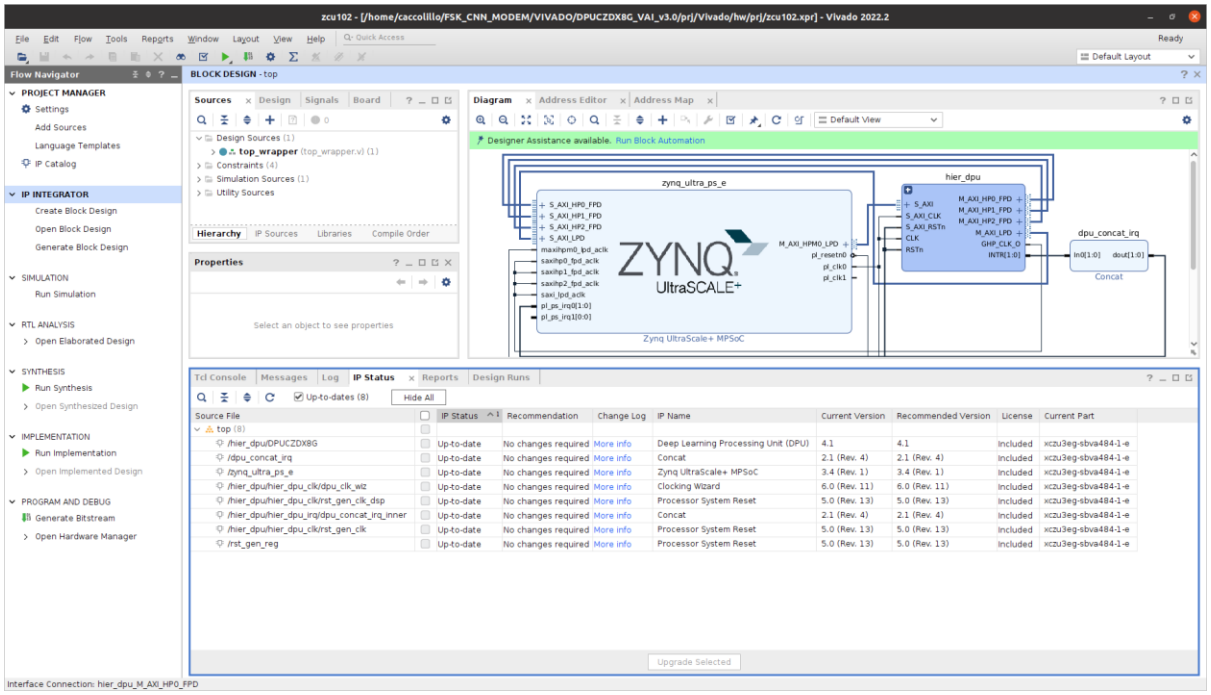


run the report IP summary:

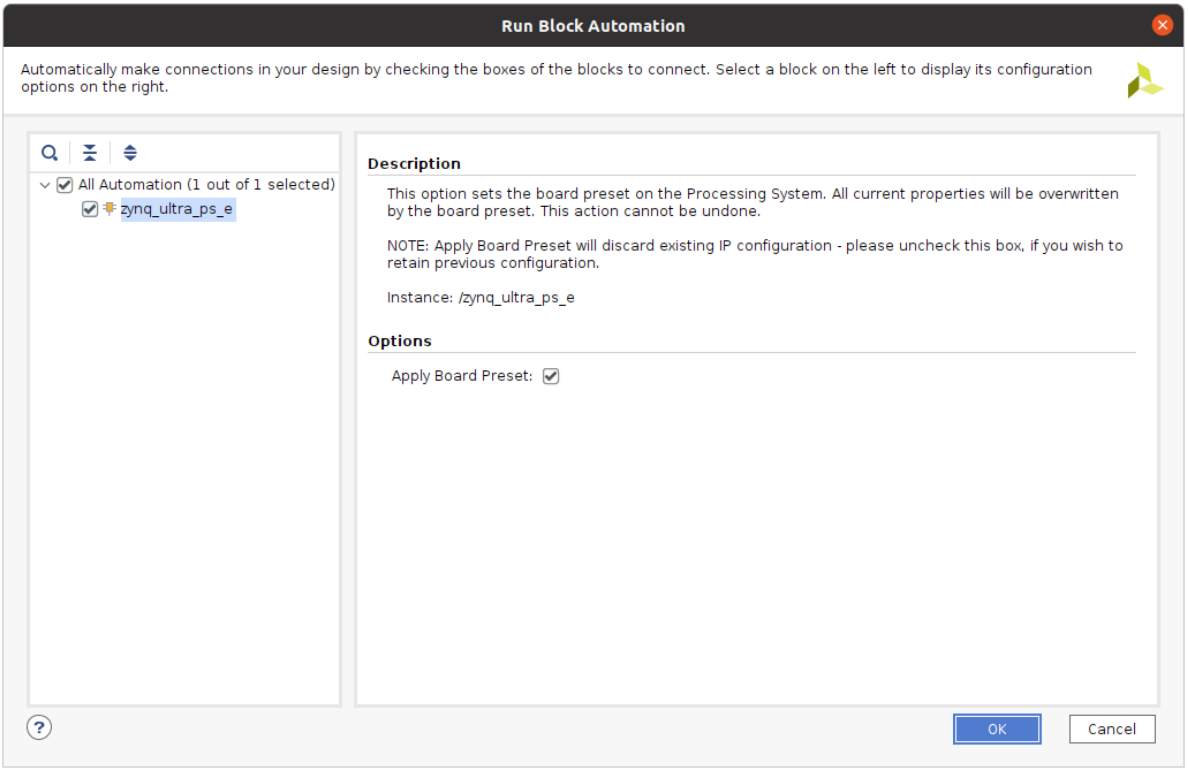




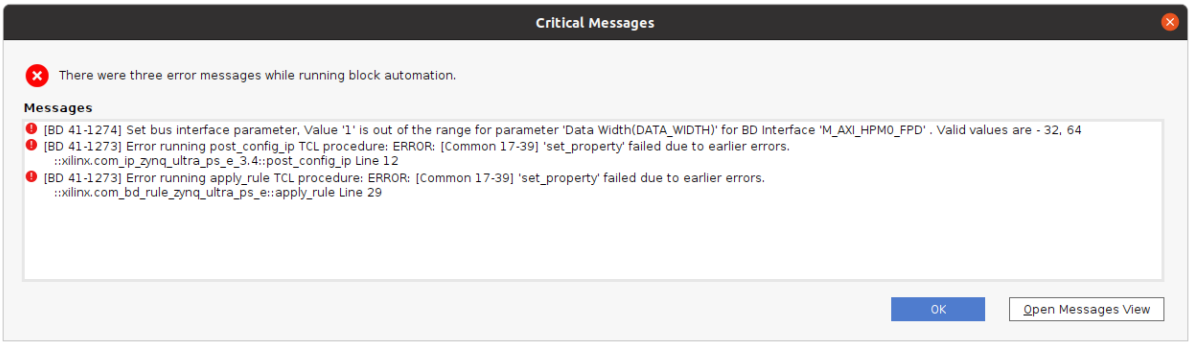
upgrade the IPs:

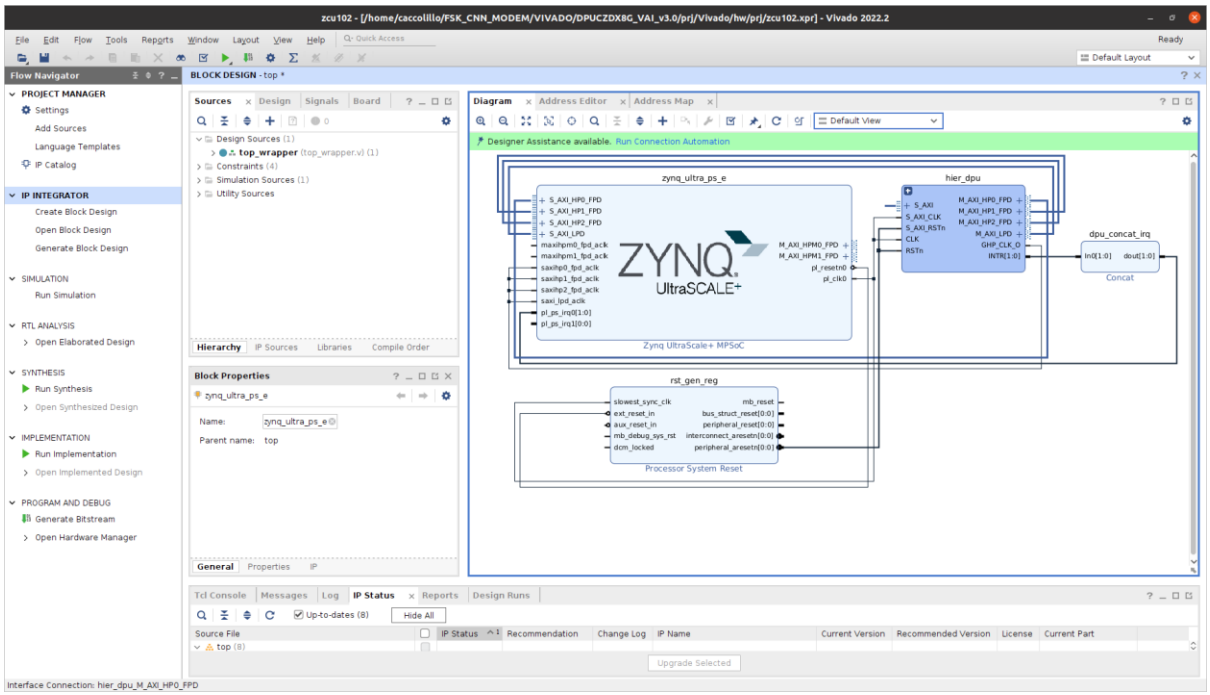


Then run block automation so to apply Ultra96 board presets:

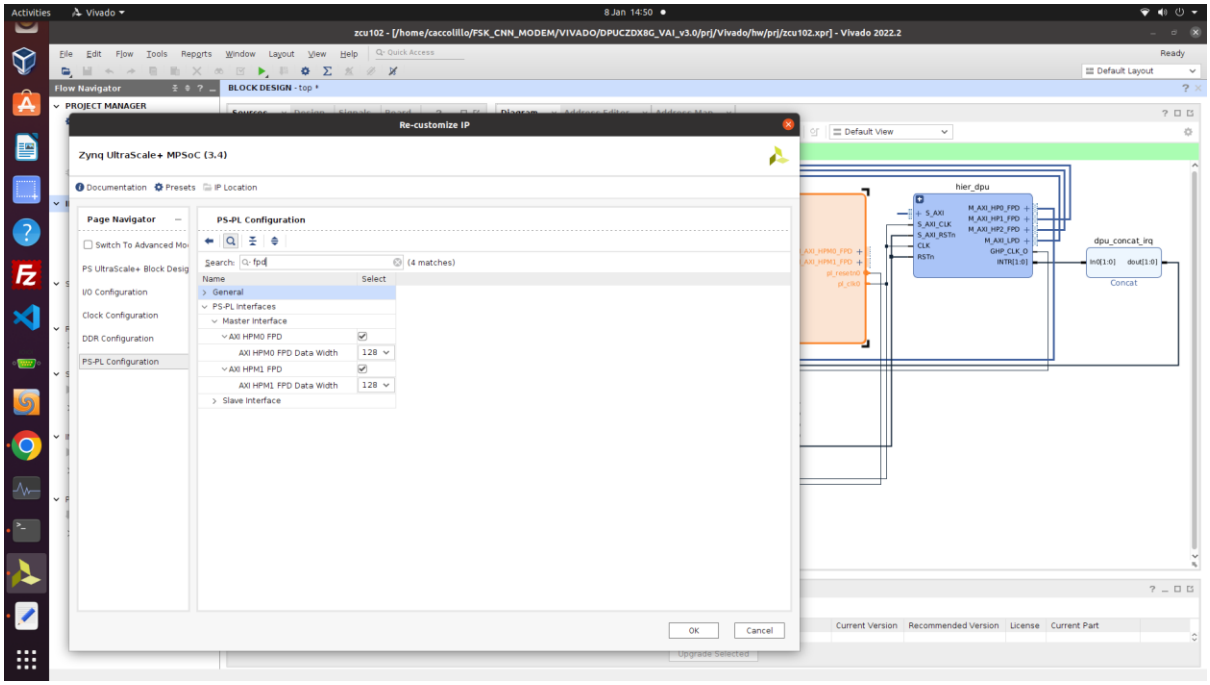


It will error, modify the block design and disconnect the DPU:

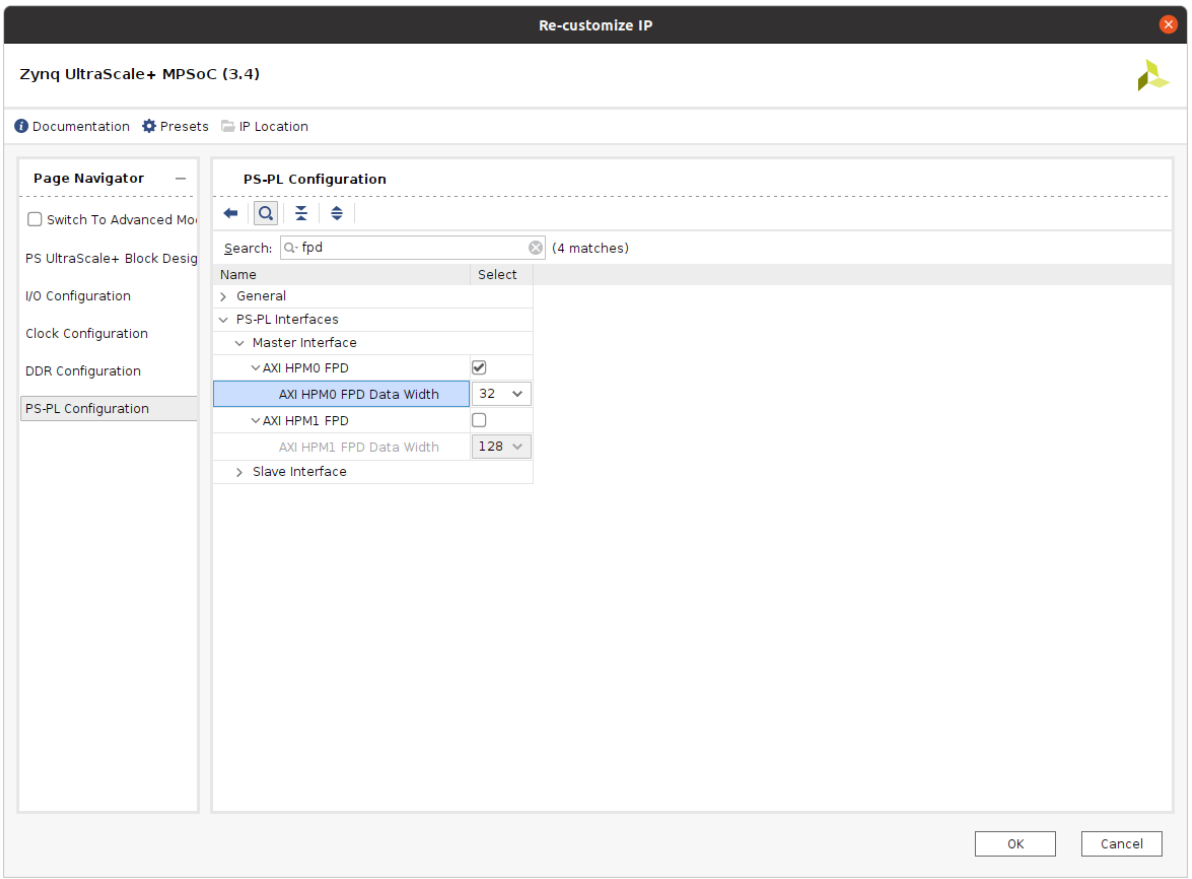




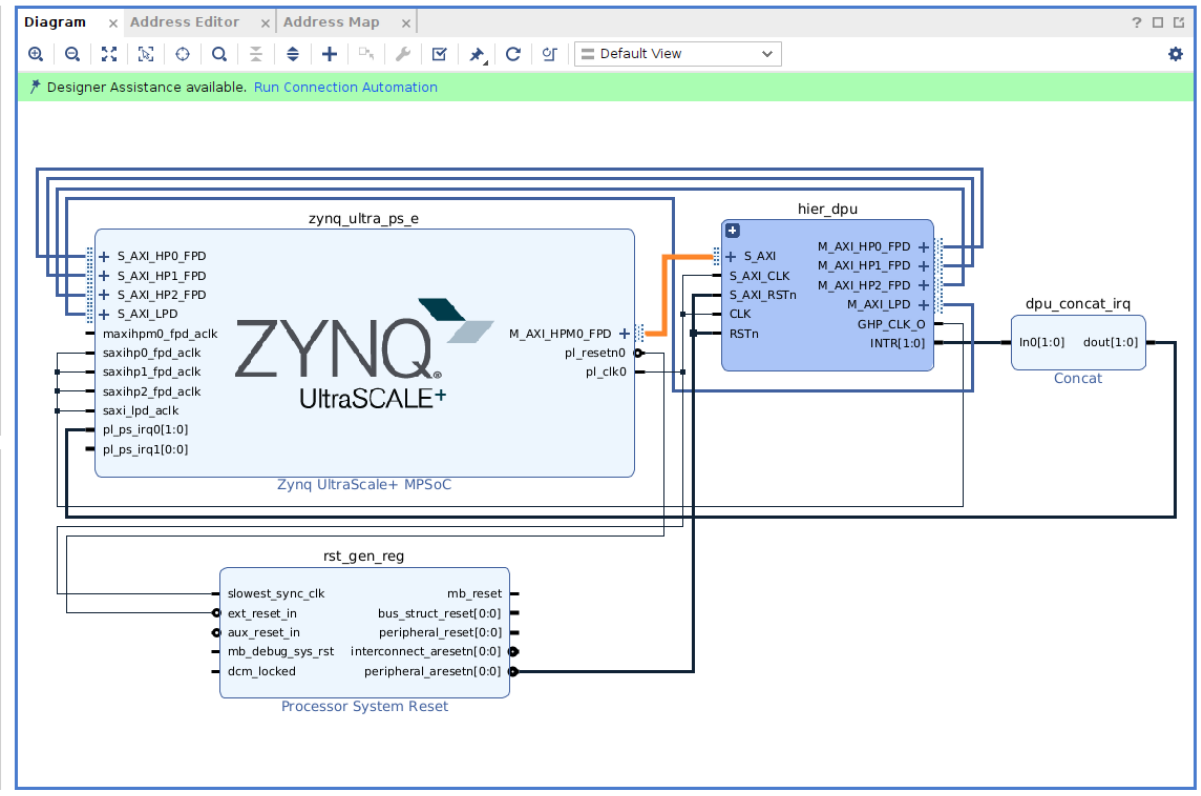
then we have to open the zynq_ultra_ps_e block and change configuration parameters for the master AXI ports:



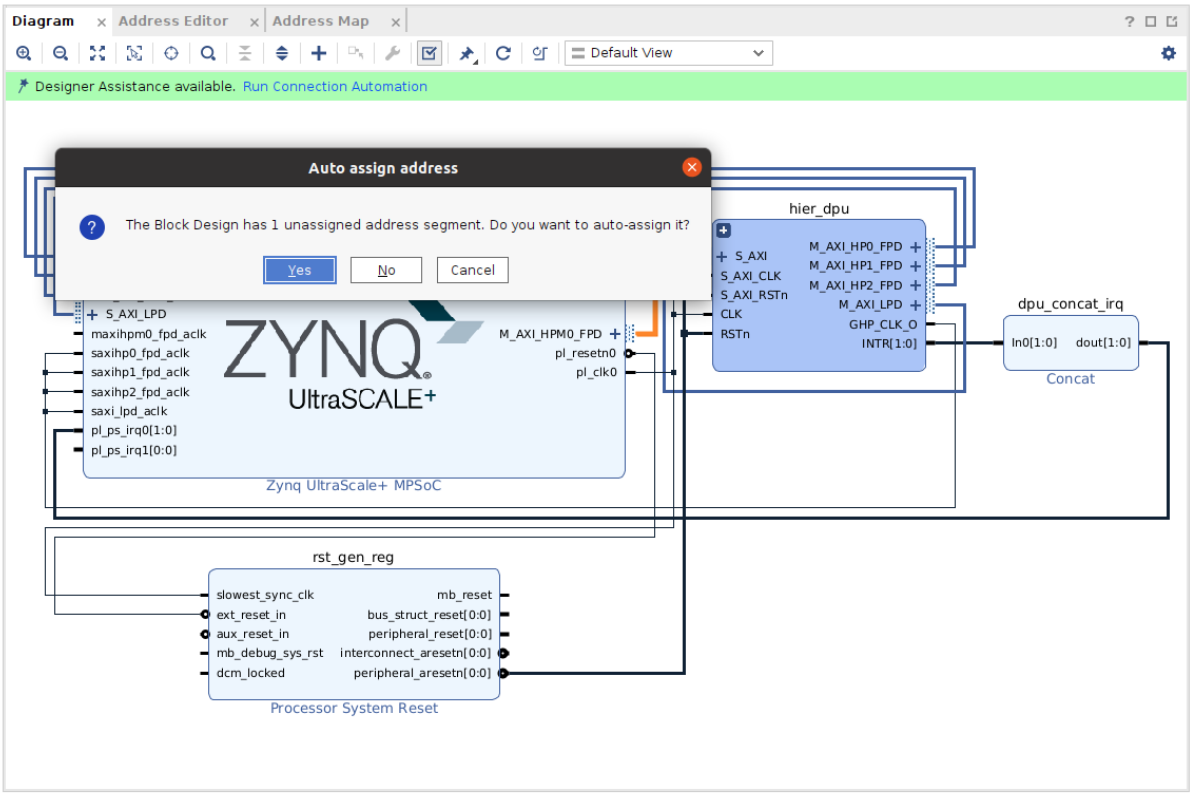
Disconnect the HPM1 FPD port, reconfigure the other one to use 32 bits:



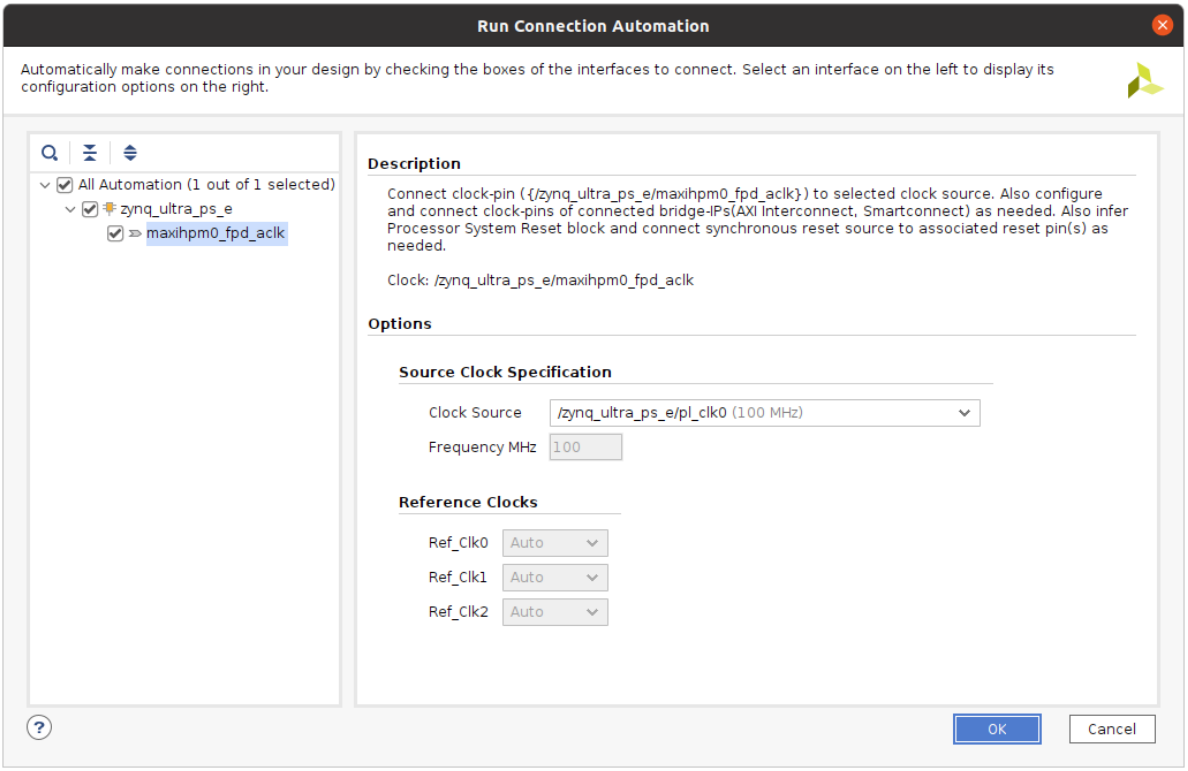
Restore the connection to the DPU:



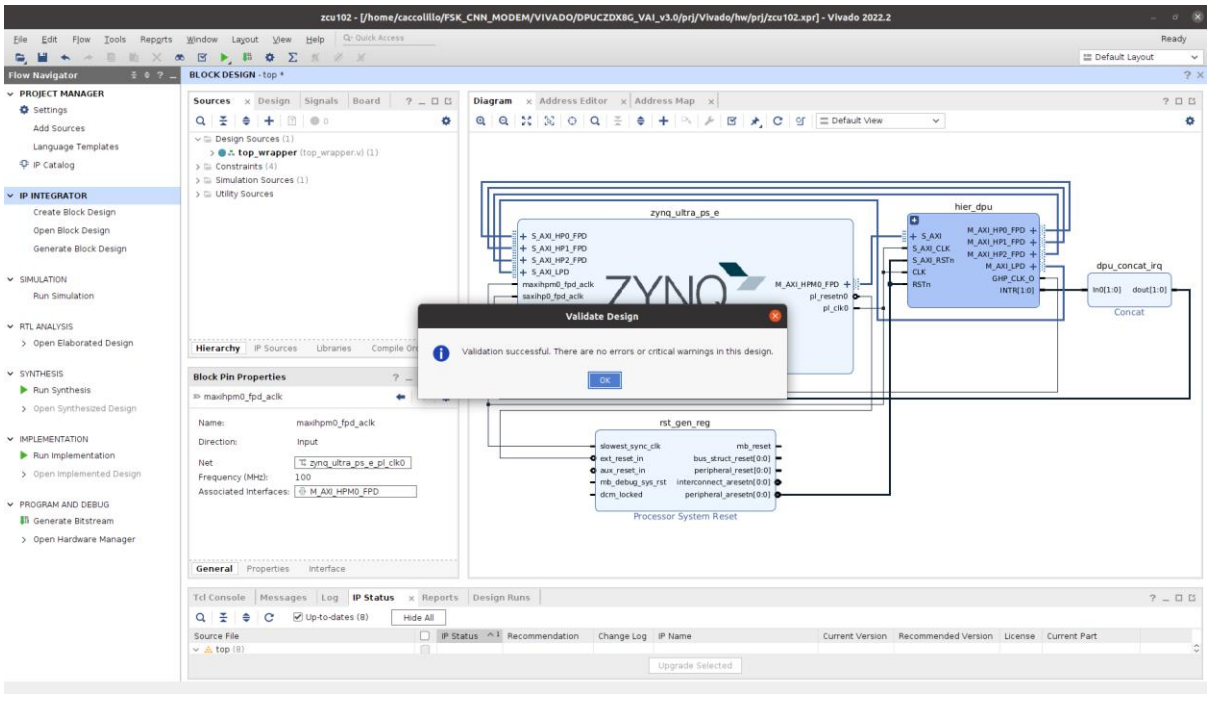
validate the block design:



Press yes, run connection automation:



Validate the design:



and we're sorted.

Now the project is retargeted to the Ultra96 v2.

SD PRE-BUILT IMAGE

As per the following hackster page:

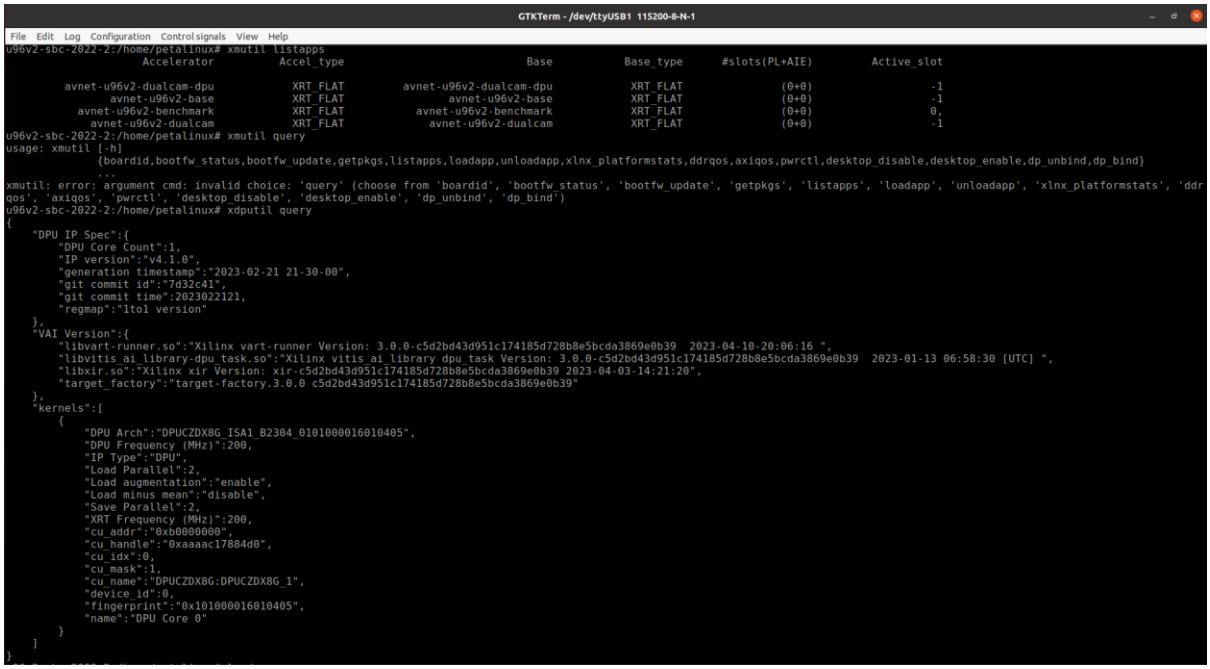
https://www.hackster.io/AlbertaBeef/ultra96-v2-adding-support-for-ros2-8ba68d?auth_token=6d7e7ae101ef33c3b16e9a5aa25a23b4&#toc-conclusion-11

A pre-built micro sd image is available, and will be used henceforth.

Once extracted the downloaded archive in a folder, it gets written on the micro sd as follows:

sudo dd if=./avnet-u96v2_sbc-v2022.2-2023-05-10.img of=/dev/sdb bs=4M status=progress conv=fsync

The board boots and the DPU is alive:



VITIS AI

Vitis AI gets downloaded from:

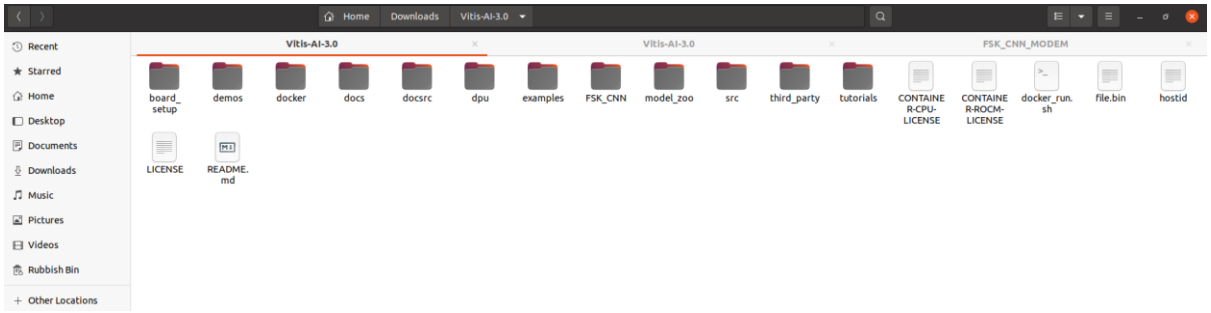
https://github.com/Xilinx/Vitis-AI/tree/3.0

And the cpu image is obtained and run as follows:

./docker_run.sh xilinx/vitis-ai-pytorch-cpu:ubuntu2004-3.0.0.106

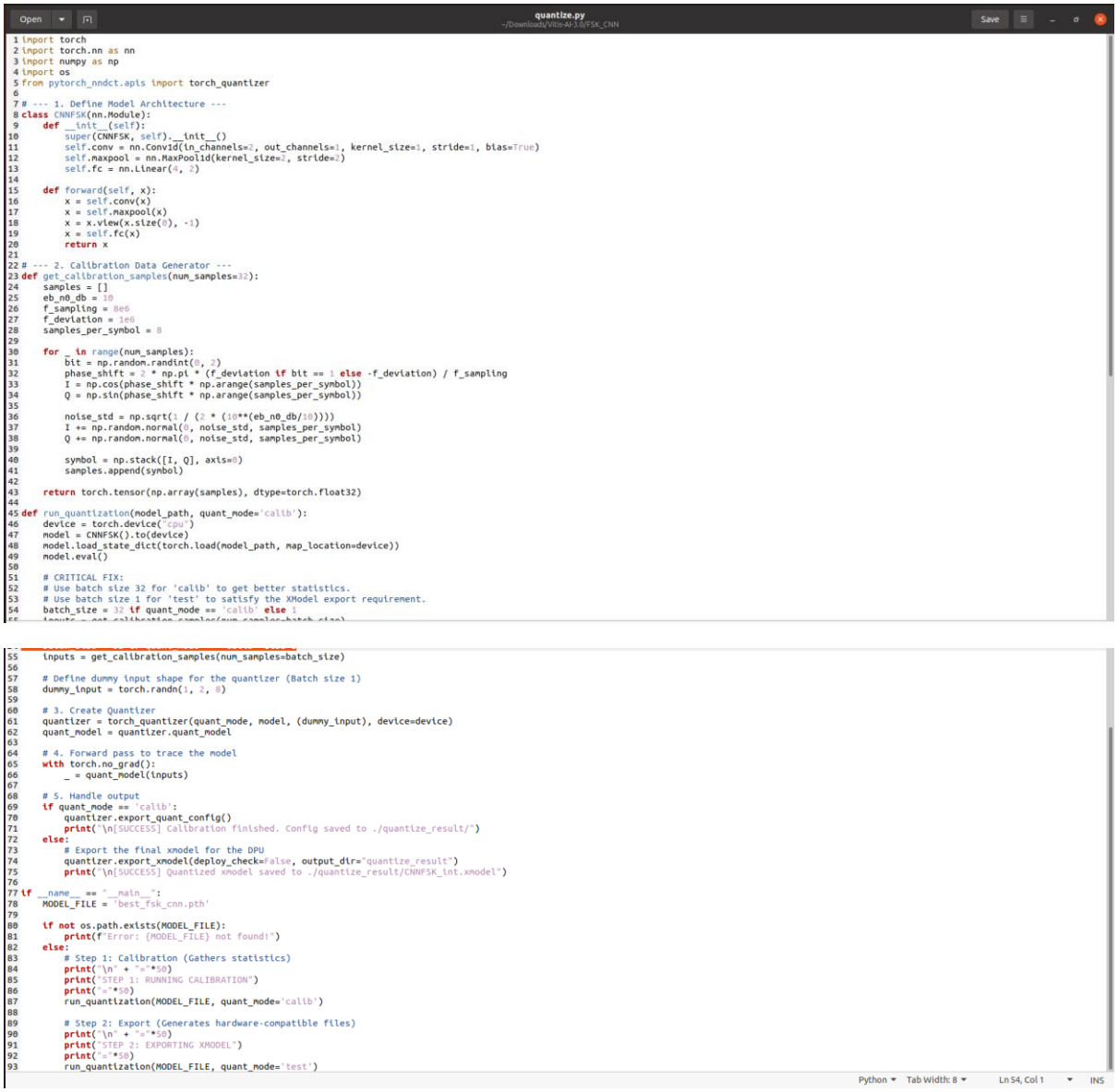
```
caccolillo@caccolillo-OMEN-25L-Desktop-GT12-1xxx:~/Downloads/Vitis-AI-3.0$ ./docker_run.sh xilinx/vitis-ai-pytorch-cpu:ubuntu2004-3.0.0.106
find: ‘/dev/vboxusb’: Permission denied
find: ‘/dev/vboxusb’: Permission denied
ubuntu2004-3.0.0.106: Pulling from xilinx/vitis-ai-pytorch-cpu
Digest: sha256:f55bd069ffd56c6358cae29df19e6085f2bcf8ea5e045744aa412fd72db521ed
Status: Image is up to date for xilinx/vitis-ai-pytorch-cpu:ubuntu2004-3.0.0.106
docker.io/xilinx/vitis-ai-pytorch-cpu:ubuntu2004-3.0.0.106
Setting up caccolillo 's environment in the Docker container...
usermod: no changes
Running as vitis-ai-user with ID 0 and group 0


==========================================


__      ___ _   _                 _____
\ \    / (_) | (_)          /\   |_   _|
 \ \  / / _| |_ _ ___ _____ /  \    | |
  \ \/ / | | __| / __|_____/ /\ \   | |
   \  /  | | |_| \__ \     / ____ \ _| |_
    \/   |_|\__|_|___/    /_/    \_\_____|

==========================================

Docker Image Version: ubuntu2004-3.0.0.106   (CPU)
Vitis AI Git Hash: d4ec26f
Build Date: 2023-01-08
WorkFlow: pytorch

vitis-ai-user@caccolillo-OMEN-25L-Desktop-GT12-1xxx:/workspace$ dir
board_setup           CONTAINER-ROCM-LICENSE  docker         docs     dpu       file.bin  hostid   model_zoo  src          tutorials
CONTAINER-CPU-LICENSE  demos                  docker_run.sh  docsrc   examples  FSK_CNN   LICENSE  README.md  third_party
vitis-ai-user@caccolillo-OMEN-25L-Desktop-GT12-1xxx:/workspace$
```

When we run the docker, the home directory is:

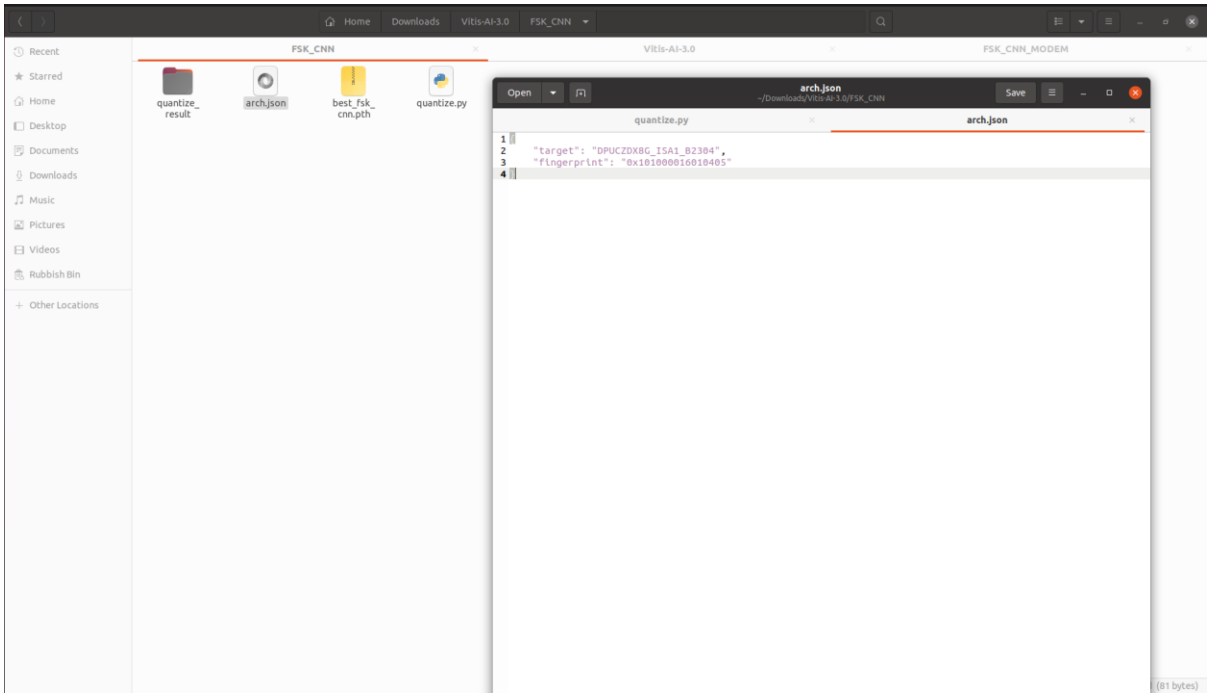


The one where the docker_run.sh script is.

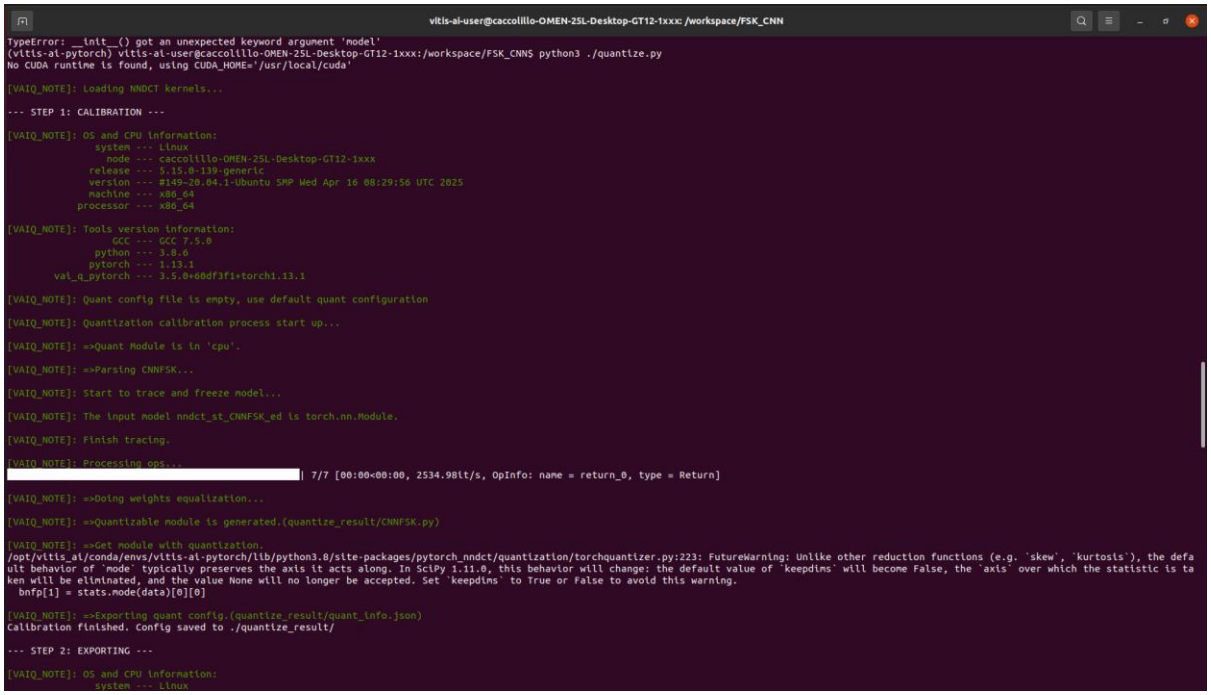
We’ve create a folder with the following quantization script:

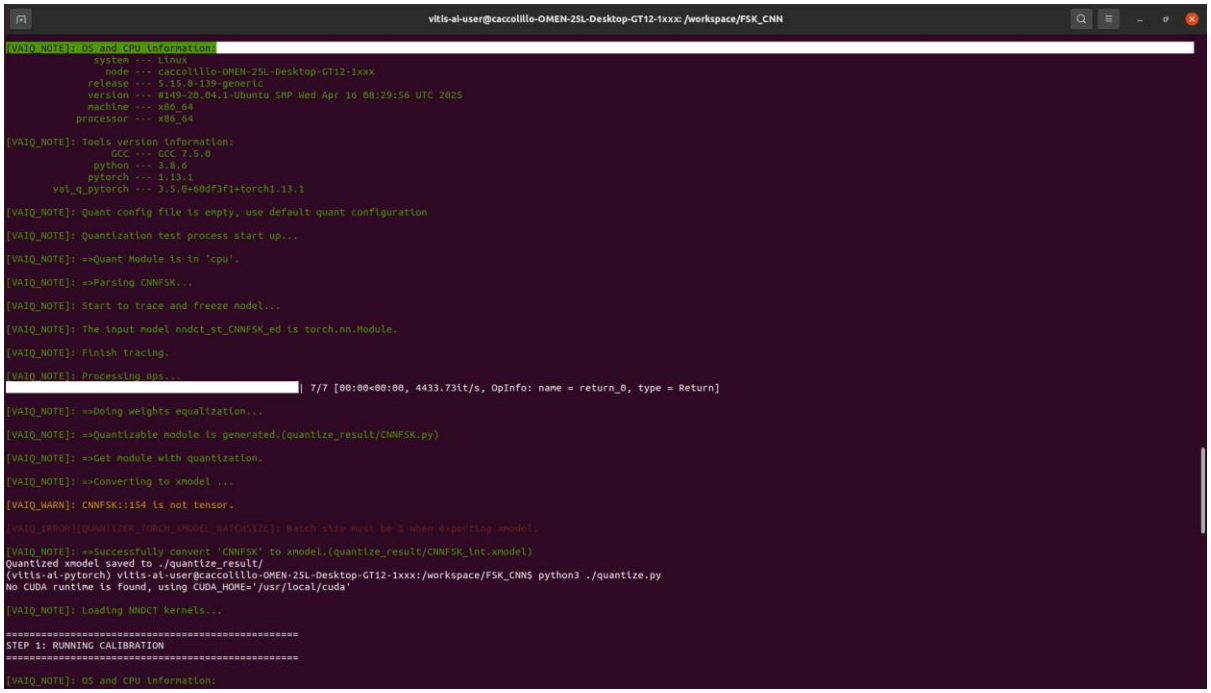


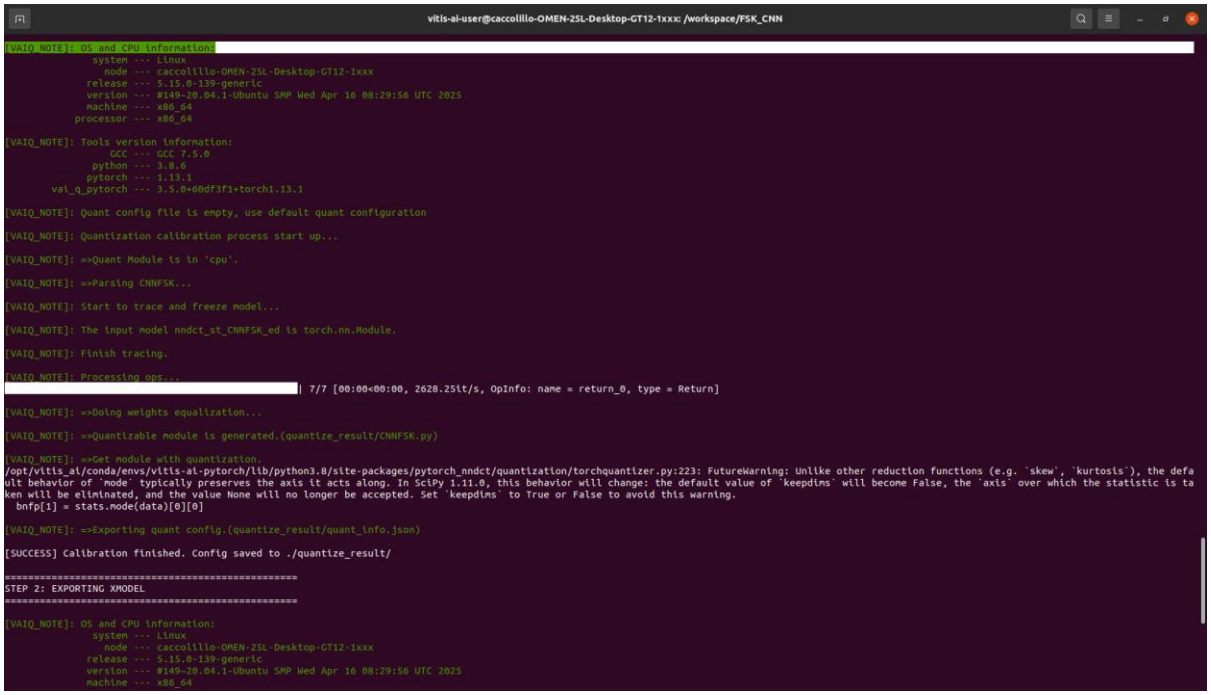
And created a folder where the script, the arch.json file describing the DPU in use and the trained model parameters obtained so far by using the CUDA capable GPU.



When we launch the script in the Vitis AI docker:




vitis-ai-user@caccolillo-OMEN-25L-Desktop-GT12-1xxx: /workspace/FSK_CNN
[VAIQ_NOTE]: OS and CPU information:
system --- Linux
node --- caccolillo-OMEN-25L-Desktop-GT12-1xxx
release --- 5.15.0-139-generic
version --- #149~20.04.1-Ubuntu SMP Wed Apr 16 08:29:56 UTC 2025
machine --- x86_64
processor --- x86_64
[VAIQ_NOTE]: Tools version information:
GCC --- GCC 7.5.0
python --- 3.8.6
pytorch --- 1.13.1
vai_q_pytorch --- 3.5.0+60df3f1+torch1.13.1
[VAIQ_NOTE]: Quant config file is empty, use default quant configuration
[VAIQ_NOTE]: Quantization test process start up...
[VAIQ_NOTE]: =>Quant Module is in 'cpu'.
[VAIQ_NOTE]: =>Parsing CNNFSK...
[VAIQ_NOTE]: Start to trace and freeze model...
[VAIQ_NOTE]: The input model nndct_st_CNNFSK_ed is torch.nn.Module.
[VAIQ_NOTE]: Finish tracing.
[VAIQ_NOTE]: Processing ops...
| 7/7 [00:00<00:00, 4433.73it/s, OpInfo: name = return_0, type = Return]
[VAIQ_NOTE]: =>Doing weights equalization...
[VAIQ_NOTE]: =>Quantizable module is generated.(quantize_result/CNNFSK.py)
[VAIQ_NOTE]: =>Get module with quantization.
[VAIQ_NOTE]: =>Converting to xmodel ...
[VAIQ_WARN]: CNNFSK::154 is not tensor.
[VAIQ_ERROR][QUANTIZER_TORCH_XMODEL_BATCHSIZE]: Batch size must be 1 when exporting xmodel.
[VAIQ_NOTE]: =>Successfully convert 'CNNFSK' to xmodel.(quantize_result/CNNFSK_int.xmodel)
Quantized xmodel saved to ./quantize_result/
(vitis-ai-pytorch) vitis-ai-user@caccolillo-OMEN-25L-Desktop-GT12-1xxx:/workspace/FSK_CNN$ python3 ./quantize.py
No CUDA runtime is found, using CUDA_HOME='/usr/local/cuda'
[VAIQ_NOTE]: Loading NNDCT kernels...
==
STEP 1: RUNNING CALIBRATION
==
[VAIQ_NOTE]: OS and CPU information:


vitis-ai-user@caccolillo-OMEN-25L-Desktop-GT12-1xxx: /workspace/FSK_CNN
[VAIQ_NOTE]: OS and CPU information:
system --- Linux
node --- caccolillo-OMEN-25L-Desktop-GT12-1xxx
release --- 5.15.0-139-generic
version --- #149~20.04.1-Ubuntu SMP Wed Apr 16 08:29:56 UTC 2025
machine --- x86_64
processor --- x86_64
[VAIQ_NOTE]: Tools version information:
GCC --- GCC 7.5.0
python --- 3.8.6
pytorch --- 1.13.1
vai_q_pytorch --- 3.5.0+60df3f1+torch1.13.1
[VAIQ_NOTE]: Quant config file is empty, use default quant configuration
[VAIQ_NOTE]: Quantization calibration process start up...
[VAIQ_NOTE]: =>Quant Module is in 'cpu'.
[VAIQ_NOTE]: =>Parsing CNNFSK...
[VAIQ_NOTE]: Start to trace and freeze model...
[VAIQ_NOTE]: The input model nndct_st_CNNFSK_ed is torch.nn.Module.
[VAIQ_NOTE]: Finish tracing.
[VAIQ_NOTE]: Processing ops...
| 7/7 [00:00<00:00, 2628.25it/s, OpInfo: name = return_0, type = Return]
[VAIQ_NOTE]: =>Doing weights equalization...
[VAIQ_NOTE]: =>Quantizable module is generated.(quantize_result/CNNFSK.py)
[VAIQ_NOTE]: =>Get module with quantization.
/opt/vitis_ai/conda/envs/vitis-ai-pytorch/lib/python3.8/site-packages/pytorch_nndct/quantization/torchquantizer.py:223: FutureWarning: Unlike other reduction functions (e.g. `skew`, `kurtosis`), the defa
ult behavior of `mode` typically preserves the axis it acts along. In SciPy 1.11.0, this behavior will change: the default value of `keepdims` will become False, the `axis` over which the statistic is ta
ken will be eliminated, and the value None will no longer be accepted. Set `keepdims` to True or False to avoid this warning.
bnfp[1] = stats.mode(data)[0][0]
[VAIQ_NOTE]: =>Exporting quant config.(quantize_result/quant_info.json)
[SUCCESS] Calibration finished. Config saved to ./quantize_result/
==
STEP 2: EXPORTING XMODEL
==
[VAIQ_NOTE]: OS and CPU information:
system --- Linux
node --- caccolillo-OMEN-25L-Desktop-GT12-1xxx
release --- 5.15.0-139-generic
version --- #149~20.04.1-Ubuntu SMP Wed Apr 16 08:29:56 UTC 2025
machine --- x86_64

```
            machine --- x86_64
          processor --- x86_64

[VAIQ_NOTE]: Tools version information:
                GCC --- GCC 7.5.0
             python --- 3.8.6
            pytorch --- 1.13.1
       vai_q_pytorch --- 3.5.0+60df3f1+torch1.13.1

[VAIQ_NOTE]: Quant config file is empty, use default quant configuration

[VAIQ_NOTE]: Quantization test process start up...

[VAIQ_NOTE]: =>Quant Module is in 'cpu'.

[VAIQ_NOTE]: =>Parsing CNNFSK...

[VAIQ_NOTE]: Start to trace and freeze model...

[VAIQ_NOTE]: The input model nndct_st_CNNFSK_ed is torch.nn.Module.

[VAIQ_NOTE]: Finish tracing.

[VAIQ_NOTE]: Processing ops...
| 7/7 [00:00<00:00, 4493.44it/s, OpInfo: name = return_0, type = Return]

[VAIQ_NOTE]: =>Doing weights equalization...

[VAIQ_NOTE]: =>Quantizable module is generated.(quantize_result/CNNFSK.py)

[VAIQ_NOTE]: =>Get module with quantization.

[VAIQ_NOTE]: =>Converting to xmodel ...

[VAIQ_WARN]: CNNFSK::154 is not tensor.

[VAIQ_NOTE]: =>Successfully convert 'CNNFSK' to xmodel.(quantize_result/CNNFSK_int.xmodel)

[SUCCESS] Quantized xmodel saved to ./quantize_result/CNNFSK_int.xmodel
(vitis-ai-pytorch) vitis-ai-user@caccolillo-OMEN-25L-Desktop-GT12-1xxx:/workspace/FSK_CNN$
```

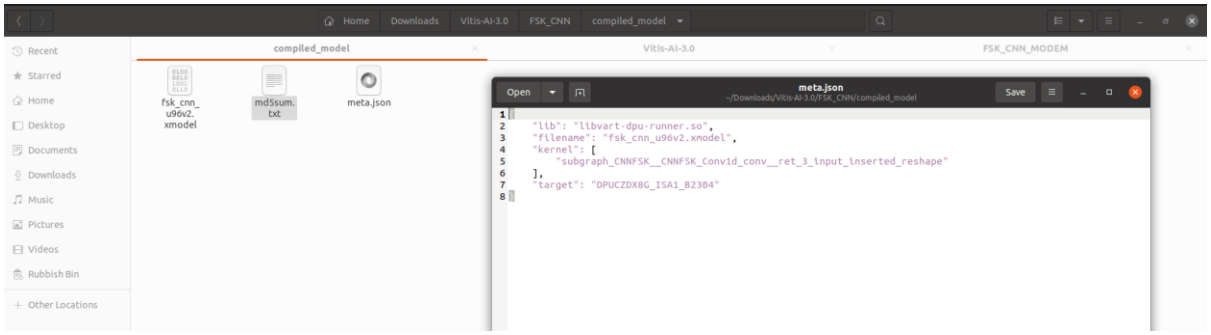
We have successfully quantized your model and generated the CNNFSK_int.xmodel. This file is a hardware-agnostic intermediate representation of the network:

The final step before running this on an a Zynq UltraScale+ is Compilation. The compiler (vai_c_pytorch) translates that .xmodel into specific instructions for the DPU (Deep Learning Processing Unit) on the target board.

vai_c_xir --xmodel quantize_result/CNNFSK_int.xmodel --arch arch.json --net_name fsk_cnn_u96v2 --output_dir compiled_model

```
ERROR: NO FRONT END SPECIFIED
(vitis-ai-pytorch) vitis-ai-user@caccolillo-OMEN-25L-Desktop-GT12-1xxx:/workspace/FSK_CNN$ vai_c_xir --xmodel quantize_result/CNNFSK_int.xmodel --arch arch.json --net_name fsk_cnn_u96
v2 --output_dir compiled_model
**************************************************
* VITIS_AI Compilation - Xilinx Inc.
**************************************************
[UNILOG][INFO] Compile mode: dpu
[UNILOG][INFO] Debug mode: null
[UNILOG][INFO] Target architecture: DPUCZDX8G_ISA1_B2304
[UNILOG][INFO] Graph name: CNNFSK, with op num: 20
[UNILOG][INFO] Begin to compile...
[UNILOG][INFO] Total device subgraph number 3, DPU subgraph number 1
[UNILOG][INFO] Compile done.
[UNILOG][INFO] The meta json is saved to "/workspace/FSK_CNN/compiled_model/meta.json"
[UNILOG][INFO] The compiled xmodel is saved to "/workspace/FSK_CNN/compiled_model/fsk_cnn_u96v2.xmodel"
[UNILOG][INFO] The compiled xmodel's md5sum is e73b01e5765ed9157285866fec801fa0, and has been saved to "/workspace/FSK_CNN/compiled_model/md5sum.txt"
(vitis-ai-pytorch) vitis-ai-user@caccolillo-OMEN-25L-Desktop-GT12-1xxx:/workspace/FSK_CNN$
```

In the end we get what follows:

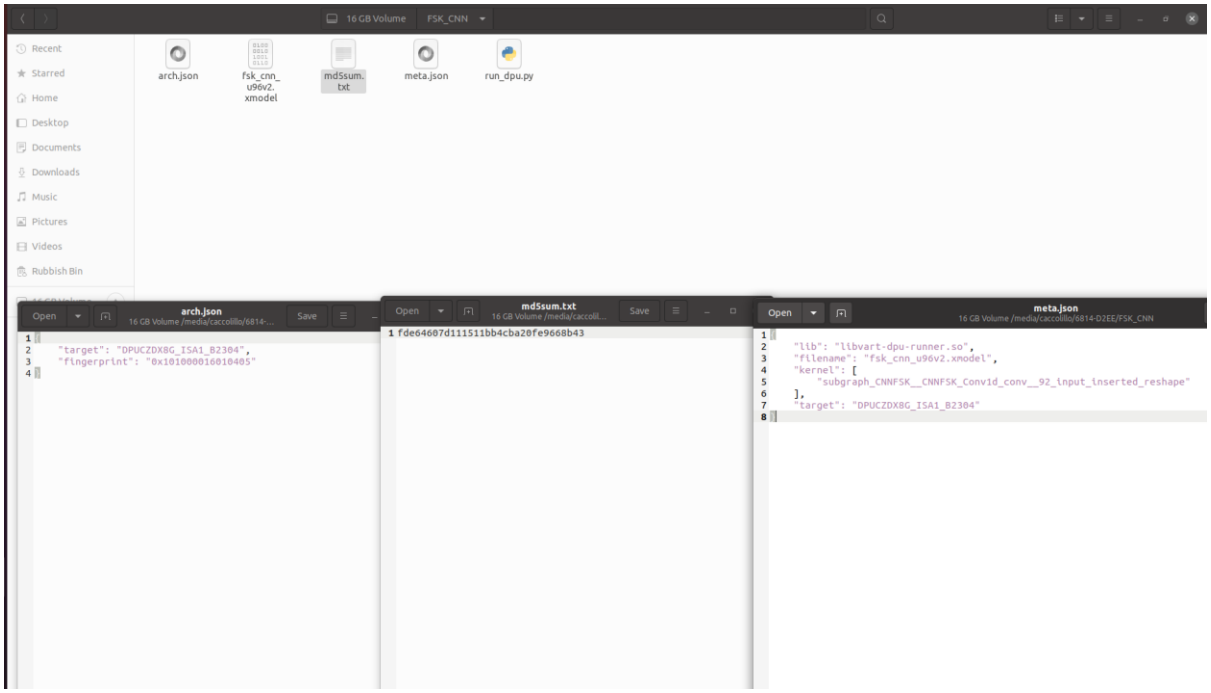


At the end of this process, we get a xmodel hardware-ready binary file "fsk_cnn_u96v2.xmodel".

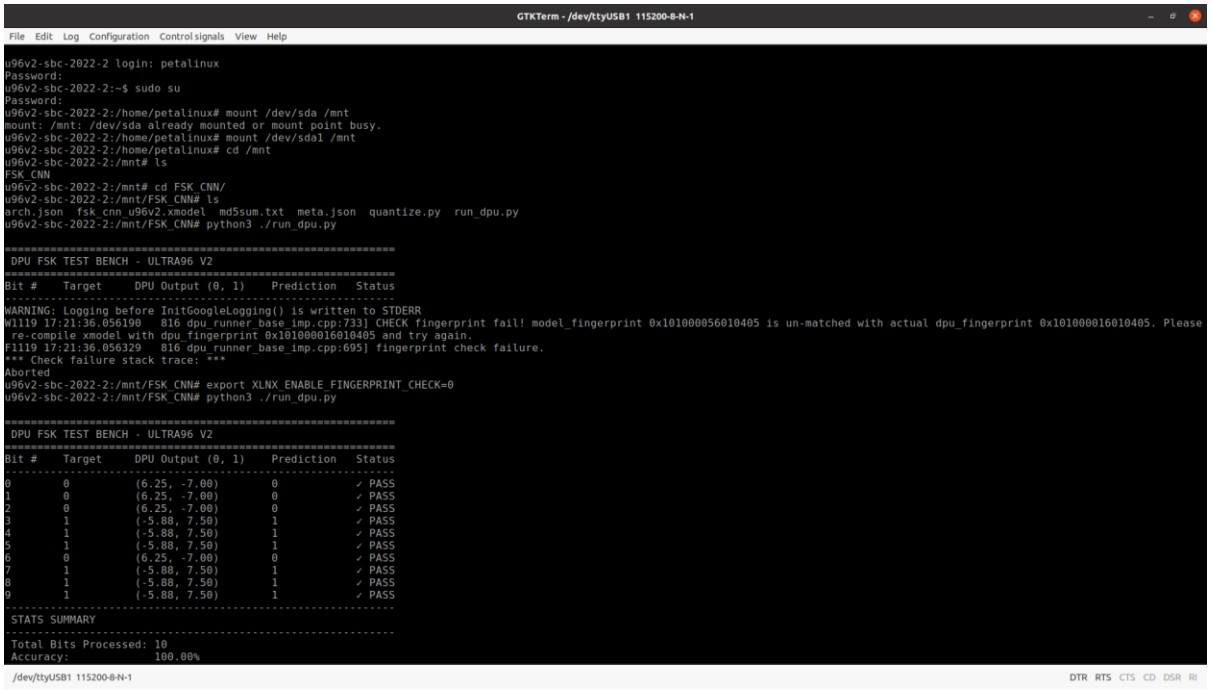
This is the "executable" for the FPGA. Unlike the .pth file (which is a list of weights) or the quantized .xmodel (which is a generic graph), the compiled .xmodel contains:

- DPU Instructions: A sequence of low-level commands telling the DPU exactly how to move data through its internal buffers.
- Hardware Schedule: A roadmap of which mathematical operations (convolutions, pooling) happen in which clock cycle.
- Quantized Weights: the 12 parameters converted into 8-bit integers, optimized for the B2304 architecture.

A USB stick has been prepared with the following files in it:



Then it’s plugged in the Ultra96 board and we run the python script:



To be noted that before running the python script, we need to give:

export XLNX_ENABLE_FINGERPRINT_CHECK=0

Due to issues with the fingerprint checkings.

The model running on the DPU behaves as expected: we have successfully ported and deployed the starting CNN described in the IEEE article on a Ultra96 v2 board, using a DPU for HW acceleration, using Vitis AI 3.0.

CONCLUSIONS

This project successfully demonstrated the complete workflow for implementing a CNN-based FSK demodulator on embedded FPGA hardware, validating the approach across multiple implementation methodologies. Key accomplishments include:

Training and Validation: A minimal CNN architecture with only 12 trainable parameters was successfully trained to demodulate binary FSK signals, achieving performance comparable to ideal demodulators even in noisy conditions. The training infrastructure, including synthetic dataset generation and quantization procedures, proved robust and repeatable.

HLS Implementation: The trained network was translated into a fully pipelined hardware implementation using Vitis HLS, featuring AXI-Stream interfaces and achieving an initiation interval of one sample. Both C simulation and C/RTL cosimulation validated functional correctness with bit-accurate matching to the Python reference model.

Vivado Verification: A comprehensive SystemVerilog testbench using AXI4-Stream VIP components provided thorough verification of the hardware implementation, confirming proper handshaking, timing, and demodulation accuracy across randomized test sequences.

DPU Deployment: The model was successfully quantized, compiled, and deployed on a Zynq UltraScale+ DPU using Vitis AI 3.0, demonstrating that the same trained weights can be leveraged for hardware-accelerated inference on specialized deep learning processing units.

Several important insights emerged during this implementation:

Intentional Undersampling Works: The deliberate aliasing strategy proved highly effective, reducing sampling rates by orders of magnitude while the CNN learned to correctly interpret the aliased baseband signals.

Extreme Minimalism is Viable: With only 12 parameters, this network demonstrates that effective signal processing can be achieved with remarkably small models, making neural network approaches practical even on resource-constrained embedded systems.

Multiple Implementation Paths: The same trained model can be deployed through direct HLS synthesis for standalone operation or through DPU acceleration for integration with processor-based systems, providing flexibility in system architecture.

Quantization Robustness: The 8-bit quantization process preserved model accuracy, confirming that fixed-point arithmetic is sufficient for this application and enabling efficient hardware implementation.

Verification Methodology: The progression from Python simulation through HLS C simulation, C/RTL cosimulation, and finally SystemVerilog testbench provided multiple validation checkpoints, catching issues early in the development cycle.

DMA Integration: While the standalone AXI-Stream implementation proved effective, integration with DMA controllers for processor-based systems requires additional consideration of stream width conversion and buffer management.

This project demonstrates that neural network-based signal processing is transitioning from research curiosity to practical implementation. The combination of minimal model complexity, robust performance, and multiple hardware deployment options suggests that machine learning techniques will increasingly complement or replace traditional DSP approaches in embedded communication systems.

The workflow established here—from Python training through multiple hardware implementation paths—provides a template for deploying other neural network applications on FPGA platforms, extending beyond communications to domains such as sensor fusion, image processing, and real-time control systems.

The complete documentation provided herein serves as both a technical reference for this specific implementation and a practical guide for engineers seeking to deploy neural networks on FPGA platforms, whether through direct HLS synthesis or through specialized accelerators like the Xilinx DPU. As machine learning continues to evolve, such implementations will play an increasingly important role in next-generation embedded systems.