

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>

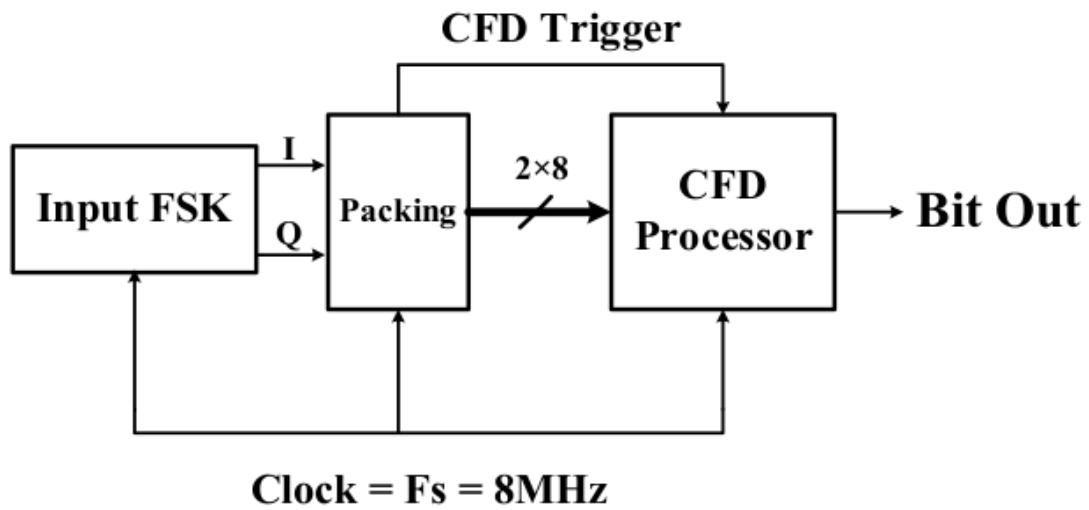
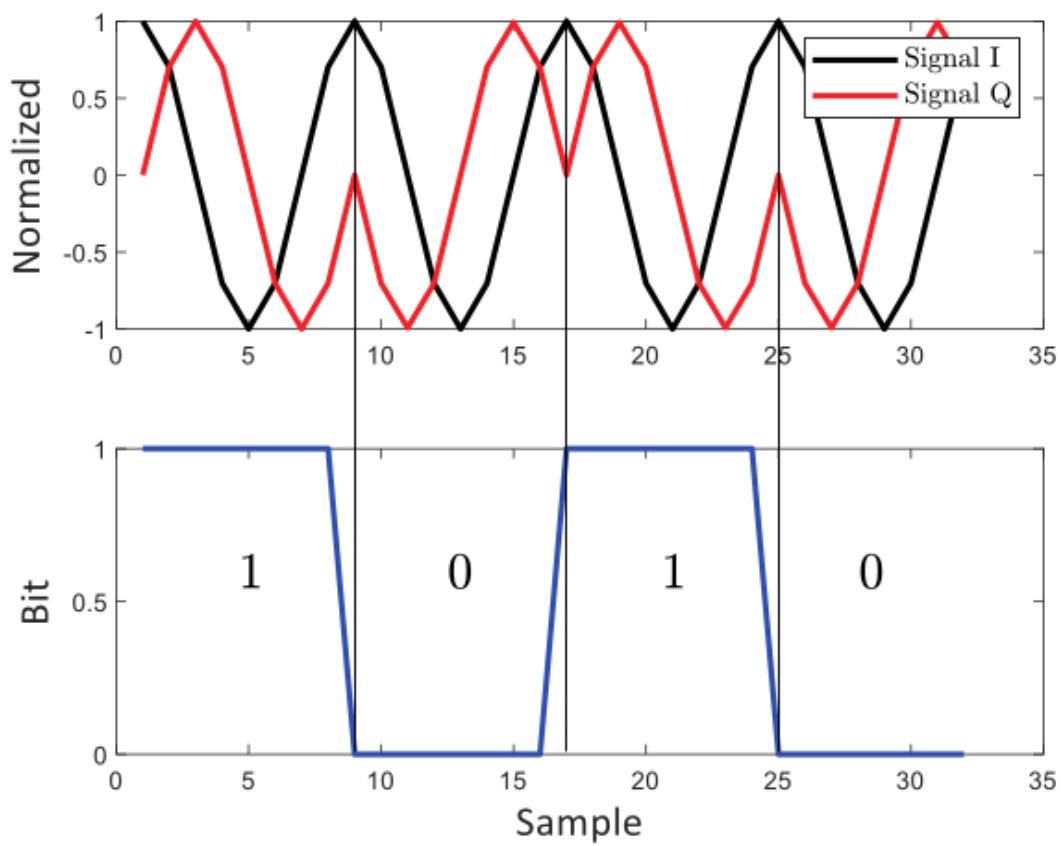


Fig. 4. CNN-Based FSK demodulator architecture

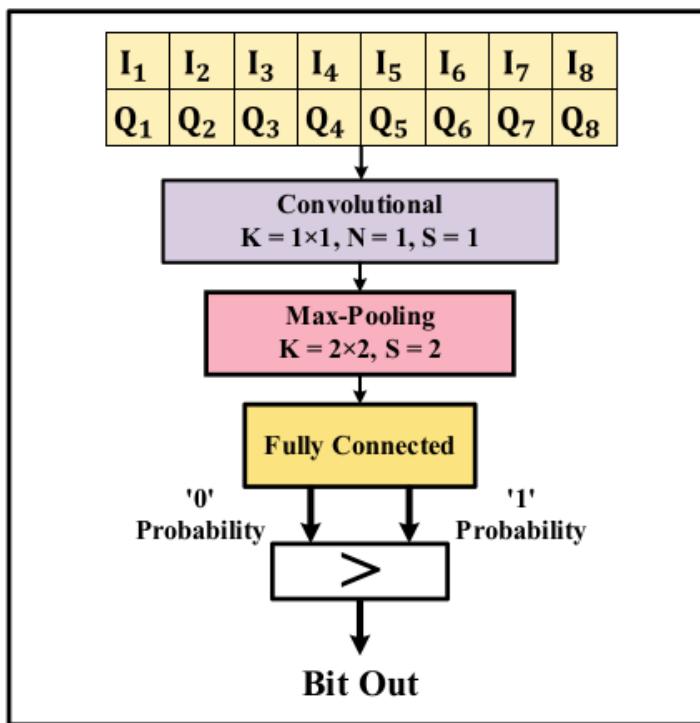


Fig. 5. The proposed convolutional neural network for data classification.

This paper presents an **ultra-compact CNN-based FSK demodulator** with only 12 trainable parameters that achieves comparable performance to ideal demodulators ($\text{BER} < 10^{-6}$ 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 ⚙️
1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4 from torch.utils.data import Dataset, DataLoader
5 import numpy as np
6
7 # Set random seeds for reproducibility
8 torch.manual_seed(42)
9 np.random.seed(42)
10
11 # FSK Signal Parameters
12 F_CARRIER = 28e6 # 28 MHz
13 F_BITRATE = 1e6 # 1 Mbps
14 F_DEVIATION = 1e6 # 1 MHz
15 F_SAMPLING = 8e6 # 8 MHz
16 SAMPLES_PER_SYMBOL = int(F_SAMPLING / F_BITRATE) # 8 samples
17
18 # Training Parameters
19 TRAIN_SAMPLES = 200000 # 2e5
20 VAL_SAMPLES = 50000 # 5e4
21 BATCH_SIZE = 256
22 EPOCHS = 50
23
24 class FSKDataset(Dataset):
25     """Generate FSK symbols with AWGN noise"""
26
27     def __init__(self, num_samples, eb_n0_range=(0, 25)):
28         self.num_samples = num_samples
29         self.eb_n0_min, self.eb_n0_max = eb_n0_range
30
31     def __len__(self):
32         return self.num_samples
33
34     def generate_fsk_symbol(self, bit, eb_n0_db):
35         """Generate one FSK symbol (I/Q components)"""
36         # Frequency for bit 0: 10MHz, bit 1: 23MHz
37         freq = F_CARRIER + (F_DEVIATION if bit == 1 else -F_DEVIATION)
38
39         # Time vector for one symbol
40         t = np.arange(SAMPLES_PER_SYMBOL) / F_SAMPLING
41
42         # Generate I and Q components (baseband after aliasing)
43         # Due to undersampling, the signal appears at baseband
44         baseband_freq = freq % F_SAMPLING
45         if baseband_freq > F_SAMPLING / 2:
46             baseband_freq = F_SAMPLING - baseband_freq
47
48         # Adjust for the actual aliased frequencies
49         if bit == 0:
50             phase_shift = 2 * np.pi * (-F_DEVIATION) / F_SAMPLING
51         else:
52             phase_shift = 2 * np.pi * F_DEVIATION / F_SAMPLING
53
54         I = np.cos(phase_shift * np.arange(SAMPLES_PER_SYMBOL))
55         Q = np.sin(phase_shift * np.arange(SAMPLES_PER_SYMBOL))
56
57         # Add AWGN noise
58         eb_n0_linear = 10 ** (eb_n0_db / 10)
59         noise_power = 1 / (2 * eb_n0_linear)
60         noise_std = np.sqrt(noise_power)
61
62         I += np.random.normal(0, noise_std, SAMPLES_PER_SYMBOL)
63         Q += np.random.normal(0, noise_std, SAMPLES_PER_SYMBOL)
64
65         # Stack I and Q into 2x8 array
66         symbol = np.stack([I, Q], axis=0)
67

```

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

```
1 #!/bin/bash
2
3 echo "== FSK CNN Training Environment Setup =="
4 echo ""
5
6 # Create virtual environment
7 echo "Creating virtual environment..."
8 python3 -m venv fsk_cnn_env
9
10 # Activate virtual environment
11 echo "Activating virtual environment..."
12 source fsk_cnn_env/bin/activate
13
14 # Upgrade pip
15 echo "Upgrading pip..."
16 pip install --upgrade pip
17
18 # Install PyTorch 1.11 with CUDA 11.3
19 echo "Installing PyTorch 1.11 with CUDA 11.3..."
20 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
21
22 # Install numpy
23 echo "Installing numpy..."
24 pip install numpy
25
26 # Verify installation
27 echo ""
28 echo "== Verifying Installation =="
29 python -c "import torch; print(f'Torch: {torch.__version__}')"; print(f'CUDA available: {torch.cuda.is_available()}'); print(f'GPU device: {torch.cuda.get_device_name(0)} if torch.cuda.is_available() else \"N/A\"')"
30
31 echo ""
32 echo "== Setup Complete! =="
33 echo "Virtual environment is activated."
34 echo "You can now run: python train_fsk_cnn.py"
35 echo ""
36 echo "To deactivate later, run: deactivate"
37 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              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:

```
caccolillo@caccolillo-OptiPlex-5090:~/Desktop/GT12-xxxx-/FSK_CNN_MODEN/PYTHON$ source ./setup_env.sh
*** FSK CNN Training Environment Setup ***

Creating virtual environment...
The virtual environment was not created successfully because ensurepip is not
available. On Debian/Ubuntu systems, you need to install the python3-venv
package using the following command.

    apt install python3.8-venv

You may need to use sudo with that command. After installing the python3-venv
package, recreate your virtual environment.

Failing command: ['/home/caccolillo/FSK_CNN_MODEN/PYTHON/fsk_cnn_env/bin/python3', '-Im', 'ensurepip', '--upgrade', '--default-pip']

Activating virtual environment...
bash: fsk_cnn_env/bin/activate: No such file or directory
Upgrading pip...
Collecting pip
  Downloading pip-25.0.1-py3-none-any.whl (1.8 MB)
Installing collected packages: pip
Successfully installed pip-25.0.1
Installing Pytorch 1.11 with CUDA 11.3...
Defaulting to user installation because normal site-packages is not writeable
Looking in indexes: https://pypi.org/simple, https://download.pytorch.org/wheel/cu113
Collecting torchcu11.11.0cu113
  Using cached https://download.pytorch.org/whl/cu113/torch-1.11.0%2Bcu113-cp38-cp38-linux_x86_64.whl (1637.0 MB)
Collecting torchvision==0.12.0+cu113
  Using cached https://download.pytorch.org/whl/cu113/torchvision-0.12.0%2Bcu113-cp38-cp38-linux_x86_64.whl (22.3 MB)
Collecting torchaudio==0.11.0
  Using cached https://download.pytorch.org/whl/cu113/torchaudio-0.11.0%2Bcu113-cp38-cp38-linux_x86_64.whl (2.9 MB)
Collecting typing_extensions (from torchcu11.11.0cu113)
  Using cached https://download.pytorch.org/whl/typing_extensions-4.15.0-py3-none-any.whl.metadata (3.3 kB)
Requirement already satisfied: numpy in /usr/lib/python3/dist-packages (from torchcu11.11.0cu113) (1.17.4)
Requirement already satisfied: requests in /usr/lib/python3/dist-packages (from torchcu11.11.0cu113) (2.22.0)
Requirement already satisfied: pillow<8.3,*,>=5.3.0 in /usr/lib/python3/dist-packages (from torchcu11.11.0cu113) (7.0.0)
INFO: pip is looking at multiple versions of typing-extensions to determine which version is compatible with other requirements. This could take a while.
Requirement already satisfied: requests in /usr/lib/python3/dist-packages (from torchcu11.11.0cu113) (2.22.0)
Requirement already satisfied: numpy in /usr/lib/python3/dist-packages (from torchcu11.11.0cu113) (1.17.4)
Using cached https://download.pytorch.org/whl/typing_extensions-4.14.0-py3-none-any.whl.metadata (3.0 kB)
Using cached https://download.pytorch.org/whl/typing_extensions-4.13.2-py3-none-any.whl (45 kB)
Using cached typing_extensions-4.13.2-py3-none-any.whl.metadata (3.0 kB)
Installing collected packages: typing-extensions, torch, torchvision, torchaudio
Successfully installed torch-1.11.0+cu113 torchaudio-0.11.0+cu113 torchvision-0.12.0+cu113 typing-extensions-4.13.2
Installing numpy...
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: numpy in /usr/lib/python3/dist-packages (1.17.4)

*** Verifying Installation ***
  File "<string>", line 1
    import torch; print(f'PyTorch: {torch.__version__}') ; print(f'CUDA available: {torch.cuda.is_available()}') ; print(f'CUDNN device: {torch.cuda.get_device_name(0)} if torch.cuda.is_available() else "None"')
                                         ^
SyntaxError: invalid syntax

*** Setup Complete! ***
Virtual environment is activated.
You can now run: python train_fsk_cnn.py

To deactivate later, run: deactivate
To reactivate, run: source fsk_cnn_env/bin/activate
caccolillo@caccolillo-OptiPlex-5090:~/Desktop/GT12-xxxx-/FSK_CNN_MODEN/PYTHON$
```

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

```

Activities Terminal 5 Jan 18:13
caccoillio@caccoillio-OMEN-2SL-Desktop-GT12-1xxx:~/FSK_CNN_MODEM/PYTHON$ python train_net.py
File "train_net.py", line 188
    print("Float parameters saved to '%(filename)''")
                                          ^
SyntaxError: invalid syntax
caccoillio@caccoillio-OMEN-2SL-Desktop-GT12-1xxx:~/FSK_CNN_MODEM/PYTHON$ python3 train_net.py
Using device: cuda
Creating datasets...
Model created with 13 parameters
Expected: 12 parameters (1 conv weight + 1 conv bias + 8 fc weights + 2 fc biases)
A Starting training...
Epoch [1/50] Train Loss: 0.2321, Train Acc: 94.89% | Val Loss: 0.0497, Val Acc: 99.60%
-> Saved best model with validation accuracy: 99.60%
Epoch [2/50] Train Loss: 0.0272, Train Acc: 99.71% | Val Loss: 0.0150, Val Acc: 99.81%
-> Saved best model with validation accuracy: 99.81%
Epoch [3/50] Train Loss: 0.0183, Train Acc: 99.80% | Val Loss: 0.0088, Val Acc: 99.83%
-> Saved best model with validation accuracy: 99.83%
Epoch [4/50] Train Loss: 0.0071, Train Acc: 99.86% | Val Loss: 0.0069, Val Acc: 99.83%
-> Saved best model with validation accuracy: 99.83%
Epoch [5/50] Train Loss: 0.0052, Train Acc: 99.88% | Val Loss: 0.0046, Val Acc: 99.88%
-> Saved best model with validation accuracy: 99.88%
Epoch [6/50] Train Loss: 0.0038, Train Acc: 99.91% | Val Loss: 0.0036, Val Acc: 99.89%
-> Saved best model with validation accuracy: 99.89%
Epoch [7/50] Train Loss: 0.0031, Train Acc: 99.91% | Val Loss: 0.0024, Val Acc: 99.93%
-> Saved best model with validation accuracy: 99.93%
Epoch [8/50] Train Loss: 0.0030, Train Acc: 99.91% | Val Loss: 0.0023, Val Acc: 99.91%
Epoch [9/50] Train Loss: 0.0029, Train Acc: 99.91% | Val Loss: 0.0026, Val Acc: 99.92%
Epoch [10/50] Train Loss: 0.0022, Train Acc: 99.91% | Val Loss: 0.0022, Val Acc: 99.93%
Epoch [11/50] Train Loss: 0.0021, Train Acc: 99.92% | Val Loss: 0.0023, Val Acc: 99.93%
Epoch [12/50] Train Loss: 0.0020, Train Acc: 99.93% | Val Loss: 0.0024, Val Acc: 99.90%
Epoch [13/50] Train Loss: 0.0019, Train Acc: 99.93% | Val Loss: 0.0025, Val Acc: 99.92%
Epoch [14/50] Train Loss: 0.0019, Train Acc: 99.93% | Val Loss: 0.0025, Val Acc: 99.91%
Epoch [15/50] Train Loss: 0.0021, Train Acc: 99.92% | Val Loss: 0.0025, Val Acc: 99.92%
Epoch [16/50] Train Loss: 0.0026, Train Acc: 99.93% | Val Loss: 0.0024, Val Acc: 99.92%
Epoch [17/50] Train Loss: 0.0022, Train Acc: 99.93% | Val Loss: 0.0017, Val Acc: 99.94%
-> Saved best model with validation accuracy: 99.94%
Epoch [18/50] Train Loss: 0.0019, Train Acc: 99.93% | Val Loss: 0.0013, Val Acc: 99.93%
Epoch [19/50] Train Loss: 0.0019, Train Acc: 99.93% | Val Loss: 0.0024, Val Acc: 99.91%
Epoch [20/50] Train Loss: 0.0022, Train Acc: 99.92% | Val Loss: 0.0017, Val Acc: 99.92%
Epoch [21/50] Train Loss: 0.0023, Train Acc: 99.92% | Val Loss: 0.0024, Val Acc: 99.93%
Epoch [22/50] Train Loss: 0.0020, Train Acc: 99.92% | Val Loss: 0.0025, Val Acc: 99.92%
Epoch [23/50] Train Loss: 0.0022, Train Acc: 99.92% | Val Loss: 0.0025, Val Acc: 99.94%
Epoch [24/50] Train Loss: 0.0022, Train Acc: 99.92% | Val Loss: 0.0023, Val Acc: 99.92%
Epoch [25/50] Train Loss: 0.0021, Train Acc: 99.93% | Val Loss: 0.0022, Val Acc: 99.92%
Epoch [26/50] Train Loss: 0.0025, Train Acc: 99.91% | Val Loss: 0.0024, Val Acc: 99.92%
Epoch [27/50] Train Loss: 0.0020, Train Acc: 99.94% | Val Loss: 0.0015, Val Acc: 99.95%
-> Saved best model with validation accuracy: 99.95%
Epoch [28/50] Train Loss: 0.0021, Train Acc: 99.92% | Val Loss: 0.0019, Val Acc: 99.94%
Epoch [29/50] Train Loss: 0.0023, Train Acc: 99.92% | Val Loss: 0.0022, Val Acc: 99.92%
Epoch [30/50] Train Loss: 0.0026, Train Acc: 99.93% | Val Loss: 0.0025, Val Acc: 99.91%
Epoch [31/50] Train Loss: 0.0023, Train Acc: 99.92% | Val Loss: 0.0030, Val Acc: 99.96%
Epoch [32/50] Train Loss: 0.0026, Train Acc: 99.92% | Val Loss: 0.0029, Val Acc: 99.95%
Epoch [33/50] Train Loss: 0.0022, Train Acc: 99.92% | Val Loss: 0.0021, Val Acc: 99.92%
Epoch [34/50] Train loss: 0.0023, Train Acc: 99.92% | Val loss: 0.0026, Val Acc: 99.92%

```

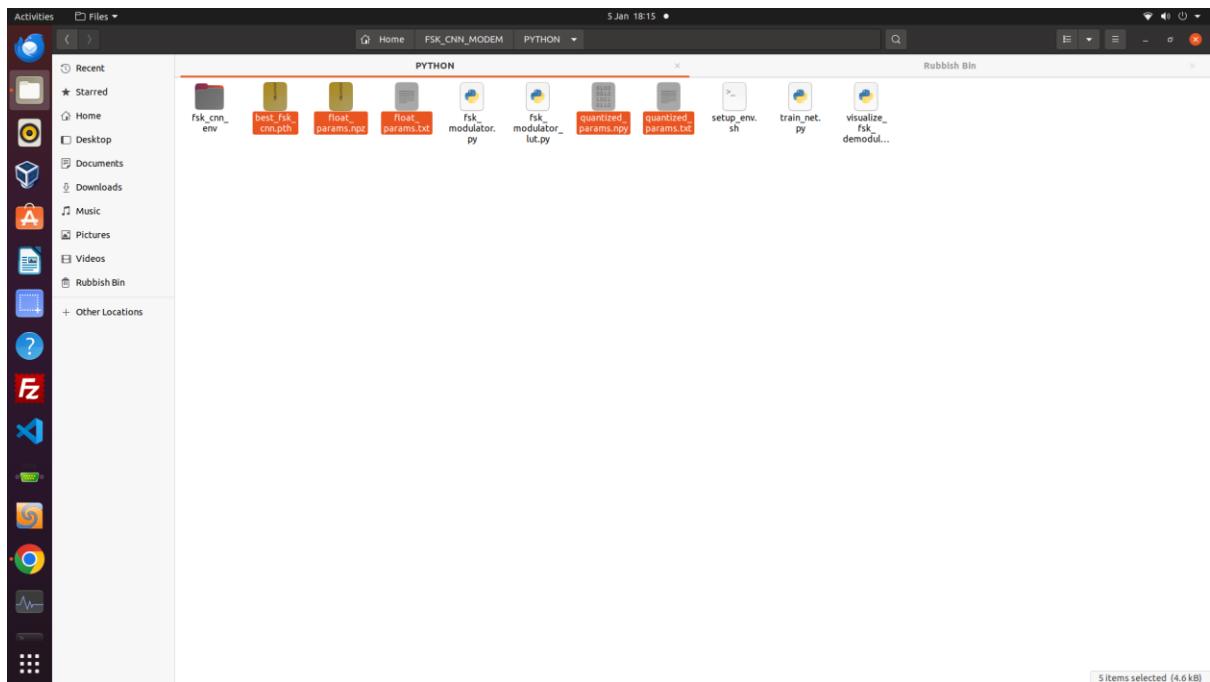
```

Activities Terminal 5 Jan 18:14
caccoillio@caccoillio-OMEN-2SL-Desktop-GT12-1xxx:~/FSK_CNN_MODEM/PYTHON$ python3 train_net.py
Epoch [35/50] Train Loss: 0.0019, Train Acc: 99.93% | Val Loss: 0.0032, Val Acc: 99.91%
Epoch [36/50] Train Loss: 0.0018, Train Acc: 99.93% | Val Loss: 0.0018, Val Acc: 99.94%
Epoch [37/50] Train Loss: 0.0019, Train Acc: 99.93% | Val Loss: 0.0019, Val Acc: 99.94%
Epoch [38/50] Train Loss: 0.0021, Train Acc: 99.93% | Val Loss: 0.0019, Val Acc: 99.94%
Epoch [39/50] Train Loss: 0.0023, Train Acc: 99.92% | Val Loss: 0.0019, Val Acc: 99.95%
Epoch [40/50] Train Loss: 0.0021, Train Acc: 99.93% | Val Loss: 0.0020, Val Acc: 99.93%
Epoch [41/50] Train Loss: 0.0019, Train Acc: 99.91% | Val Loss: 0.0026, Val Acc: 99.91%
Epoch [42/50] Train Loss: 0.0019, Train Acc: 99.92% | Val Loss: 0.0027, Val Acc: 99.91%
Epoch [43/50] Train Loss: 0.0021, Train Acc: 99.93% | Val Loss: 0.0029, Val Acc: 99.92%
Epoch [44/50] Train Loss: 0.0024, Train Acc: 99.91% | Val Loss: 0.0020, Val Acc: 99.91%
Epoch [45/50] Train Loss: 0.0022, Train Acc: 99.92% | Val Loss: 0.0026, Val Acc: 99.92%
Epoch [46/50] Train Loss: 0.0020, Train Acc: 99.93% | Val Loss: 0.0026, Val Acc: 99.92%
Epoch [47/50] Train Loss: 0.0018, Train Acc: 99.93% | Val Loss: 0.0029, Val Acc: 99.92%
Epoch [48/50] Train Loss: 0.0018, Train Acc: 99.94% | Val Loss: 0.0029, Val Acc: 99.93%
Epoch [49/50] Train Loss: 0.0023, Train Acc: 99.91% | Val Loss: 0.0022, Val Acc: 99.91%
Epoch [50/50] Train Loss: 0.0023, Train Acc: 99.92% | Val Loss: 0.0019, Val Acc: 99.93%
Training completed. Best validation accuracy: 99.95%
=====
Saving float32 parameters...
Float parameters saved to 'float_params.npy'
Float parameters (text) saved to 'float_params.txt'
=====
Quantizing model to FPS...
train_net.py:219: RuntimeWarning: Invalid value encountered in true_divide
  param_scaled = (param_np - param_min) / (param_max - param_min)
Quantized parameters saved to 'quantized_params.npy'
Quantized parameters (text) saved to 'quantized_params.txt'
=====
== Parameter Summary ==
=====
Float32 Parameters:
conv.weight: shape=(1, 2, 1), range=[0.006389, 2.499234]
conv.bias: shape=(1,), ranges=[-0.628164, -0.628164]
fc.weight: shape=(2, 4), range=[-1.293140, 1.505391]
fc.bias: shape=(2,), range=[0.222194, 0.240754]
=====
Quantized (FPS) Parameters:
conv.weight: quantized_range=[-15, 15]
conv.bias: quantized_range=[0, 0]
fc.weight: quantized_range=[-15, 15]
fc.bias: quantized_range=[-15, 15]
=====
Files saved:
1. best_fsk_cnn.pth - PyTorch model weights
2. float_params.npy - Float32 parameters (numpy)
3. float_params.txt - Float32 parameters (readable)
4. quantized_params.npy - Quantized FPS parameters
5. quantized_params.txt - Quantized FPS parameters (readable)
=====

caccoillio@caccoillio-OMEN-2SL-Desktop-GT12-1xxx:~/FSK_CNN_MODEM/PYTHON$ 

```

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



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 

```
1 import torch
2 import torch.nn as nn
3 import numpy as np
4 import matplotlib.pyplot as plt
5
6 # FSK Signal Parameters
7 F_CARRIER = 20e6 # 20 MHz
8 F_BITRATE = 1e6 # 1 Mbps
9 F_DEVIATION = 1e6 # 1 MHz
10 F_SAMPLING = 8e6 # 8 MHz
11 SAMPLES_PER_SYMBOL = int(F_SAMPLING / F_BITRATE) # 8 samples
12
13 class CNNFSK(nn.Module):
14     """Compact CNN for FSK demodulation - only 12 parameters"""
15
16     def __init__(self):
17         super(CNNFSK, self).__init__()
18
19         # Convolutional layer: 1x1 kernel, 1 filter, stride=1
20         self.conv = nn.Conv1d(in_channels=2, out_channels=1,
21                             kernel_size=1, stride=1, bias=True)
22
23         # Max pooling: 2x2 window, stride=2
24         self.maxpool = nn.MaxPool1d(kernel_size=2, stride=2)
25
26         # Fully connected layer: 4 inputs, 2 outputs (bit 0 or 1)
27         self.fc = nn.Linear(4, 2)
28
29     def forward(self, x):
30         # x shape: (batch, 2, 8)
31         x = self.conv(x) # (batch, 1, 8)
32         x = self.maxpool(x) # (batch, 1, 4)
33         x = x.view(x.size(0), -1) # (batch, 4)
34         x = self.fc(x) # (batch, 2)
35
36         return x
37
38     def generate_fsk_symbol(self, eb_n0_db):
39         """Generate one FSK symbol (I/Q components) with AWGN noise"""
40         # Frequency for bit 0: 19MHz, bit 1: 21MHz
41         freq = F_CARRIER + (F_DEVIATION if bit == 1 else -F_DEVIATION)
42
43         # Adjust for the actual aliased frequencies at baseband
44         if bit == 0:
45             phase_shift = 2 * np.pi * (-F_DEVIATION) / F_SAMPLING
46         else:
47             phase_shift = 2 * np.pi * F_DEVIATION / F_SAMPLING
48
49         # Generate I and Q components
50         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.

```
caccoollo@caccoollo-OMEN-2SL-Desktop-GT12-1xx:~/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
=====

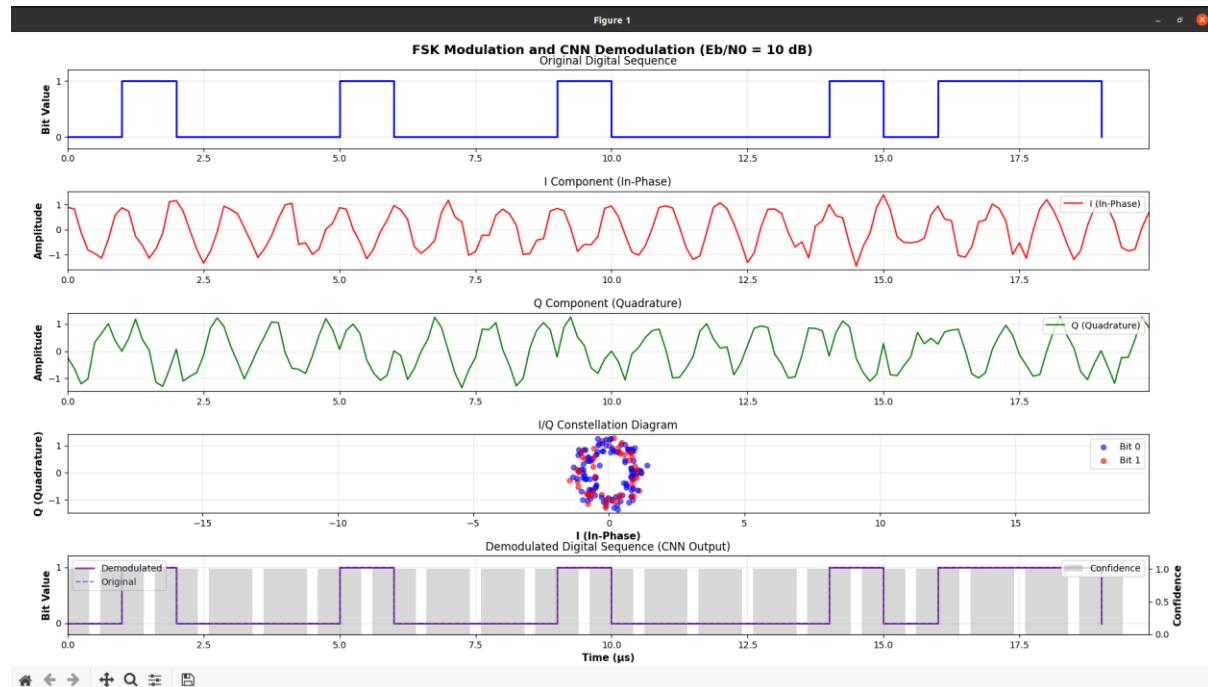
Bit #    Original    Demodulated    Prob[0]    Prob[1]    Match
-----
0        0          0            1.0000    0.0000    ✓
1        1          1            0.0000    1.0000    ✓
2        0          0            1.0000    0.0000    ✓
3        0          0            1.0000    0.0000    ✓
4        0          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](#) ↗

 **caccollito** Added python inference script using quantized coefficients.

[Code](#) [Blame](#) 87 lines (72 loc) · 2.94 KB 

```
1 import numpy as np
2
3 # === 1. Load quantized parameters ===
4 quantized_params = np.load('quantized_params.npy', allow_pickle=True).item()
5 print("Loaded quantized parameters from 'quantized_params.npy'")
6
7 # === 2. Tiny FFS CNN inference function ===
8 def cnn_fsk_fp5(symbol, quantized_params):
9     conv_w = quantized_params['conv.weight']['quantized'] # (1,2,1)
10    conv_b = quantized_params['conv.bias']['quantized'] # (1,)
11    scale_w = quantized_params['conv.weight']['scale']
12    scale_b = quantized_params['conv.bias']['scale']
13
14    conv_out = np.zeros(8, dtype=np.float32)
15    for t in range(8):
16        conv_sum = 0
17        for i in range(2):
18            conv_sum += symbol[i, t] * conv_w[0, i, 0] * scale_w
19        conv_sum += conv_b[0] * scale_b
20        conv_out[t] = conv_sum
21
22    # MaxPoolId
23    pool_out = np.array([max(conv_out[0], conv_out[1]),
24                        max(conv_out[2], conv_out[3]),
25                        max(conv_out[4], conv_out[5]),
26                        max(conv_out[6], conv_out[7]))]
27
28    # Fully connected
29    fc_w = quantized_params['fc.weight']['quantized'] # (2,4)
30    fc_b = quantized_params['fc.bias']['quantized'] # (2,)
31    scale_w_fc = quantized_params['fc.weight']['scale']
32    scale_b_fc = quantized_params['fc.bias']['scale']
33
34    fc_out = np.zeros(2, dtype=np.float32)
35    for j in range(2):
36        s = 0
37        for i in range(4):
38            s += pool_out[i] * fc_w[j, i] * scale_w_fc
39        s += fc_b[j] * scale_b_fc
40        fc_out[j] = s
41
42    # Softmax
43    exp_out = np.exp(fc_out - np.max(fc_out))
44    probs = exp_out / np.sum(exp_out)
45    pred_bit = np.argmax(probs)
46
47    return pred_bit, probs
48
49 # === 3. FSK Signal Parameters ===
50 SAMPLES_PER_SYMBOL = 8
```

It uses the quantized coefficients discussed so far:

```
Open ▾ + quantized_params.txt
~/FSK_CNN_MODEM/PYTHON

1 === Quantized Parameters (FP5) for FPGA Implementation ===
2
3 conv.weight:
4   Quantized values (int8): [[[ -15
5   [ 15]]]
6   Min: 0.006389
7   Max: 2.499234
8   Scale: 0.083095
9
10 conv.bias:
11   Quantized values (int8): [0]
12   Min: -0.628164
13   Max: -0.628164
14   Scale: 0.000000
15
16 fc.weight:
17   Quantized values (int8): [[ -2 -14    3   15]
18   [ 8   15  -8 -15]]
19   Min: -1.293140
20   Max: 1.505391
21   Scale: 0.093284
22
23 fc.bias:
24   Quantized values (int8): [-15   15]
25   Min: 0.222194
26   Max: 0.240754
27   Scale: 0.000619
28
```

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:

```
1 Synthesis Summary(solution1)  fsk_cnn_demod.cpp x
2 #include "fsk_cnn_demod.hpp"
3 #include <hls_math.hpp>
4
5 // Optimized helper functions for dataflow
6
7 static void read_iq_parallel(
8     hls::stream<axis_data> &input_I,
9     hls::stream<axis_data> &input_Q,
10    fixed_t I_buffer[SAMPLES_PER_SYMBOL],
11    fixed_t Q_buffer[SAMPLES_PER_SYMBOL]
12 ) {
13 #pragma HLS INLINE off
14     // Read I and Q samples in parallel with II=1
15     READ_IQ: for (int i = 0; i < SAMPLES_PER_SYMBOL; i++) {
16 #pragma HLS PIPELINE II=1
17         axis_data i_data = input_I.read();
18         axis_data q_data = input_Q.read();
19         I_buffer[i] = i_data.data;
20         Q_buffer[i] = q_data.data;
21     }
22 }
23
24 static void compute_convolution(
25     fixed_t I_buffer[SAMPLES_PER_SYMBOL],
26     fixed_t Q_buffer[SAMPLES_PER_SYMBOL],
27     acc_t conv_out[SAMPLES_PER_SYMBOL]
28 ) {
29 #pragma HLS INLINE off
30
31     // Precompute scaled weights (constant folding optimization)
32     const acc_t w_i = (acc_t)CONV_WEIGHT_QUANT[0][0] * CONV_WEIGHT_SCALE;
33     const acc_t w_q = (acc_t)CONV_WEIGHT_QUANT[0][1] * CONV_WEIGHT_SCALE;
34     const acc_t bias = (acc_t)CONV_BIAS_QUANT[0] * CONV_BIAS_SCALE;
35
36     CONV: for (int t = 0; t < SAMPLES_PER_SYMBOL; t++) {
37 #pragma HLS PIPELINE II=1
38 #pragma HLS UNROLL factor=2
39         acc_t sum = bias;
40         sum += (acc_t)I_buffer[t] * w_i;
41         sum += (acc_t)Q_buffer[t] * w_q;
42         conv_out[t] = sum;
43     }
44 }
45
46 static void compute_pooling(
47     acc_t conv_out[SAMPLES_PER_SYMBOL],
48     fixed_t pooled[POOLED_SIZE]
49 ) {
50
51     POOL: for (int i = 0; i < POOLED_SIZE; i++) {
52 #pragma HLS PIPELINE II=1
53 #pragma HLS UNROLL factor=2
54         int idx = i << 1; // i * 2
55         pooled[i] = (conv_out[idx] > conv_out[idx + 1]) ?
56             (fixed_t)conv_out[idx] :
57             (fixed_t)conv_out[idx + 1];
58     }
59 }
60
61 static void compute_fc(
62     fixed_t pooled[POOLED_SIZE],
63     acc_t fc_out[FC_OUT]
64 ) {
65 #pragma HLS INLINE off
66
67     // Precompute scaled biases
68     const acc_t bias_0 = (acc_t)FC_BIAS_QUANT[0] * FC_BIAS_SCALE;
69     const acc_t bias_1 = (acc_t)FC_BIAS_QUANT[1] * FC_BIAS_SCALE;
70
71     // Fully unrolled FC computation (only 2 outputs, 4 inputs)
72     acc_t acc_0 = bias_0;
73     acc_t acc_1 = bias_1;
74
75     FC COMPUTE: for (int i = 0; i < FC_IN; i++) {
76 #pragma HLS UNROLL
77         acc_t weight_0 = (acc_t)FC_WEIGHT_QUANT[0][i] * FC_WEIGHT_SCALE;
78         acc_t weight_1 = (acc_t)FC_WEIGHT_QUANT[1][i] * FC_WEIGHT_SCALE;
79
80         acc_0 += (acc_t)pooled[i] * weight_0;
81         acc_1 += (acc_t)pooled[i] * weight_1;
82     }
83
84     fc_out[0] = acc_0;
85     fc_out[1] = acc_1;
86 }
87
88 static void output_decision(
89     acc_t fc_out[FC_OUT],
90     hls::stream<axis_output> &predicted_bit
91 ) {
92 #pragma HLS INLINE off
93 }
```

```
1 Synthesis Summary(solution1)  fsk_cnn_demod.cpp x
2
3     fixed_t pooled[POOLED_SIZE]
4 ) {
5 #pragma HLS INLINE off
6
7     POOL: for (int i = 0; i < POOLED_SIZE; i++) {
8 #pragma HLS PIPELINE II=1
9 #pragma HLS UNROLL factor=2
10        int idx = i << 1; // i * 2
11        pooled[i] = (conv_out[idx] > conv_out[idx + 1]) ?
12            (fixed_t)conv_out[idx] :
13            (fixed_t)conv_out[idx + 1];
14    }
15 }
16
17 static void compute_fc(
18     fixed_t pooled[POOLED_SIZE],
19     acc_t fc_out[FC_OUT]
20 ) {
21 #pragma HLS INLINE off
22
23     // Precompute scaled biases
24     const acc_t bias_0 = (acc_t)FC_BIAS_QUANT[0] * FC_BIAS_SCALE;
25     const acc_t bias_1 = (acc_t)FC_BIAS_QUANT[1] * FC_BIAS_SCALE;
26
27     // Fully unrolled FC computation (only 2 outputs, 4 inputs)
28     acc_t acc_0 = bias_0;
29     acc_t acc_1 = bias_1;
30
31     FC COMPUTE: for (int i = 0; i < FC_IN; i++) {
32 #pragma HLS UNROLL
33         acc_t weight_0 = (acc_t)FC_WEIGHT_QUANT[0][i] * FC_WEIGHT_SCALE;
34         acc_t weight_1 = (acc_t)FC_WEIGHT_QUANT[1][i] * FC_WEIGHT_SCALE;
35
36         acc_0 += (acc_t)pooled[i] * weight_0;
37         acc_1 += (acc_t)pooled[i] * weight_1;
38     }
39
40     fc_out[0] = acc_0;
41     fc_out[1] = acc_1;
42 }
43
44 static void output_decision(
45     acc_t fc_out[FC_OUT],
46     hls::stream<axis_output> &predicted_bit
47 ) {
48 #pragma HLS INLINE off
49 }
```

```

93     axis_output out;
94     out.data = (ap_uint<1>)((fc_out[1] > fc_out[0]) ? 1 : 0);
95     out.last = 1;
96     predicted_bit.write(out);
97 }
98
99 void fsk_cnn_demod(
100     hls::stream<axis_data> &input_I,
101     hls::stream<axis_data> &input_Q,
102     hls::stream<axis_output> &predicted_bit
103 ) {
104 #pragma HLS INTERFACE axis port=input_I
105 #pragma HLS INTERFACE axis port=input_Q
106 #pragma HLS INTERFACE axis port=predicted_bit
107 #pragma HLS INTERFACE ap_ctrl_none port=return
108
109 // Enable task-level pipelining - allows new input while processing previous
110 #pragma HLS DATAFLOW
111
112 // Intermediate buffers
113 fixed_t I_buffer[SAMPLES_PER_SYMBOL];
114 fixed_t Q_buffer[SAMPLES_PER_SYMBOL];
115 acc_t conv_out[SAMPLES_PER_SYMBOL];
116 fixed_t pooled[POOLED_SIZE];
117 acc_t fc_out[FC_OUT];
118
119 #pragma HLS ARRAY_PARTITION variable=I_buffer complete
120 #pragma HLS ARRAY_PARTITION variable=Q_buffer complete
121 #pragma HLS ARRAY_PARTITION variable=conv_out complete
122 #pragma HLS ARRAY_PARTITION variable=pooled complete
123 #pragma HLS ARRAY_PARTITION variable=fc_out complete
124
125 // Dataflow pipeline stages
126 read_iq_parallel(input_I, input_Q, I_buffer, Q_buffer);
127 compute_convolution(I_buffer, Q_buffer, conv_out);
128 compute_pooling(conv_out, pooled);
129 compute_fc(pooled, fc_out);
130 output_decision(fc_out, predicted_bit);
131
132 }
```

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:

```
#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";
    }
}
```

```
std::cout << "-----\\n";

// 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
```

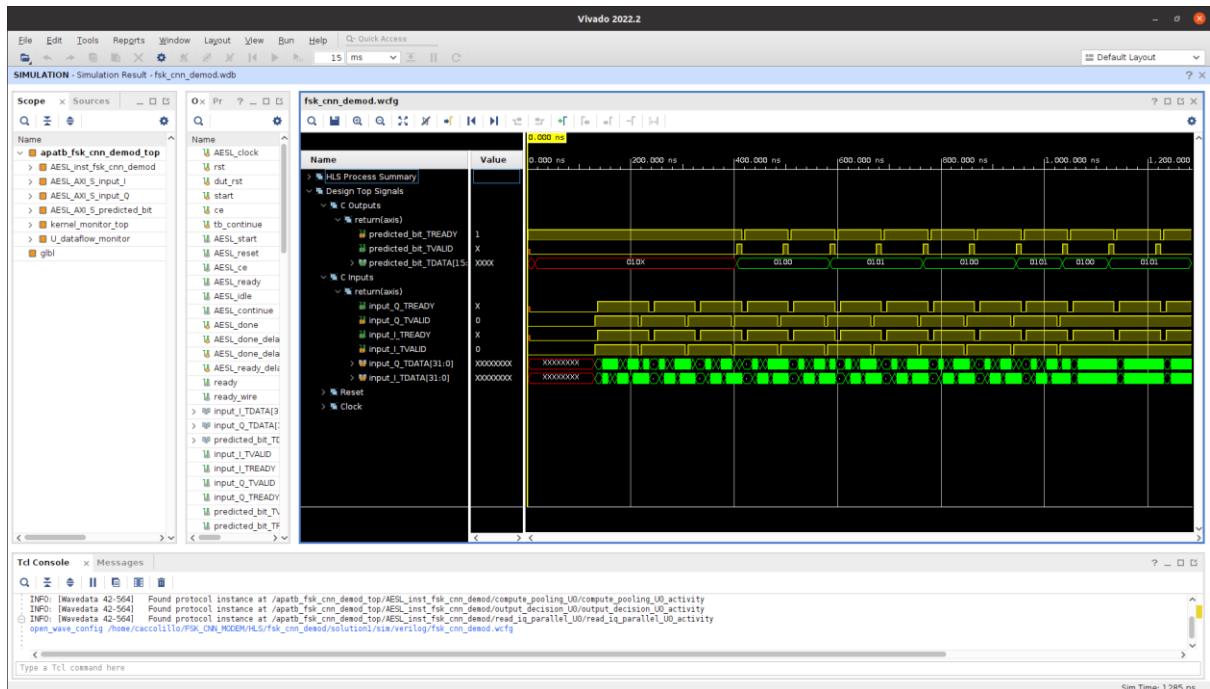
```

93     << " (tlast)" << predicted_output.last << ")\n";
94     std::cout << "Result: " << (is_correct ? "\x2022 PASS" : "\x2022 FAIL") << "\n\n";
95 }
96
97 // Print summary statistics
98 std::cout << "=====\\n";
99 std::cout << "TEST SUMMARY\\n";
100 std::cout << "=====\\n";
101 std::cout << "Total bits tested: " << total_bits << "\\n";
102 std::cout << "Correct predictions: " << correct_count << "\\n";
103 std::cout << "Incorrect predictions: " << (total_bits - correct_count) << "\\n";
104 std::cout << "Accuracy: " << (100.0 * correct_count / total_bits) << "%\\n";
105 std::cout << "=====\\n";
106
107 if (correct_count == total_bits) {
108     std::cout << "\x2022 ALL TESTS PASSED\\n";
109 } else {
110     std::cout << "\x2022 SOME TESTS FAILED\\n";
111 }
112 std::cout << "=====\\n";
113
114 return (correct_count == total_bits) ? 0 : 1;
115 }
116#endif
117

```

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:



```

Console ✘ Errors 🚨 Warnings ✉ Guidance Properties Man Pages Git Repositories Modules/Loops Dataflow
Vitis HLS Console
Test #9 - Bit = 1
-----
Generated I/O samples:
t=0: I=1.000000, O=0.000000
t=1: I=0.707107, O=0.707107
t=2: I=0.000000, O=1.000000
t=3: I=-0.707107, O=0.707107
t=4: I=-1.000000, O=-0.000000
t=5: I=-0.707107, O=-0.707107
t=6: I=0.000000, O=-1.000000
t=7: I=0.707107, O=-0.707107
Actual bit: 1
Predicted bit: 1 (last=1)
Result: ✓ PASS

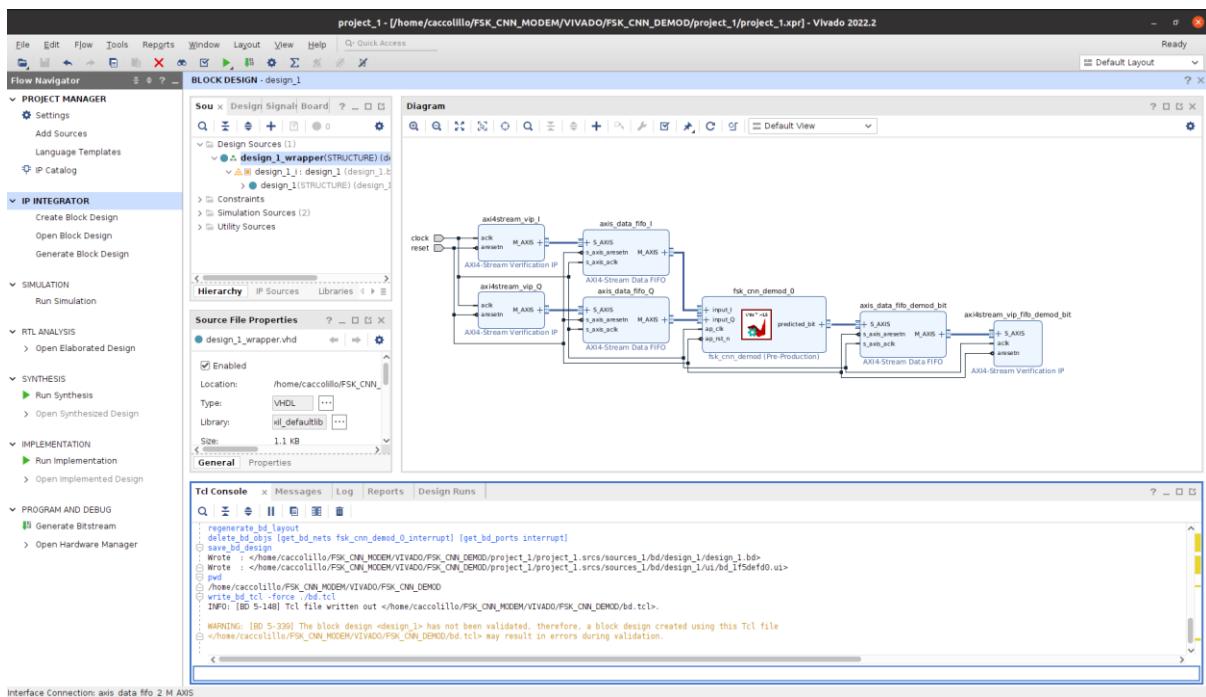
Test #10 - Bit = 1
-----
Generated I/O samples:
t=0: I=1.000000, O=0.000000
t=1: I=0.707107, O=0.707107
t=2: I=0.000000, O=1.000000
t=3: I=-0.707107, O=0.707107
t=4: I=-1.000000, O=-0.000000
t=5: I=-0.707107, O=-0.707107
t=6: I=0.000000, O=-1.000000
t=7: I=0.707107, O=-0.707107
Actual bit: 1
Predicted bit: 1 (last=1)
Result: ✓ PASS

=====
TEST SUMMARY
=====
Total bits tested: 10
Correct predictions: 10
Incorrect predictions: 0
Accuracy: 100.000000%
=====
✓ ALL TESTS PASSED!
=====
INFO [HLS SIM]: The maximum depth reached by any hls::stream() instance in the design is 8
INFO: [COSIM 212-1000] *** C/RTL co-simulation finished: PASS ***
INFO: [HLS 200-111] Finished Command cosim design CPU user time: 24.78 seconds. CPU system time: 4.03 seconds. Elapsed time: 27.42 seconds; current allocated memory: 13.148 MB.
INFO: [HLS 200-112] Total CPU user time: 26.54 seconds. Total CPU system time: 4.62 seconds. Total elapsed time: 39.81 seconds; peak allocated memory: 770.480 MB.
Finished C/RTL cosimulation.

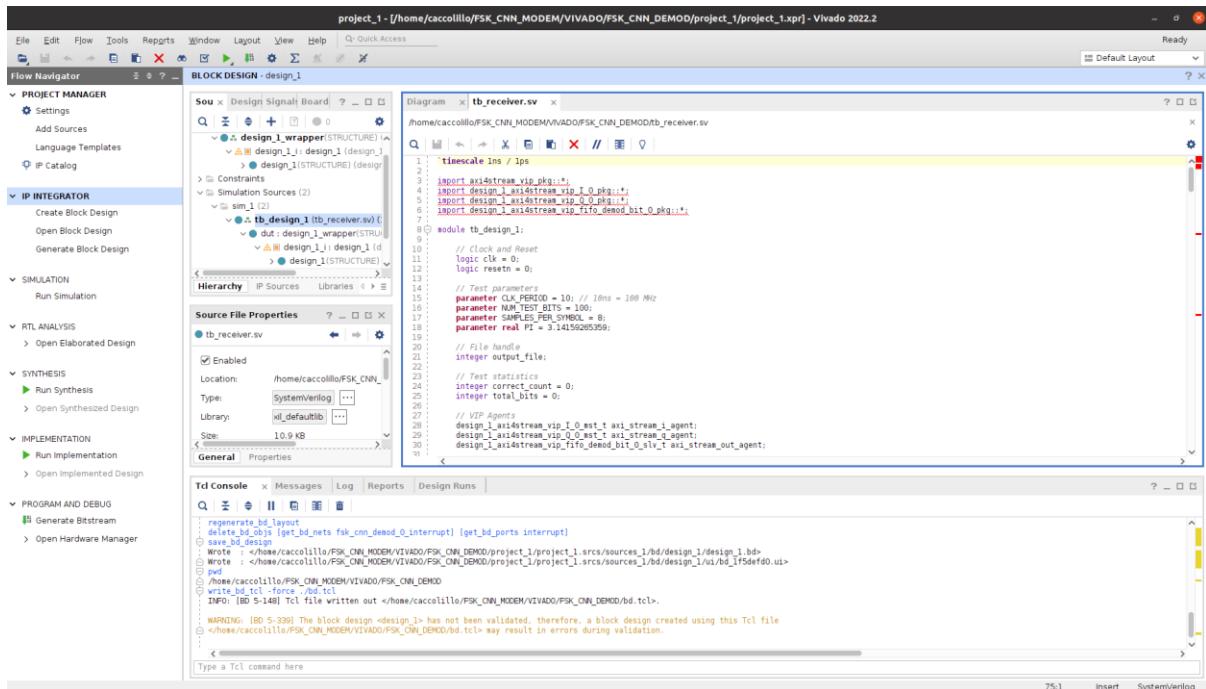
```

HLS CNN VIVADO TESTBENCH

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

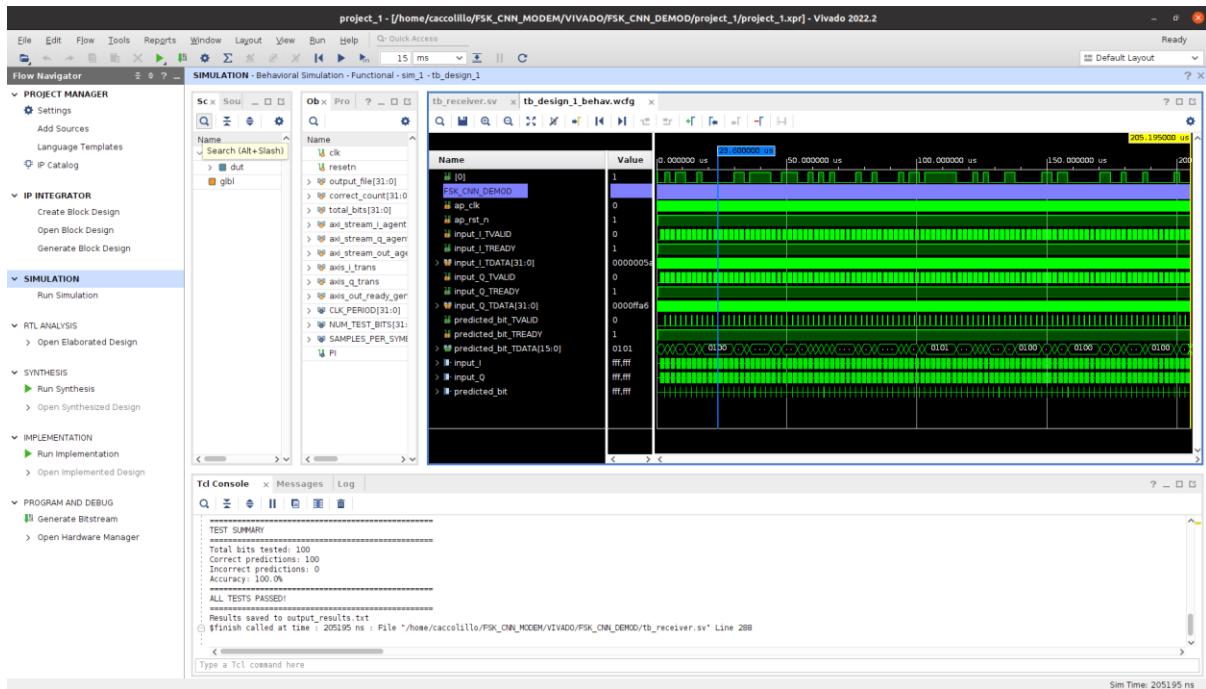


A system verilog testbench mimics the Vitis HLS one:



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 following 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>

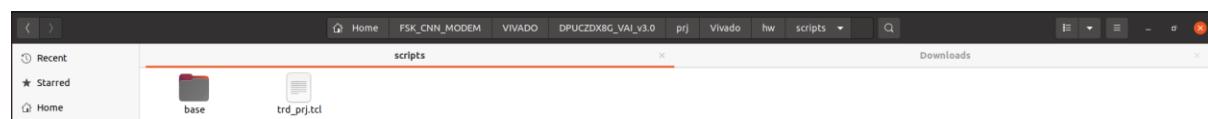
A reference Vivado design is available for download:

The screenshot shows a table listing the DPUCZDX8G Edge IP. The table has columns for IP Name, Supported Platforms, Description, Reference Design, Product Guide, Read Me, and IP-only Download. The DPUCZDX8G entry is highlighted.

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:



```

trd_prj.tcl
trd_bd.tcl

1 # /*
2 # * Copyright 2019 Xilinx Inc.
3 # *
4 # * Licensed under the Apache License, Version 2.0 (the "License");
5 # * you may not use this file except in compliance with the License.
6 # * You may obtain a copy of the License at
7 # * http://www.apache.org/licenses/LICENSE-2.0
8 # *
9 # * Unless required by applicable law or agreed to in writing, software
10 # * distributed under the License is distributed on an "AS IS" BASIS,
11 # * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # * See the License for the specific language governing permissions and
13 # * limitations under the License.
14 # */
15 #
16 #
17 # ****
18 # set global dict_prj
19 # ****
20 set dict_prj {}
21 #
22 # ****
23 # set project
24 # ****
25 dict set dict_prj dict_sys proj_name [xcu102]
26 dict set dict_prj dict_sys proj_part [xcu09eg-ffvb1156-2-i]
27 dict set dict_prj dict_sys proj_board [xcu102v2x00]
28 #
29 # ****
30 # set bd
31 # for bd_doc: None for global, Hierarchical for doc per IP
32 # ****
33 dict set dict_prj dict_sys bd_name top
34 dict set dict_prj dict_sys bd_doc None
35 #
36 # ****
37 # set param
38 # ****
39 dict set dict_prj dict_param DPU_CLK_MHZ [325]
40 dict set dict_prj dict_param REC_CLK_MHZ [100]
41 #
42 #The following parameters correspond to Arch Tab of the IP GUI
43 dict set dict_prj dict_param DPU_NUM [3]
44 dict set dict_prj dict_param DPU_ARCH [4096]
45 dict set dict_prj dict_param DPU_RAM_USAGE [low]
46 dict set dict_prj dict_param DPU_CLK_GATE_ENA [1]
47 dict set dict_prj dict_param DPU_DSP48_ENA [1]
48 dict set dict_prj dict_param DPU_CONV_RELU_TYPE [3]
49 dict set dict_prj dict_param DPU_ALU_PARALLEL_USER [4]
50 dict set dict_prj dict_param DPU_ALU_LEAKYRELU [0]
51 dict set dict_prj dict_param DPU_SF_NUM [1]
52
53 #The following parameters correspond to Advanced Tab of the IP GUI
54 dict set dict_prj dict_param DPU_NUM [1]
55 dict set dict_prj dict_param DPU_CLK_GATE_ENA [1]
56 dict set dict_prj dict_param DPU_DSP48_USAGE [1]
57 dict set dict_prj dict_param DPU_DSP48_ENA [1]
58 dict set dict_prj dict_param DPU_DSRB_USAGE [1]
59 dict set dict_prj dict_param DPU_UNARY_PER_DPU [0]
60
61 # ****
62 # ****
63 dict set dict_prj dict_sys work_dtr [file normalize [linfo script]]
64 dict set dict_prj dict_sys work_dtr [dict get select_prj dict_sys work_dtr]base
65 source -notrace
66
67 # run Flow
68
69 lib_12m
70
71

```

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

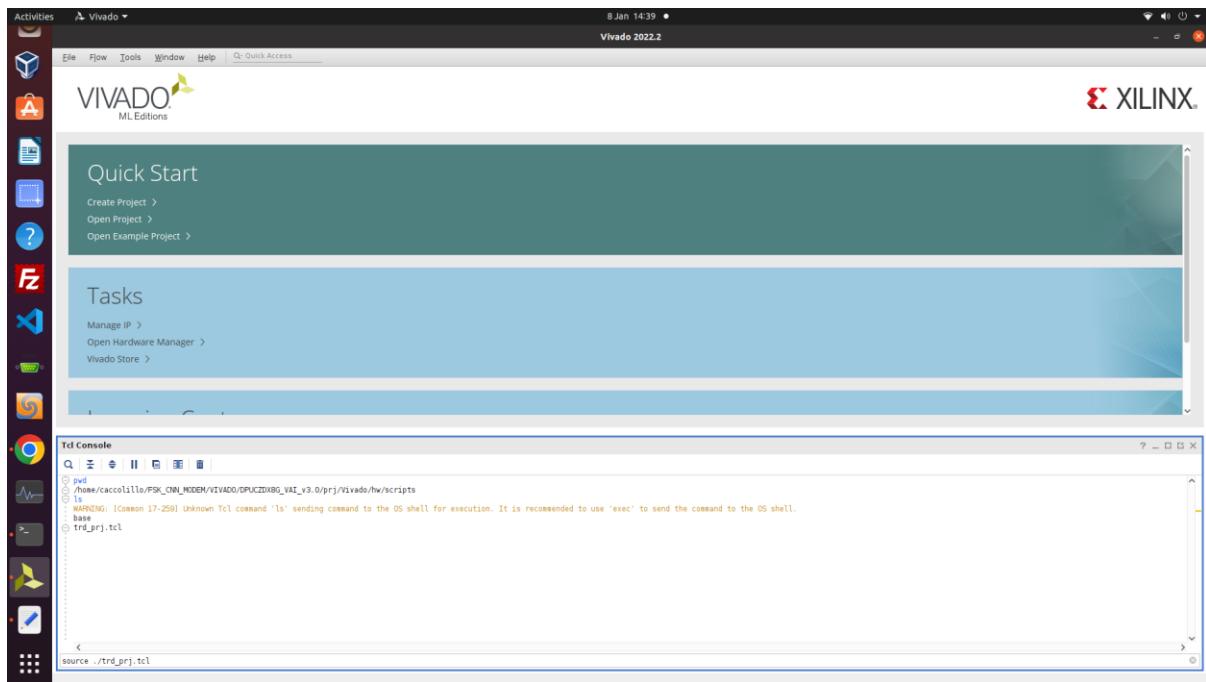
```

trd_prj.tcl
trd_bd.tcl

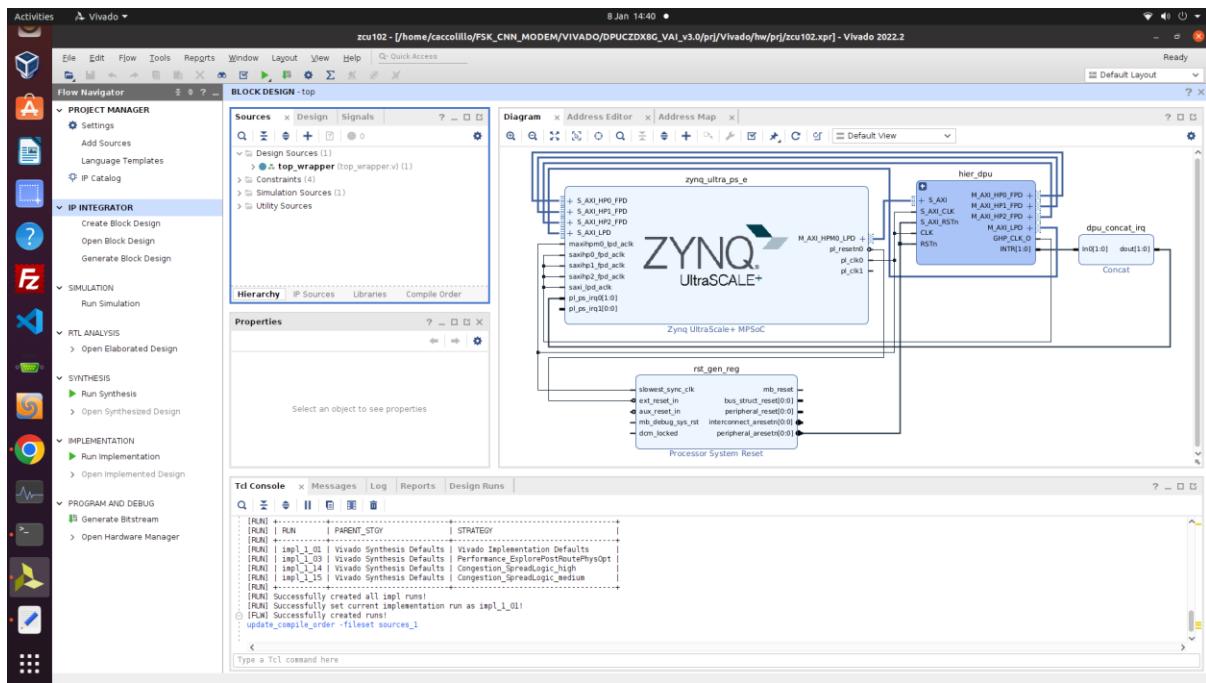
10
11 #
12 # ****
13 # set global dict_prj
14 # ****
15 set dict_prj {}
16 #
17 # ****
18 # set project
19 # ****
20 dict set dict_prj dict_sys proj_name [xcu102]
21 dict set dict_prj dict_sys proj_part [xcu09eg-ffvb1156-2-i]
22 dict set dict_prj dict_sys proj_board [xcu102v2x00]
23 #
24 # ****
25 # set param
26 # ****
27 dict set dict_prj dict_param DPU_CLK_MHZ [325]
28 dict set dict_prj dict_param REC_CLK_MHZ [100]
29 #
30 # ****
31 # for bd_doc: None for global, Hierarchical for doc per IP
32 # ****
33 dict set dict_prj dict_sys bd_name top
34 dict set dict_prj dict_sys bd_doc None
35 #
36 # ****
37 # set param
38 # ****
39 dict set dict_prj dict_param DPU_CLK_MHZ [325]
40 dict set dict_prj dict_param REC_CLK_MHZ [100]
41 #
42 #The following parameters correspond to Arch Tab of the IP GUI
43 dict set dict_prj dict_param DPU_NUM [1]
44 dict set dict_prj dict_param DPU_ARCH [2304]
45 dict set dict_prj dict_param DPU_RAM_USAGE [low]
46 dict set dict_prj dict_param DPU_CLK_GATE_ENA [1]
47 dict set dict_prj dict_param DPU_DSP48_ENA [1]
48 dict set dict_prj dict_param DPU_CONV_RELU_TYPE [3]
49 dict set dict_prj dict_param DPU_ALU_PARALLEL_USER [4]
50 dict set dict_prj dict_param DPU_ALU_LEAKYRELU [0]
51 dict set dict_prj dict_param DPU_SF_NUM [1]
52
53 #The following parameters correspond to Advanced Tab of the IP GUI
54 dict set dict_prj dict_param DPU_NUM [1]
55 dict set dict_prj dict_param DPU_CLK_GATE_ENA [1]
56 dict set dict_prj dict_param DPU_DSP48_USAGE [1]
57 dict set dict_prj dict_param DPU_DSP48_ENA [1]
58 dict set dict_prj dict_param DPU_DSRB_USAGE [1]
59 dict set dict_prj dict_param DPU_UNARY_PER_DPU [0]
60
61 # ****
62 # ****
63 dict set dict_prj dict_sys work_dtr [file normalize [linfo script]]
64 dict set dict_prj dict_sys work_dtr [dict get select_prj dict_sys work_dtr]base
65 source -notrace
66
67 # run Flow
68
69 lib_12m
70
71

```

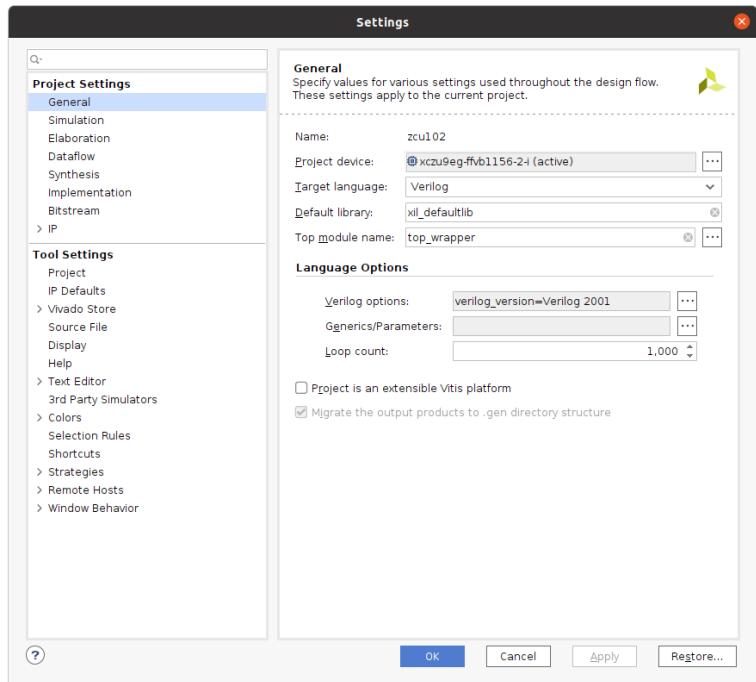
Then we run the script in Vivado getting the design, but for the zcu102 board.



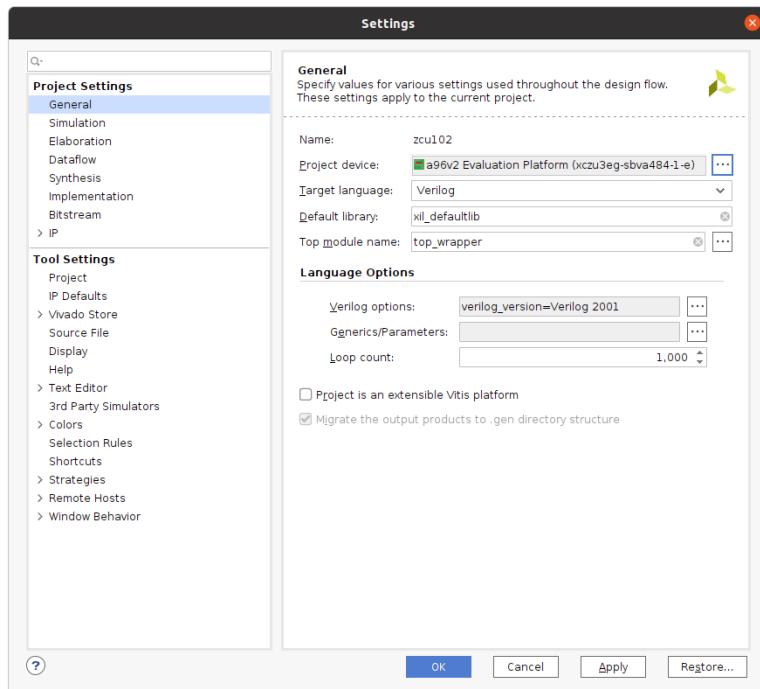
Once executed it:



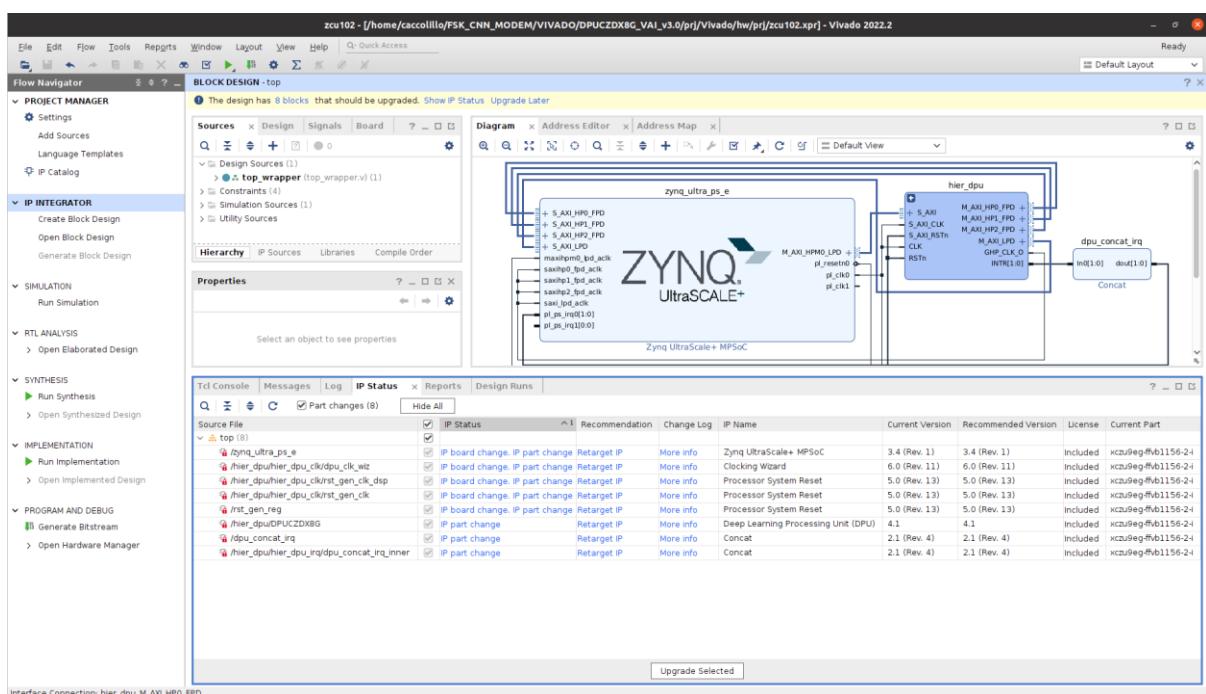
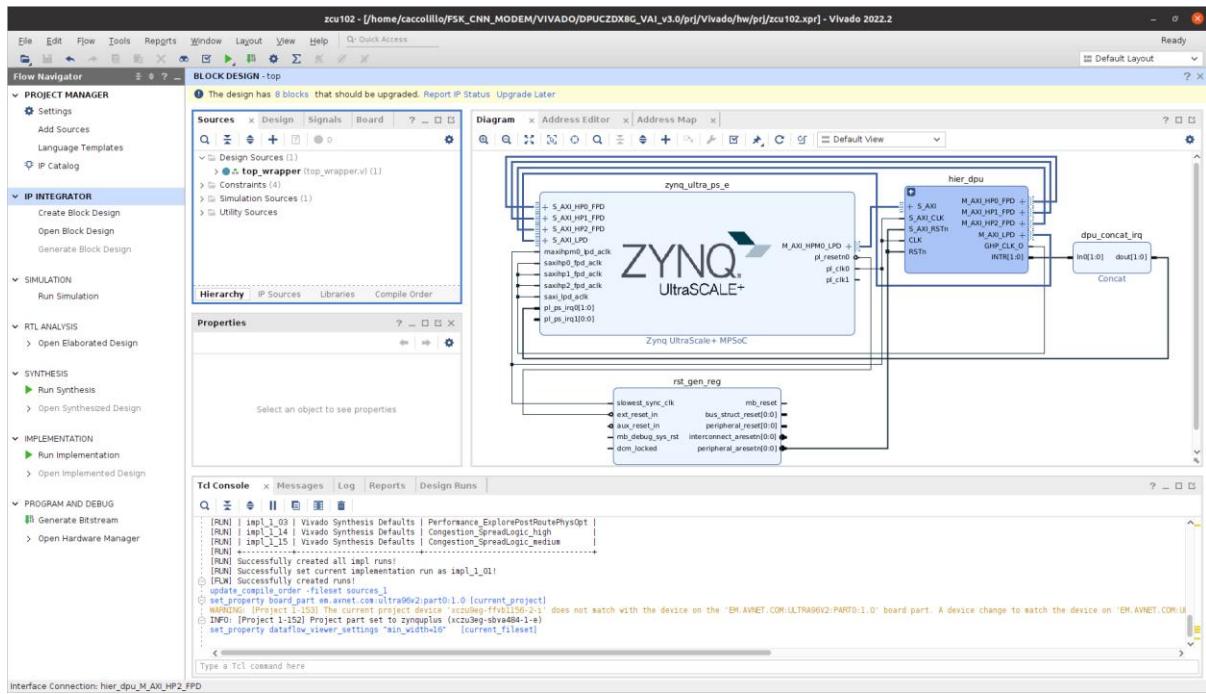
we can change the board in use:



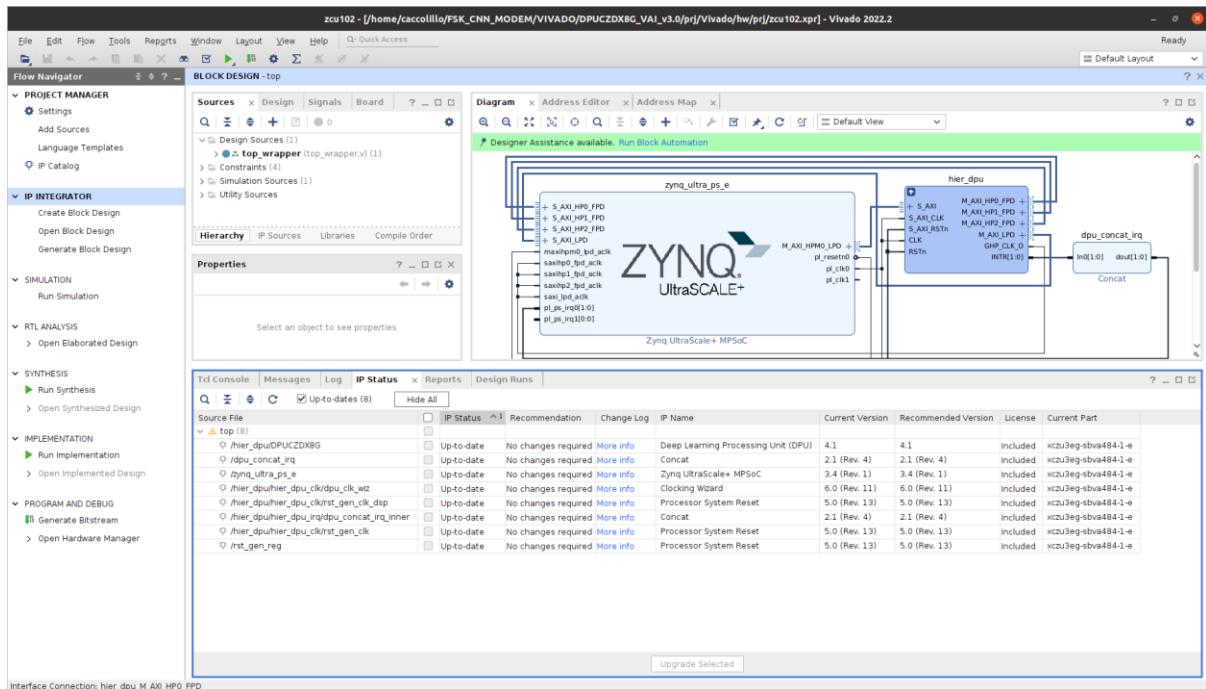
to the Ultra96 v2:



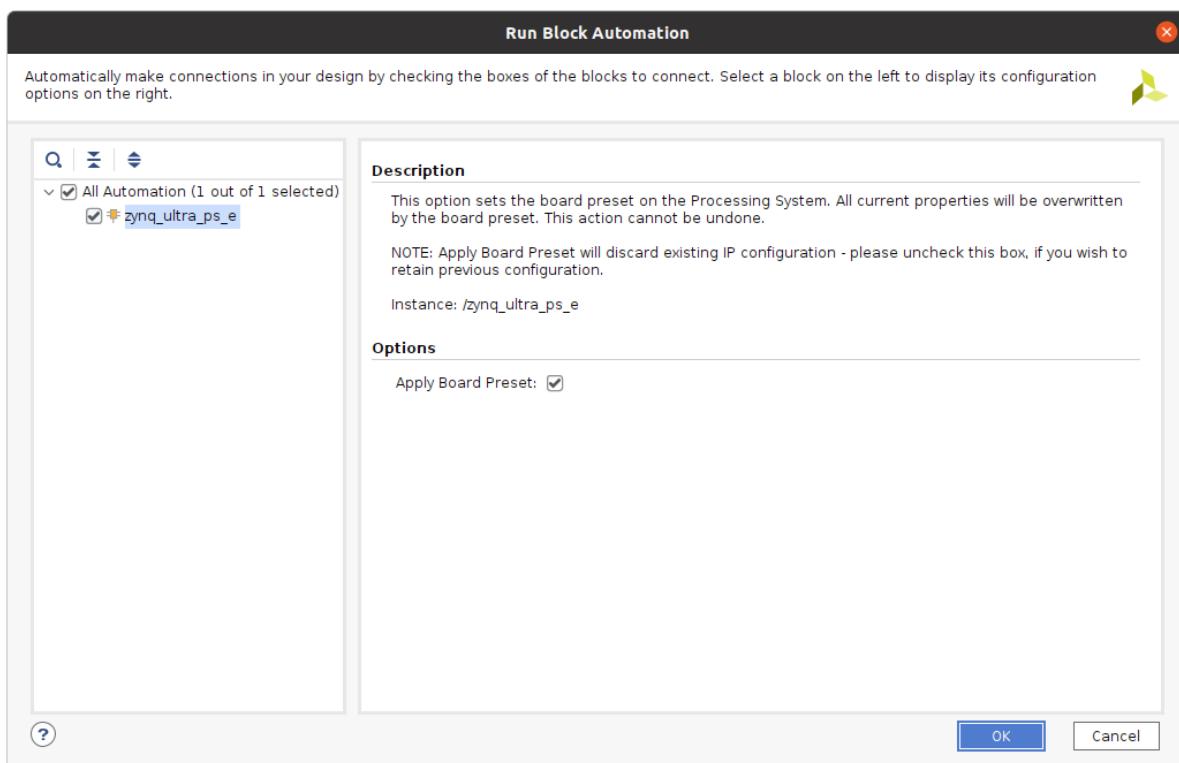
run the report IP summary:



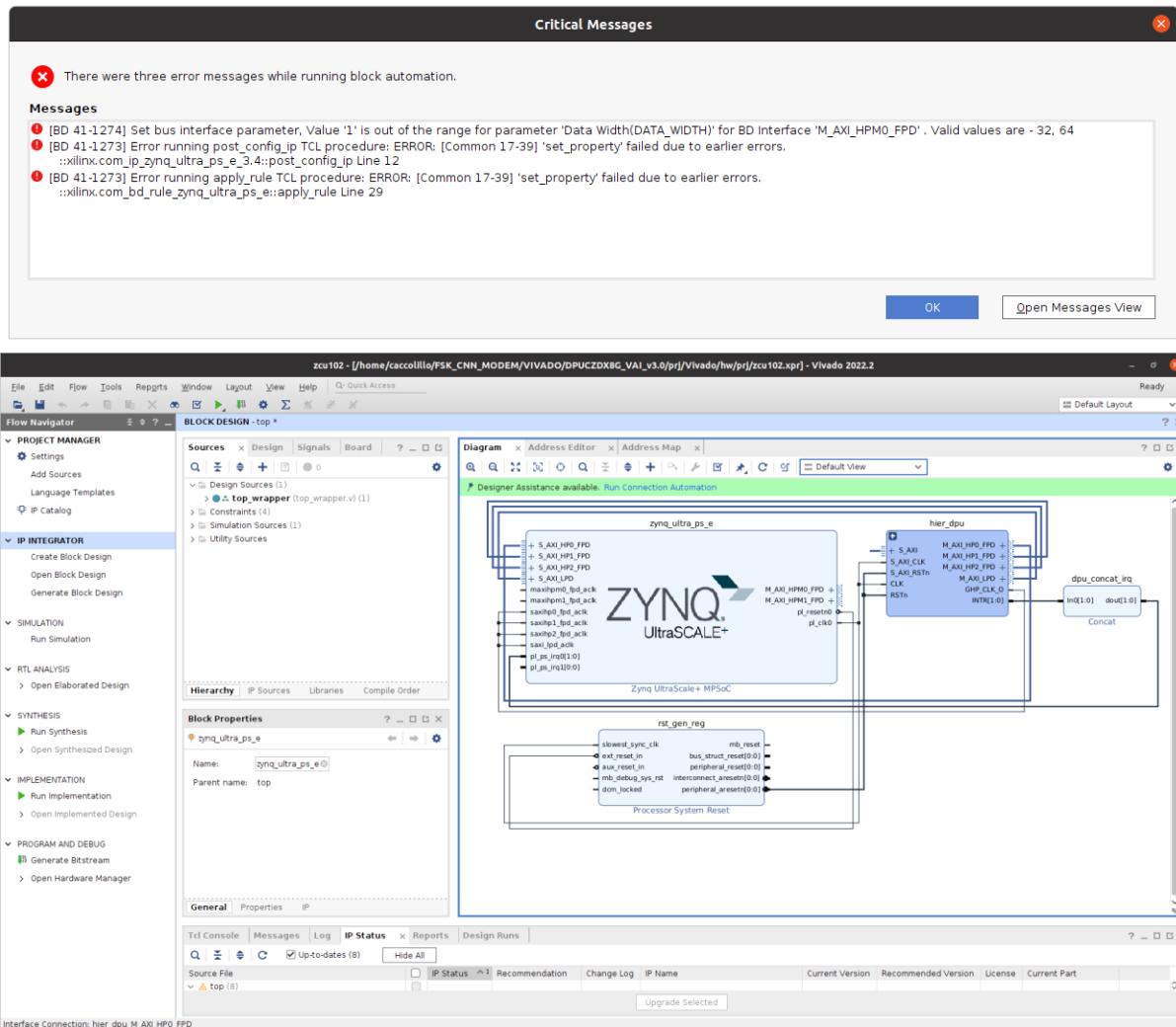
upgrade the IPs:



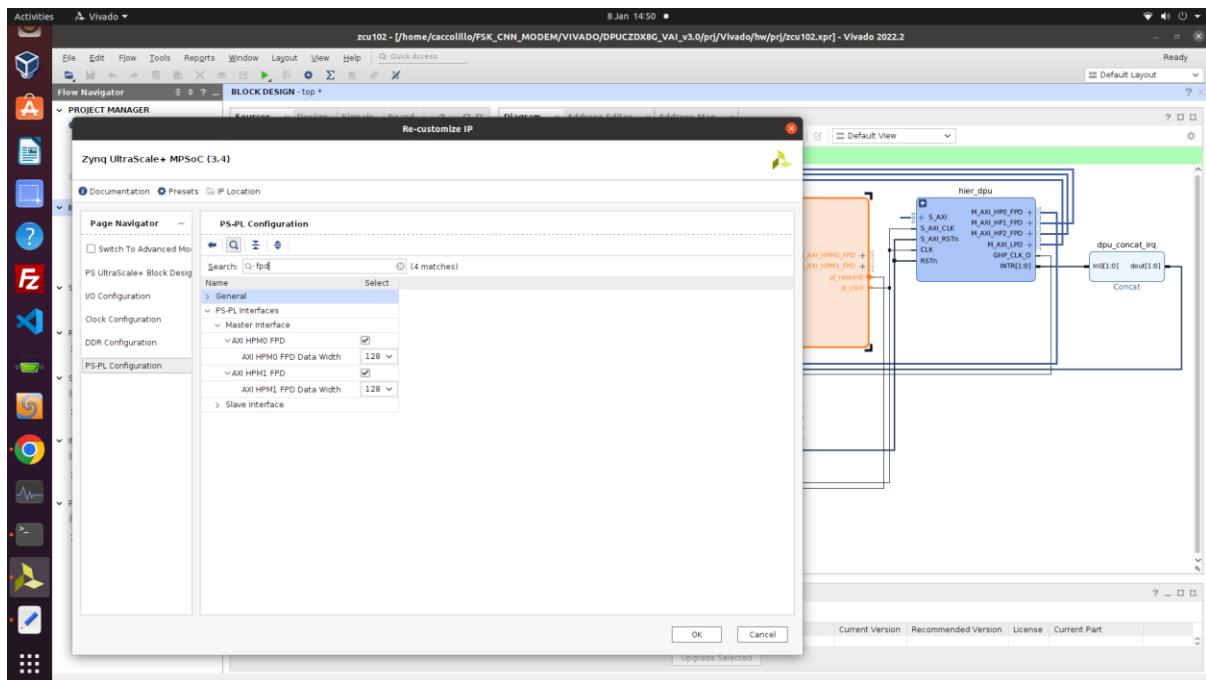
Then run block automation so to apply Ultra96 board presets:



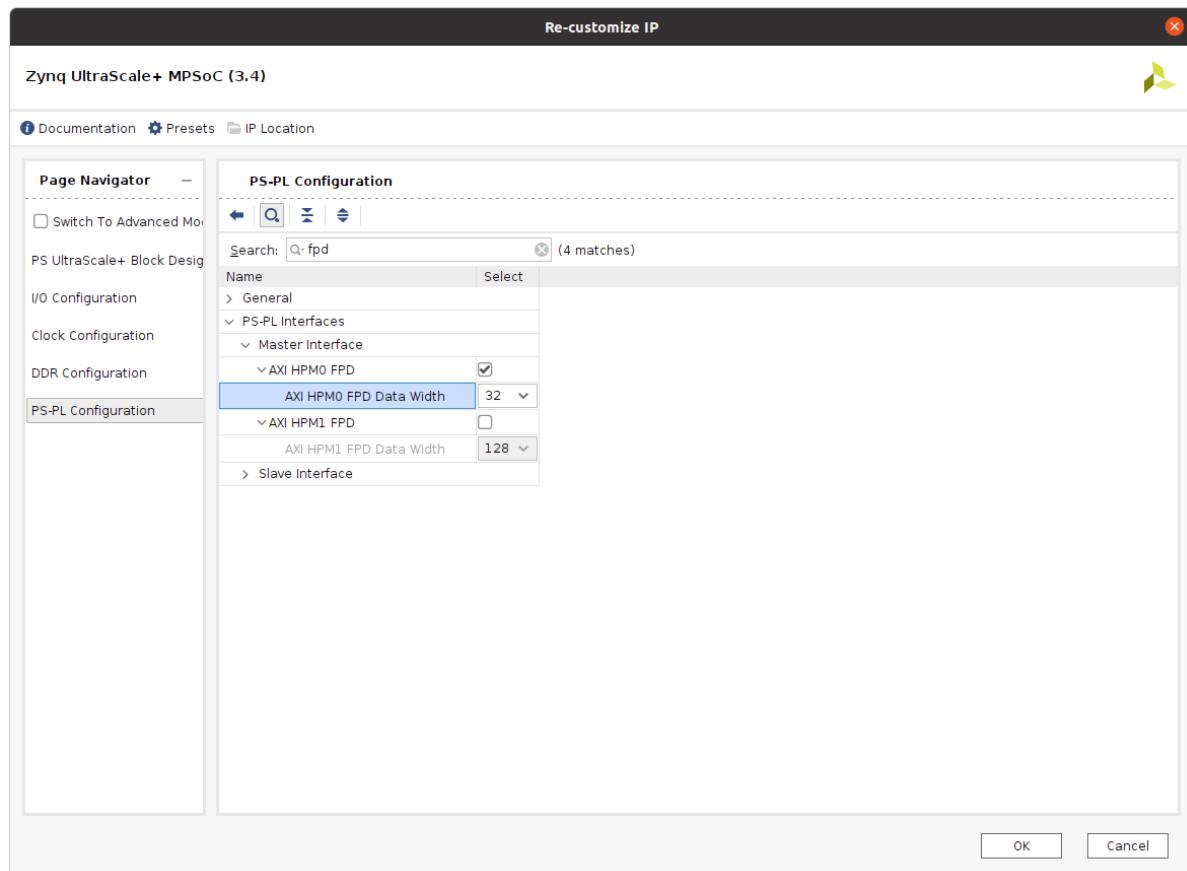
It will error, modify the block design and disconnect the DPU:



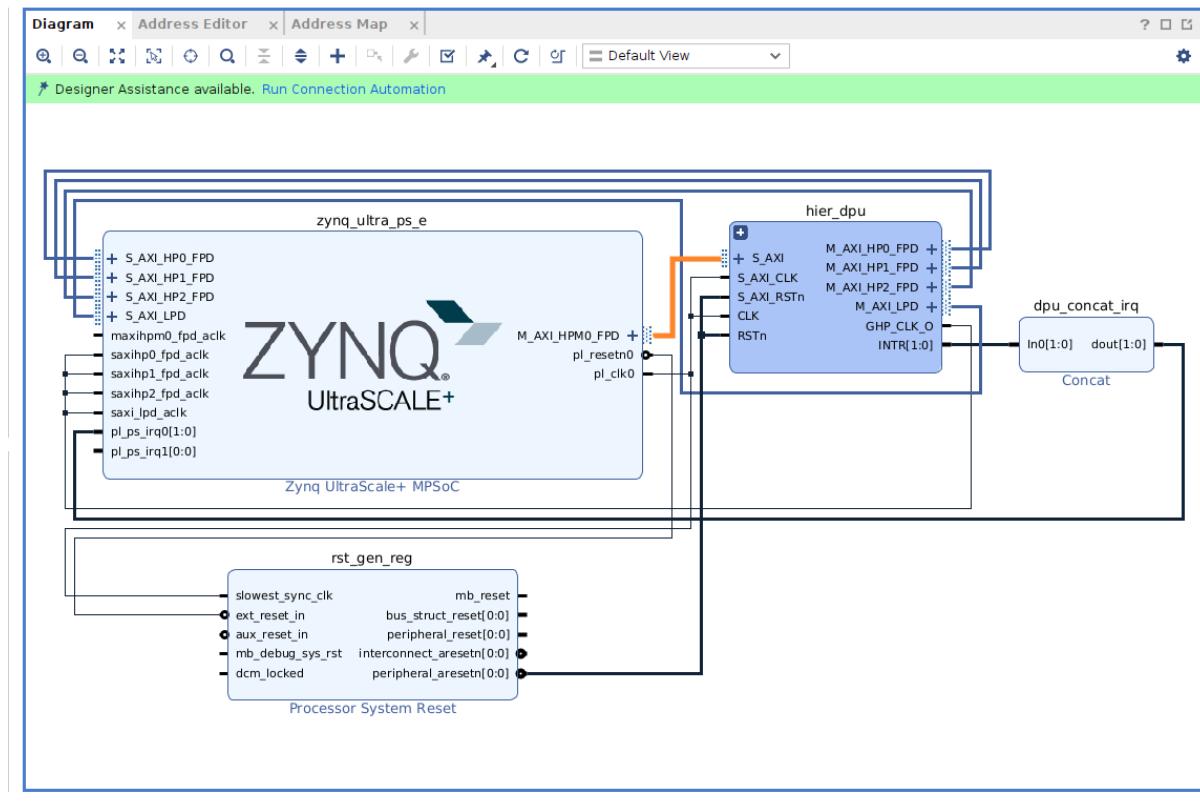
then we have to open the zynq_ultra_ps_e block and change configuration parameters for the master AXI ports:



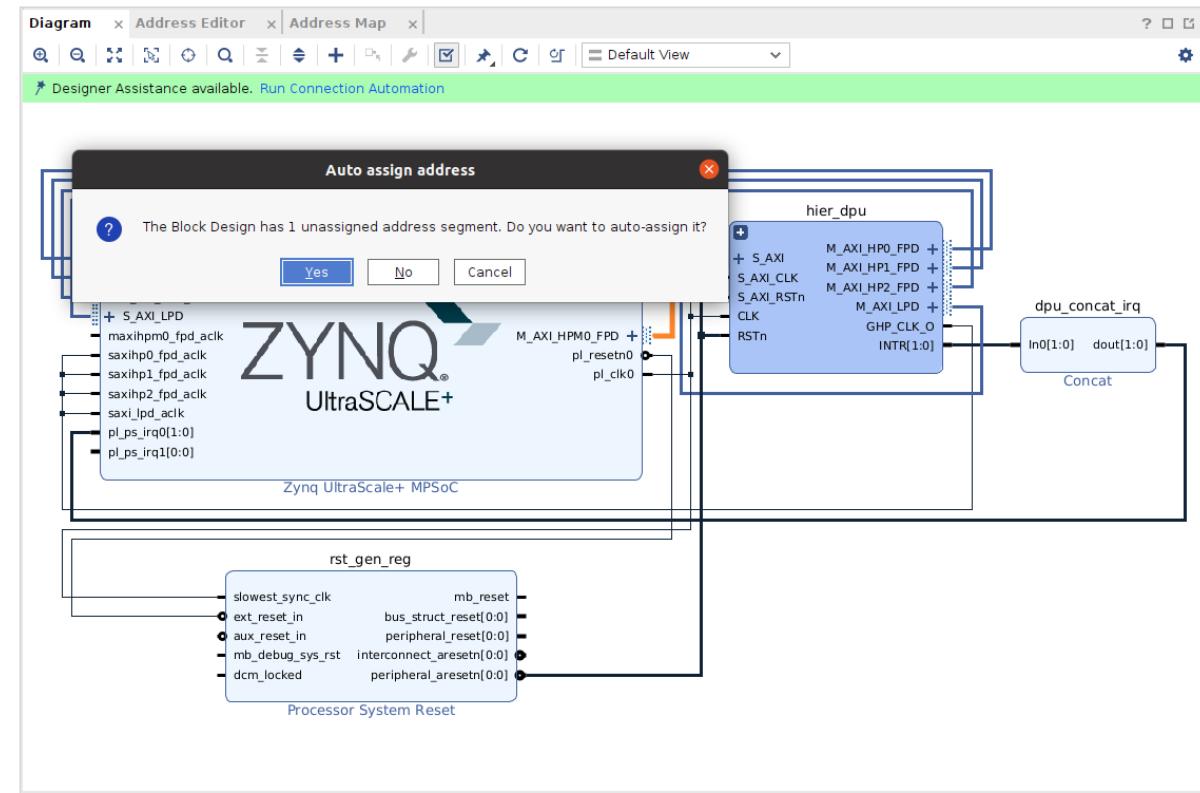
Disconnect the HPM1 FPD port, reconfigure the other one to use 32 bits:



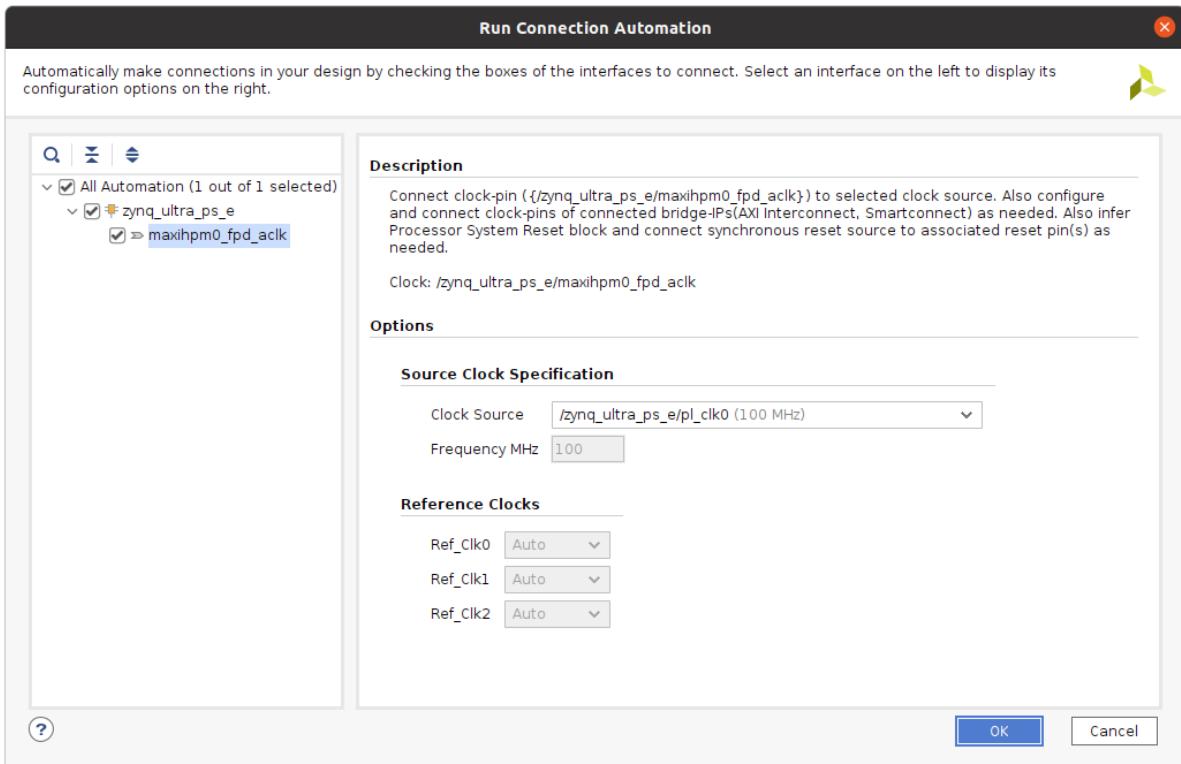
Restore the connection to the DPU:



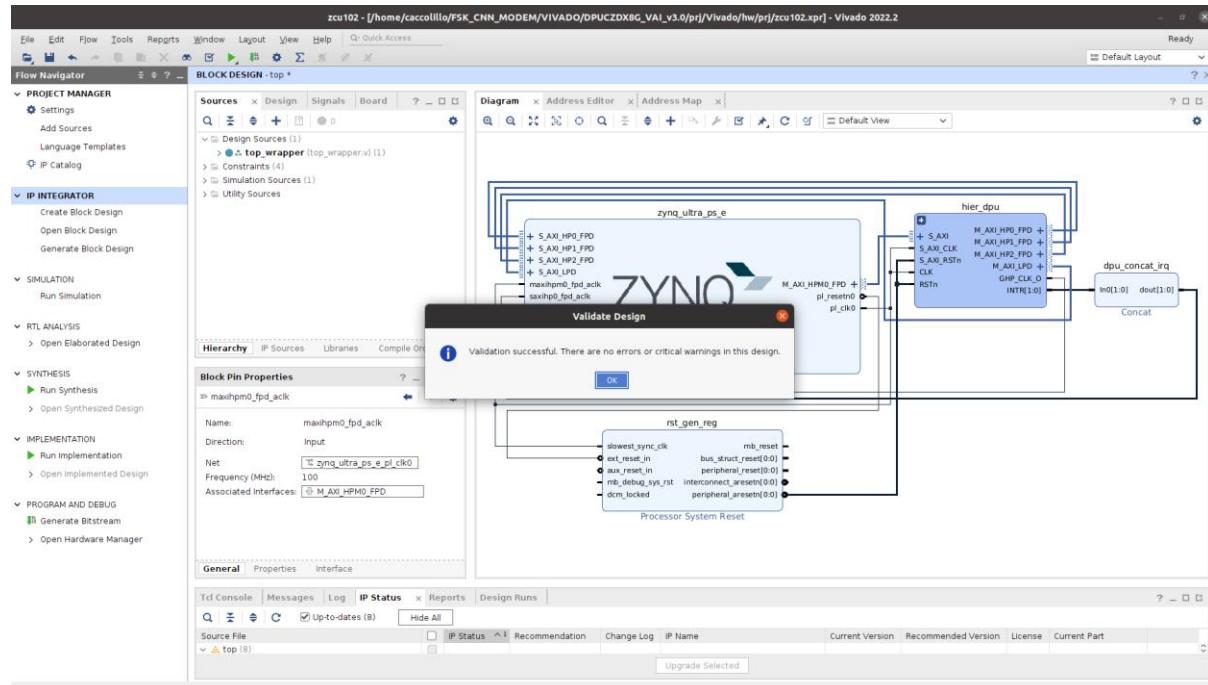
validate the block design:



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:

```
File Edit Log Configuration Controlsignals View Help  
U96V2-sbc-2022-2:/home/petalinux# xmutil listapps  
Accelerator          Accel_type           Base      Base_type    #slots(PL+AIE) Active_slot  
avnet-u96v2-dualcam-dpu XRT_FLAT        avnet-u96v2-dualcam-dpu XRT_FLAT      (0+0)       -1  
avnet-u96v2-base       XRT_FLAT        avnet-u96v2-base       XRT_FLAT      (0+0)       -1  
avnet-u96v2-benchmark  XRT_FLAT        avnet-u96v2-benchmark XRT_FLAT      (0+0)       0,  
avnet-u96v2-dualcam   XRT_FLAT        avnet-u96v2-dualcam  XRT_FLAT      (0+0)       -1  
u96v2-sbc-2022-2:/home/petalinux# xmutil query  
usage: xmutil [-h]  
  {bootfw_status,bootfw_update,getpkgs,listapps,loadapp,unloadapp,xlnx_platformstats,ddr qos,axiqos,pwrctrl,desktop_disable,desktop_enable,dp_unbind,dp_bind}  
xmutil: error: argument cmd: invalid choice: 'query' (choose from 'boardid', 'bootfw_status', 'bootfw_update', 'getpkgs', 'listapps', 'loadapp', 'unloadapp', 'xlnx_platformstats', 'ddr qos', 'axiqos', 'pwrctrl', 'desktop_disable', 'desktop_enable', 'dp_unbind', 'dp_bind')  
u96v2-sbc-2022-2:/home/petalinux# xdputil query  
{  
  "DPU IP Spec":{  
    "DPU Core Count":1,  
    "IP version": "V4.1.0",  
    "Generation timestamp": "2023-02-21 21:30:00",  
    "git commit id": "7d32c41",  
    "git commit time": "202302212121",  
    "regmap": "l1ot version"  
  },  
  "VAI Version":{  
    "libvart-runner.so": "Xilinx vart-runner Version: 3.0.0-c5d2bd43d951c174185d728b8e5bcd3869e0b39 2023-04-10-20:06:16 ",  
    "libvitis_ai_library-dpu.task.so": "Xilinx vitis ai library dpu task Version: 3.0.0-c5d2bd43d951c174185d728b8e5bcd3869e0b39 2023-01-13 06:58:30 [UTC] ",  
    "libxir.so": "Xilinx xir Version: xir-c5d2bd43d951c174185d728b8e5bcd3869e0b39 2023-04-03-14:21:20",  
    "target_factory": "target-factory_3.0.0_c5d2bd43d951c174185d728b8e5bcd3869e0b39"  
  },  
  "kernels":{  
    {  
      "DPU Arch": "DPUCZDX8G_ISA1_B2304_0101000016010405",  
      "DPU Frequency (MHz)": 200,  
      "Load type": "Parallel",  
      "Load Parallel": 2,  
      "Load Augmentation": "enable",  
      "Load min/max mean": "disable",  
      "Save Parallel": 2,  
      "XRT Frequency (MHz)": 200,  
      "cu_addr": "0xb0000000",  
      "cu_handle": "0xaac17884dd0",  
      "cu_idx": 0,  
      "cu_name": "",  
      "CU name": "DPUCZDX8G:DPUCZDX8G_1",  
      "device id": 0,  
      "fingerprint": "0x101000016010405",  
      "name": "DPU Core 0"  
    }  
  }  
}
```

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:

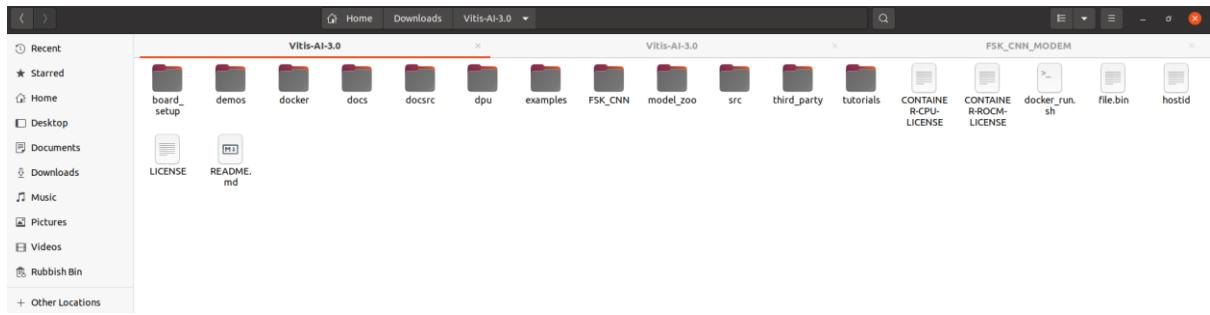
```
./docker_run.sh xilinx/vitis-ai-pytorch-cpu:ubuntu2004-3.0.0.106
```

```
caccoollo@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: d4ec20f
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 created a folder with the following quantization script:

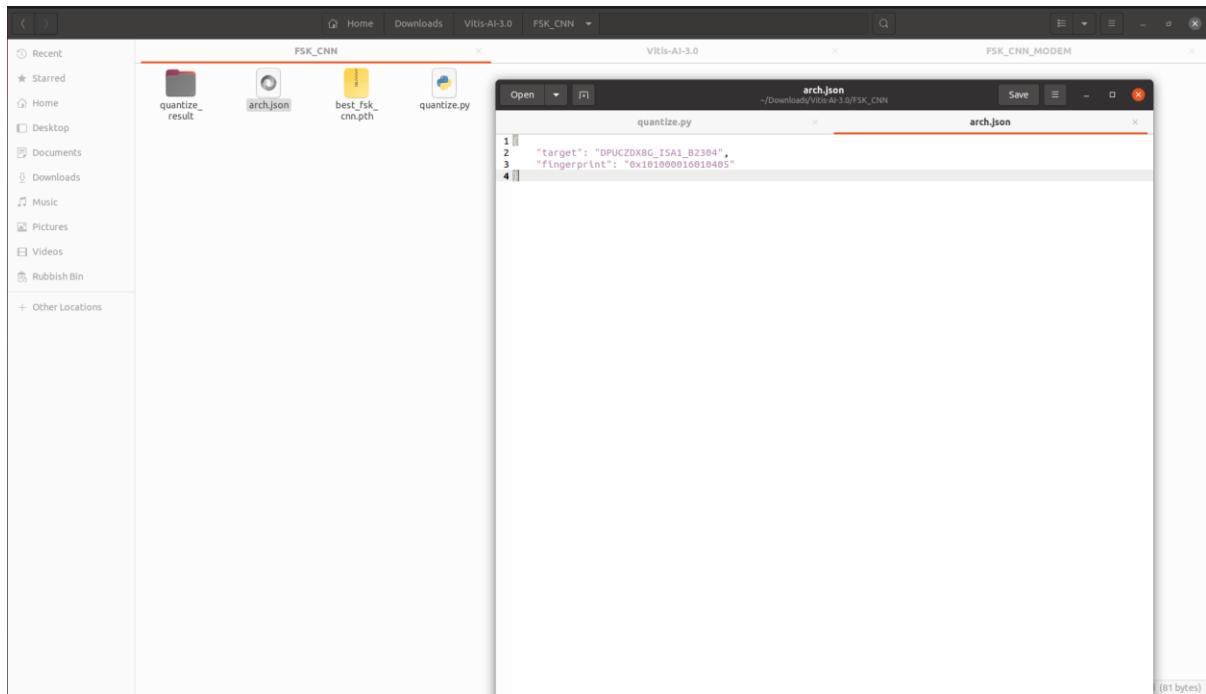
```

quantize.py
~/Downloads/Vitis-AI-3.0/FSK_CNN

1 import torch
2 import torch.nn as nn
3 import numpy as np
4 import os
5 from pytorch_mnct.apis import torch_quantizer
6
7 # ... 1. Define Model Architecture ...
8 class CNNFSK(nn.Module):
9     def __init__(self):
10         super(CNNFSK, self).__init__()
11         self.conv = nn.Conv2d(in_channels=2, out_channels=1, kernel_size=1, stride=1, bias=True)
12         self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)
13         self.fc = nn.Linear(4, 2)
14
15     def forward(self, x):
16         x = self.conv(x)
17         x = self.maxpool(x)
18         x = x.view(x.size(0), -1)
19         x = self.fc(x)
20
21         return x
22
23 # ... 2. Calibration Data Generator ...
24 def get_calibration_samples(num_samples=32):
25     samples = []
26     eb_no_db = 10
27     f_sampling = 8e6
28     f_deviation = 1e6
29     samples_per_symbol = 8
30
31     for _ in range(num_samples):
32         bit = np.random.randint(0, 2)
33         phase_shift = -np.pi * (f_deviation if bit == 1 else -f_deviation) / f_sampling
34         I = np.cos(phase_shift * np.arange(samples_per_symbol))
35         Q = np.sin(phase_shift * np.arange(samples_per_symbol))
36
37         noise_std = np.sqrt(1 / (2 * (10**(-eb_no_db/10))))
38         I += np.random.normal(0, noise_std, samples_per_symbol)
39         Q += np.random.normal(0, noise_std, samples_per_symbol)
40
41         symbol = np.stack([I, Q], axis=0)
42         samples.append(symbol)
43
44     return torch.tensor(np.array(samples), dtype=torch.float32)
45
46 def run_quantization(model_path, quant_mode='calib'):
47     device = torch.device("cpu")
48     model = CNNFSK().to(device)
49     model.load_state_dict(torch.load(model_path, map_location=device))
50     model.eval()
51
52     # CRITICAL FIX:
53     # Use batch size 32 for 'calib' to get better statistics.
54     # Use batch size 1 for 'test' to satisfy the XModel export requirement.
55     batch_size = 32 if quant_mode == 'calib' else 1
56
57     inputs = get_calibration_samples(num_samples=batch_size)
58
59     # Define dummy input shape for the quantizer (Batch size 1)
60     dummy_input = torch.randn(1, 2, 8)
61
62     # 3. Create Quantizer
63     quantizer = torch_quantizer(quant_mode, model, (dummy_input), device=device)
64     quant_model = quantizer.quant_model
65
66     # 4. Forward pass to trace the model
67     with torch.no_grad():
68         _ = quant_model(inputs)
69
70     # 5. Handle output
71     if quant_mode == 'calib':
72         quantizer.export_quant_config()
73         print("\n[SUCCESS] Calibration finished. Config saved to ./quantize_result/")
74     else:
75         # Export the final xmodel for the GPU
76         quantizer.export_xmodel(deploy_check=False, output_dir="quantize_result")
77         print("\n[SUCCESS] Quantized xmodel saved to ./quantize_result/CNNFSK_int.xmodel")
78
79     if __name__ == "__main__":
80         MODEL_FILE = "best_fsk_cnn.pth"
81
82     if not os.path.exists(MODEL_FILE):
83         print(f"Error: {MODEL_FILE} not found!")
84     else:
85         # Step 1: Calibration (Gathers statistics)
86         print("*"*50)
87         print("STEP 1: RUNNING CALIBRATION")
88         print("*"*50)
89         run_quantization(MODEL_FILE, quant_mode='calib')
90
91         # Step 2: Export (Generates hardware-compatible files)
92         print("*"*50)
93         print("STEP 2: EXPORTING XMODEL")
94         print("*"*50)
95         run_quantization(MODEL_FILE, quant_mode='test')

```

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-al-user@caccoillo-OMEN-2SL-Desktop-GT12-xxxx:/workspace/FSK_CNN
TypeError: __init__() got an unexpected keyword argument 'model'
(vitis-al-pytorch) vitis-al-user@caccoillo-OMEN-2SL-Desktop-GT12-xxxx:/workspace/FSK_CNN$ python3 ./quantize.py
No CUDA runtime is found, using CUDA_HOME='/usr/local/cuda'

[VAIO_NOTE]: Loading NNDCT kernels...
--- STEP 1: CALIBRATION ---
[VAIO_NOTE]: OS and CPU information:
    system --- Linux
        node --- caccoillo-OMEN-2SL-Desktop-GT12-xxxx
        release --- 5.15.0-139-generic
        version --- #1~22020604.1-Ubuntu SMP Wed Apr 16 08:29:56 UTC 2025
        machine --- x86_64
        processor --- x86_64

[VAIO_NOTE]: Tools version information:
    GCC --- GCC 7.5.0
    Python --- Python 3.8.12
    pytorch --- 1.13.1
    val_q_pytorch --- 3.5.8+e0df3f1+torch1.13.1

[VAIO_NOTE]: Quant config file is empty, use default quant configuration
[VAIO_NOTE]: Quantization calibration process start up...
[VAIO_NOTE]: ==Quant Module ts in 'cpu'.
[VAIO_NOTE]: ==Parsign CNNFSK...
[VAIO_NOTE]: Start to trace and freeze model...
[VAIO_NOTE]: The input model nndct_st_CNNFSK_ed is torch.nn.Module.
[VAIO_NOTE]: Finish tracing.
[VAIO_NOTE]: Processing ops... | 7/7 [00:00<00:00, 2534.98it/s, OpInfo: name = return_0, type = Return]
[VAIO_NOTE]: ==Doing weights equalization...
[VAIO_NOTE]: ==Quantizable module is generated.(quantize_result/CNNFSK.py)

[VAIO_NOTE]: ==Get module with quantization
/opt/vitis_ai/conda/envs/vitis-al-pytorch/lib/python3.8/site-packages/pytorch_nndct/quantization/torchquantizer.py:223: FutureWarning: Unlike other reduction functions (e.g. 'skew', 'kurtosis'), the default 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 taken will be eliminated, and the value None will no longer be accepted. Set 'keepdims' to True or False to avoid this warning.
    bnpf[i] = stats.mode(data)[0][0]

[VAIO_NOTE]: ==Exporting quant config.(quantize_result/quant_info.json)
Calibration finished. Config saved to ./quantize_result

--- STEP 2: EXPORTING ---
[VAIO_NOTE]: OS and CPU information:
    system --- Linux

```

```
vitis-al-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 nn dct_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:::is4 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-al-pytorch) vitis-al-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-al-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 nn dct_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_al/conda/envs/vitis-al-pytorch/lib/python3.8/site-packages/pytorch_nndct/quantization/torchquantizer.py:223: FutureWarning: Unlike other reduction functions (e.g. 'skew', 'kurtosis'), the default 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 taken will be eliminated, and the value None will no longer be accepted. Set 'keepdims' to True or False to avoid this warning.
  bnp[i] = 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
              val_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 nnct_st_CNNFSK_ed is torch.nn.Module.
[VAIQ_NOTE]: Finish tracing.
[VAIQ_NOTE]: Processing op... 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:::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
(vtits-al-pytorch) vtits-al-user@accollito-OMEN-2SL-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.

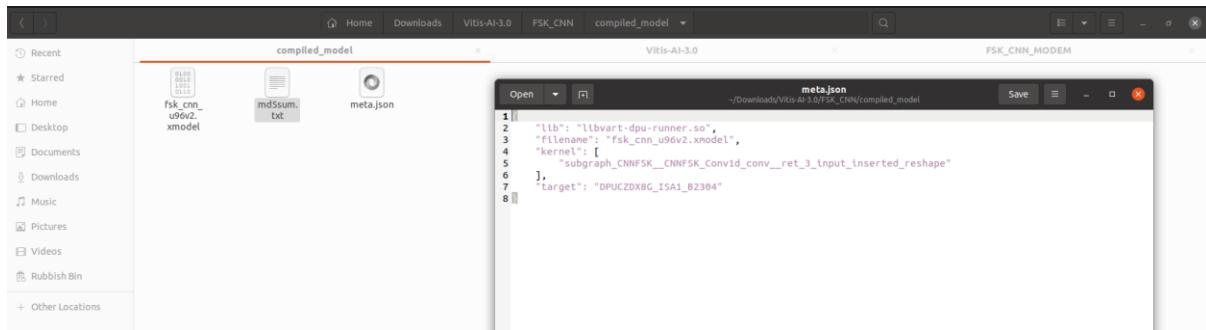
```
vai_c_xir --xmodel quantize_result/CNNFSK_int.xmodel --arch arch.json --  
net_name fsk_cnn_u96v2 --output_dir compiled_model
```

```

ERROR: NO FRONT END SPECIFIED
(vtits-al-pytorch) vtits-al-user@accollito-OMEN-2SL-Desktop-GT12-1xxx:/workspace/FSK_CNN$ val_c_xir --xmodel quantize_result/CNNFSK_int.xmodel --arch arch.json --net_name fsk_cnn_u96
=====
***** Vitis_AI compilation - Xilinx Inc. *****
=====
[UNILOG][INFO] Compile mode: dpu
[UNILOG][INFO] Target architecture: DPUCZDX8G_ISAI_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 e73bb1e5765ed9157285866fc80ffab, and has been saved to "/workspace/FSK_CNN/compiled_model/md5sum.txt"
(vtits-al-pytorch) vtits-al-user@accollito-OMEN-2SL-Desktop-GT12-1xxx:/workspace/FSK_CNN$ 

```

In the end we get what follows:

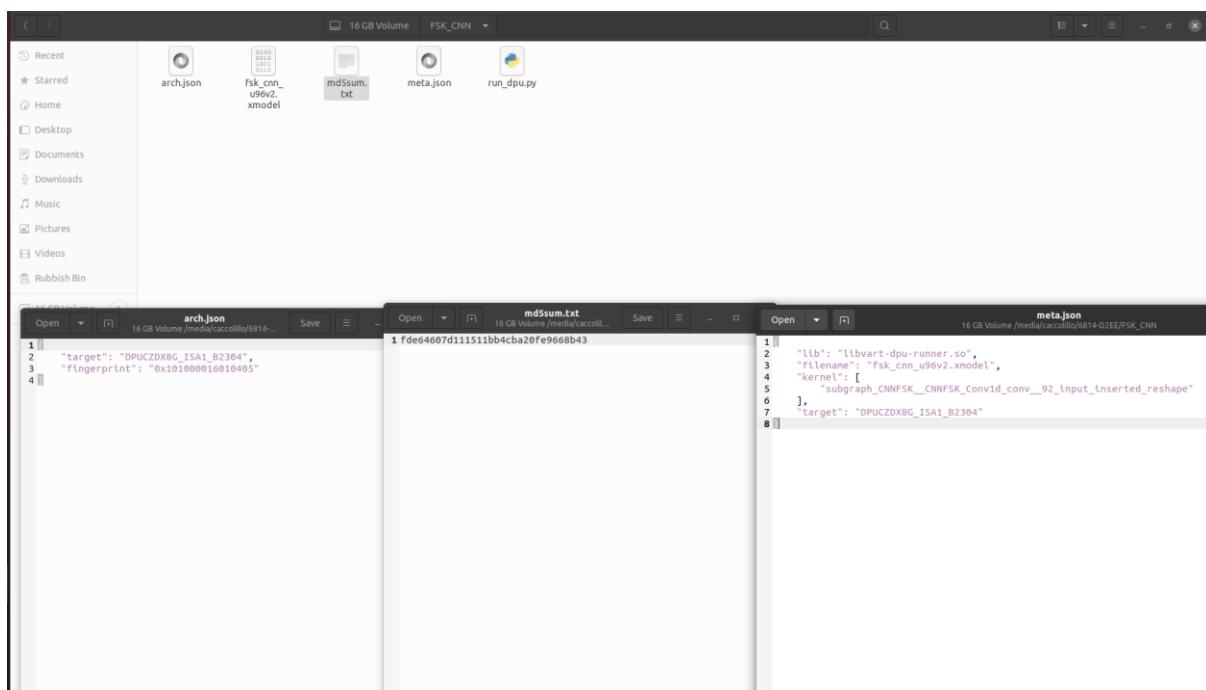


At the end of this process, we get a xmodel hardware-ready binary file "fsk_cnn_u96v2.xmodel".

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:

```

File Edit Log Configuration Control Signals View Help
u96v2-sbc-2022-2 login: petalinux
Password:
u96v2-sbc-2022-2:$ sudo su
Password:
u96v2-sbc-2022-2:/home/petalinux# mount /dev/sda /mnt
mount: /mnt: /dev/sda already mounted or mount point busy.
u96v2-sbc-2022-2:/home/petalinux# mount /dev/sdal /mnt
u96v2-sbc-2022-2:/home/petalinux cd /mnt
u96v2-sbc-2022-2:/mnt ls
FSK CNN
u96v2-sbc-2022-2:/mnt cd FSK CNN/
u96v2-sbc-2022-2:/mnt/FSK CNN# ls
arch.json fsk_cnn u96v2.xmodel md5sum.txt meta.json quantize.py run_dpu.py
u96v2-sbc-2022-2:/mnt/FSK CNN# python3 ./run_dpu.py

=====
DPU FSK TEST BENCH - ULTRA96 V2
=====
Bit #  Target  DPU Output (0, 1)  Prediction  Status
-----
WARNING: Logging before InitGoogleLogging() is written to STDERR
W1110 17:21:36.056199 816 dpu_runner_base_imp.cpp:73] CHECK fingerprint fail! model_fingerprint 0x101000056010405 is un-matched with actual dpu_fingerprint 0x101000016010405. Please re-compile xmodel with dpu_fingerprint 0x101000016010405 and try again.
FILED 17:21:36.056329 816 dpu_runner_base_imp.cpp:695] fingerprint check failure.
*** stack trace: ***
Aborted
u96v2-sbc-2022-2:/mnt/FSK CNN# export XLNX_ENABLE_FINGERPRINT_CHECK=0
u96v2-sbc-2022-2:/mnt/FSK CNN# python3 ./run_dpu.py

=====
DPU FSK TEST BENCH - ULTRA96 V2
=====
Bit #  Target  DPU Output (0, 1)  Prediction  Status
-----
0  0  (6.25, -7.00)  0  ✓ PASS
1  0  (6.25, -7.00)  0  ✓ PASS
2  0  (6.25, -7.00)  0  ✓ PASS
3  1  (-5.88, 7.50)  1  ✓ PASS
4  1  (-5.88, 7.50)  1  ✓ PASS
5  1  (-5.88, 7.50)  1  ✓ PASS
6  0  (6.25, -7.00)  0  ✓ PASS
7  1  (-5.88, 7.50)  1  ✓ PASS
8  1  (-5.88, 7.50)  1  ✓ PASS
9  1  (-5.88, 7.50)  1  ✓ PASS
-----
STATS SUMMARY
Total Bits Processed: 10
Accuracy: 100.00%

```

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.