236605 - Deep Learning
on Computation Accelerators

Neural Network Inference Acceleration on FPGA

Itay Zach
301227963

1 Abstract

The increasing use of Deep Neural Networks in many applications has driven great development in FPGA vendors' tools. FPGA's are meant to function as edge devices of a trained model in the inference phase of a system due to the low-power high-throughput and low-latency capabilities.
This work focuses on a single toy example application implementation on an FPGA using Xilinx's Vivado HLS tool. The idea is to take an existing system implemented using Python TensorFlow library in pure software and re-implement it on FPGA. To test the performance and speedup of the system, a PYNQ Z2 board was used.

2 Application Overview

The application chosen is an implementation of the Physical Layer of a communication system [2]. In classic digital communications system, each function (i.e. Matched-Filter, modulation, etc.) in the transmit-receive chains was optimized individually. The idea of implementing the physical layer with DNN comes from the assumption that optimizing the entire system as a whole will achieve better results in terms of performance (Probability of error vs. SNR). Using DNN, the entire system is trained all together rather than each block individually.

2.1 Digital Communications Theoretical Overview

The fundamental idea of a communication system is to transmit one out of M possible messages $\boldsymbol{s} \in \mathcal{M} = \{1, 2, ..., M\}$ with n discrete uses of the channel. In order to do that, the transmitter applies the transformation $f : \mathcal{M} \to \mathbb{R}^n$ such that $\boldsymbol{x} = f(\boldsymbol{s})$.
The channel then imposes imperfections to the transmitted signal (i.e. attenuation, noise, fading, distortion, etc.) and is usually described as AWGN (Additive White Gaussian Noise) channel, and is modeled with the conditional probability density function $p(\boldsymbol{y}|\boldsymbol{x})$, where $\boldsymbol{y} \in \mathbb{R}^n$ denotes the received signal.
The receiver then applies the transformation $g : \mathbb{R}^n \to \mathcal{M}$ and produces the estimated version of $\boldsymbol{s}$, $\hat{\boldsymbol{s}}$.
The communication rate of this communications system is $R = k/n$ [bit/channel use], where $k = \log_2(M)$ resulting k bits transmitted for each message M.

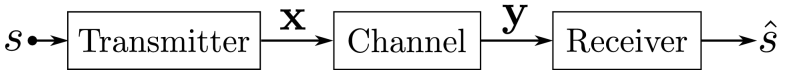


Figure 1: Basic communications system

2.2 Deep Learning Point of View

From a DL point of view, the communications system described in Figure 1 can be seen as an autoencoder. The purpose of a typical autoencoder is to find a lower dimension, non-linear representation of an input, and reconstruct it with minimal error. The goal of the proposed application is a little different. The encoder will learn a representation $\boldsymbol{x}$ of the input signal $\boldsymbol{s}$ that is robust to the channel impairments so that the recovered signal will have the minimal probability of error. Note that a typical autoencoder often removes redundancy, while this autoencoder will add redundancy in order to achieve minimal probability of error goal.

The suggested system is described in Figure 2. This system consists of a transmitter, the channel, and a receiver.
The transmitter (the encoder part) implementing $f(\boldsymbol{s})$ as several dense layers and a normalization layer. The normalization layer ensures that the physical signal $\boldsymbol{x}$ is valid with respect to the physical constraints, e.g. an energy constraint $||\boldsymbol{x}||_2^2 \leq n$, an amplitude constraint $|x_i| \leq 1, \forall i$ or an average power constraint $\mathbb{E}[|x_i|]^2 \leq 1, \forall i$. Note that the transmitter input is a one-hot vector $\mathbf{1}_{\boldsymbol{s}} \in \mathbb{R}^M$ consists of zeros in all its elements and a single one - in the sth index.
The channel consists of an additive Gaussian noise layer with a fixed variance.
The receiver (the decoder part) implements $g(\boldsymbol{y})$ and consists of a feed-forward NN as the transmitter, with a softmax activation whose output $\boldsymbol{p} \in (0,1)^M$ is a probability vector with all M possible messages. Applying $\arg\max(\boldsymbol{p})$ will result the desired $\hat{\boldsymbol{s}}$.

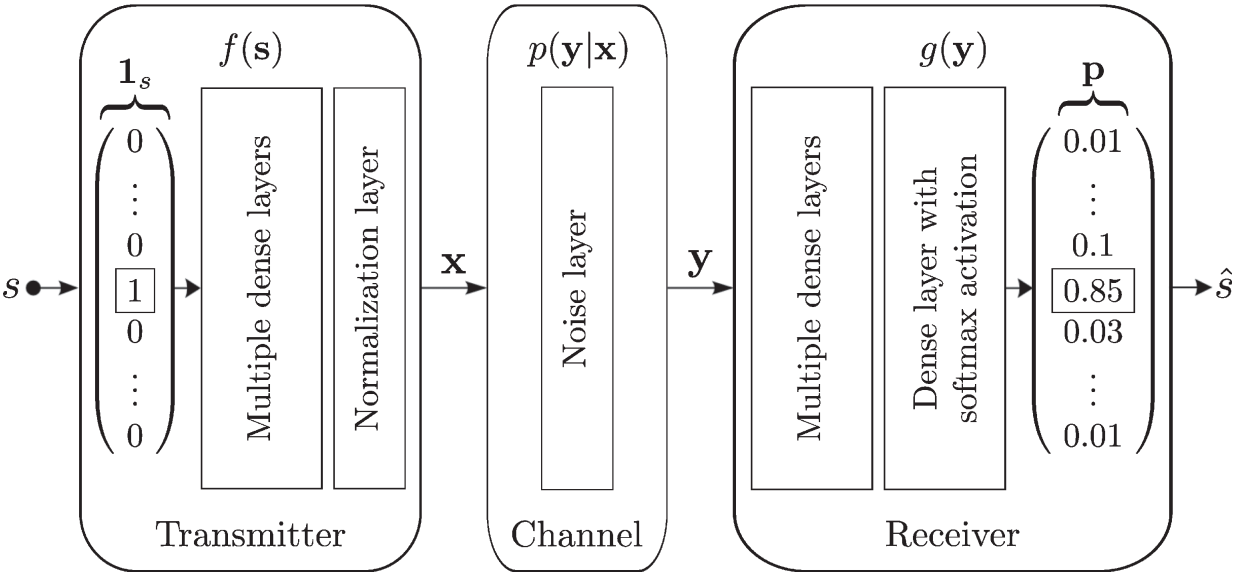


Figure 2: Autoencoder physical layer

3 TensorFlow Implementation

The reference design of the system was implemented using TensorFlow by Patel Dipkumar P. [3]. It can be seen from Table 1 that the TF implementation achieves the same results as the paper does [2].

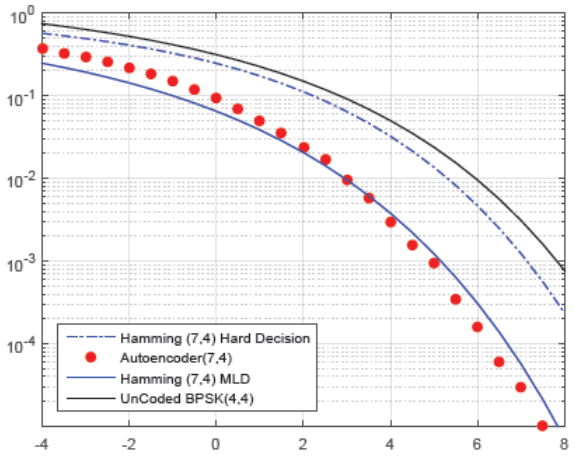
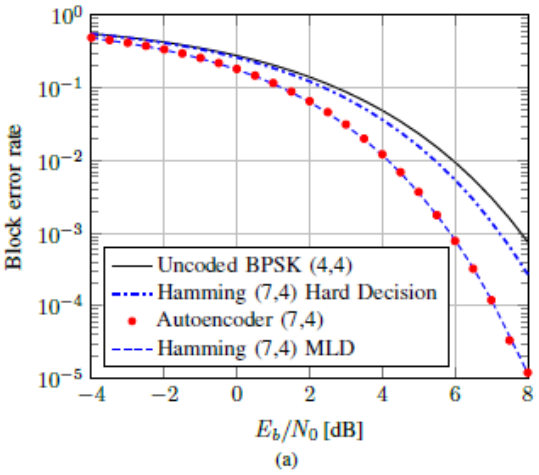
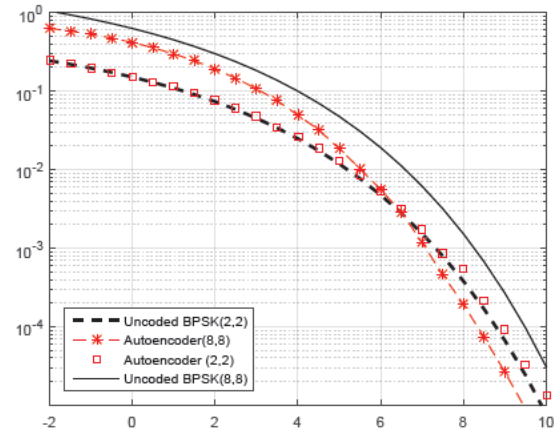
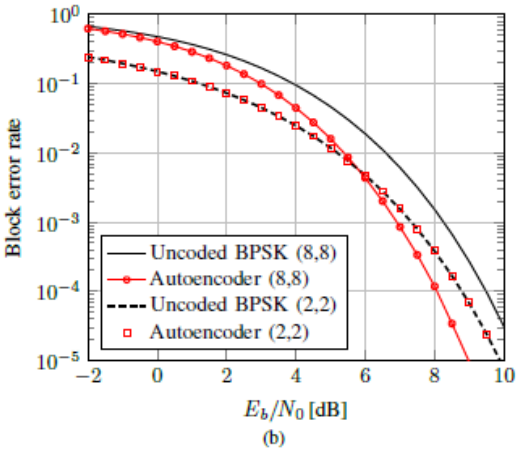
Re-generated	Paper
	
	

Table 1: Comparison of TF implementation with paper results

The following snip of code describes the implemented DNN structure of the TensorFlow model. In this implementation the following parameters were used:
`n = 2; k = 2; M = 4;`

```
# Transmitter (encoder)
in_sig = Input(shape=(M,))
enc    = Dense(M, activation='relu')(in_sig)
enc1   = Dense(n, activation='linear')(enc)
enc2   = Lambda(lambda x: np.sqrt(n)*K.l2_normalize(x,axis=1))(enc1)
# AWGN Channel
EbNo_train = 5.01187  # converted 7 db of EbNo
enc3 = GaussianNoise(np.sqrt(1/(2*R*EbNo_train)))(enc2)
# Receiver (decoder)
dec  = Dense(M, activation='relu')(enc3)
dec1 = Dense(M, activation='softmax')(dec)
# Autoencoder
autoencoder = Model(in_sig, dec1)
adam = Adam(lr=0.01)
autoencoder.compile(optimizer=adam, loss='categorical_crossentropy')
```

Running `autoencoder.summary()` prints the structure:

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 4)                 0
_________________________________________________________________
dense_1 (Dense)              (None, 4)                 20
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 10
_________________________________________________________________
lambda_1 (Lambda)            (None, 2)                 0
_________________________________________________________________
gaussian_noise_1 (GaussianNoise) (None, 2)             0
_________________________________________________________________
dense_3 (Dense)              (None, 4)                 12
_________________________________________________________________
dense_4 (Dense)              (None, 4)                 20
=================================================================
Total params: 62
Trainable params: 62
Non-trainable params: 0
```

4 High Level Synthesis

4.1 Vivado HLS Overview

Xilinx's Vivado High Level Synthesis (HLS) tool [7] provides a platform which translates C code into Register Transfer Level (RTL) code that can be synthesized into Xilinx's FPGA devices.
The design flow for HLS is described in Figure 3. The inputs to the tool are:

1. C/C++ design code (SystemC or OpenCL are supported as well).
2. C/C++ test-bench code.
3. Directives to the implemented design.

The methodology suggests to run a C-simulation in order to verify the correctness of the algorithm. After running a C-synthesis, it is possible to run an RTL simulation (called co-simulation) to verify that the translation was done as desired. After the code is verified, it is synthesized and packed into an IP using the Vivado IP Packager and instantiated in a top level design.

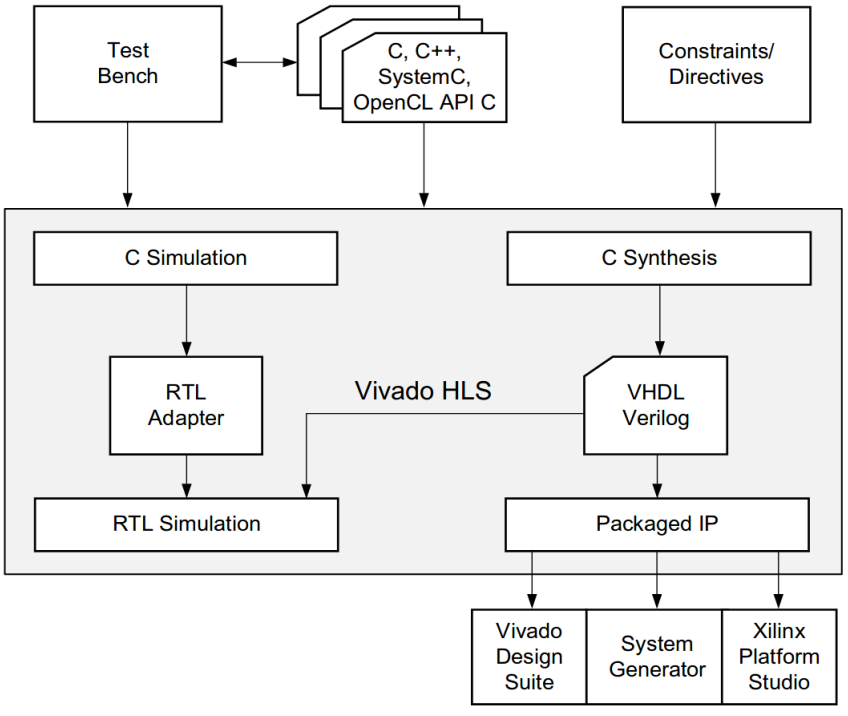


Figure 3: HLS Design Flow

4.2 HLS Packages

Several tools and packages were considered during this work.

1. CHaiDNN - "CHaiDNN is a Xilinx Deep Neural Network library for acceleration of deep neural networks on Xilinx UltraScale MPSoCs" [5].
2. hls4ml - As part of particle physics research, a group of researchers built an HLS package for Deep Learning models. This package contains common layers such as Convolution layer, Fully-Connected layer etc., and common activations such as ReLU, Sigmoid, etc. [1]

The hls4ml package was selected in this project. The pros of this package are the ease of use and the hardware-oriented implementation common layers. For example, Softmax activation which is inefficient to implement directly in its plain form $\mathbb{P}(y=j|\boldsymbol{x}) = \frac{e^{\boldsymbol{x}^T\boldsymbol{w}_j}}{\sum_{k=1}^{K} e^{\boldsymbol{x}^T\boldsymbol{w}_k}}$, was implemented using a Look-up Tables of 1024 entries for the exponent value and for the inverse values.

4.2.1 Normalization Layer

The last layer in the encoder is normalization layer, $\boldsymbol{y} = \frac{\boldsymbol{x}}{||\boldsymbol{x}||} = \frac{\boldsymbol{x}}{\sqrt{\sum_{i=1}^{N} x_i^2}}$. This layer was not implemented as part of the hls4ml package, so it had to be implemented as part of this project.

5 PYNQ

5.1 Overview

PYNQ (Python Productivity for ZYNQ) [6] is an open-source project from Xilinx® that makes it easy to design embedded systems with Xilinx Zynq® Systems on Chips (SoCs). The board that was selected to test the logic of the project is TUL's PYNQ Z2 board. The board comes with a pre-built image of Ubuntu 16.04 and board file for Vivado so it was possible to start working right on the interesting stuff.

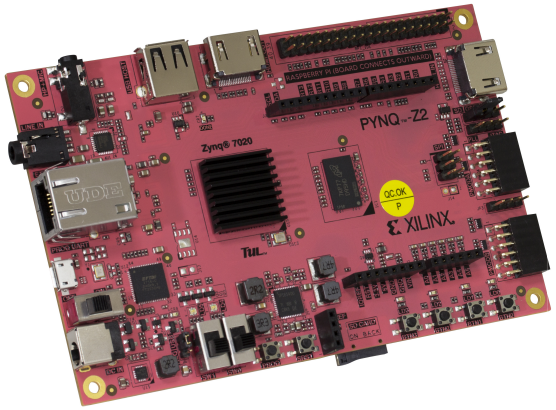


Figure 4: PYNQ Z2 board

5.2 Specifications

The PYNQ Z2 relevant features include: [4]

- ZYNQ XC7Z020-1CLG400C
  - (PS) 650MHz ARM® Cortex®-A9 dual-core processor
  - (PL) Programmable logic
    - 13,300 logic slices, each with four 6-input LUTs and 8 flipflops
    - 140 BRAM blocks of 36 Kbit each (4.9Mbit = 630KB total)
    - 220 DSP slices
- Memory and storage
  - 512MB DDR3 with 16-bit bus @ 1050Mbps
  - 16MB Quad-SPI Flash with factory programmed 48-bit globally unique EUI-48/64^{TM} compatible identifier
  - MicroSD slot

6 Implementation

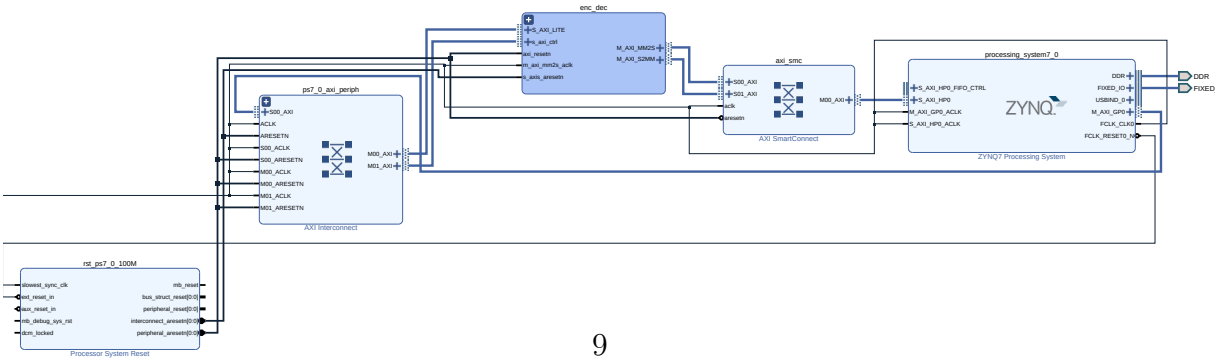
In this project, the FPGA implements an edge device for inference of a pre-trained network. The TensorFlow model was used to train the network and the weights were imported to the FPGA's internal BRAM cells.

6.1 Top Level Overview

There are 5 components in the top level of the design:

- ZYNQ - The Processing System that includes the ARM processors. The main interfaces are:
  - High performance 0 - Through the HP0 channel the PS transmits and receives data to/from the DDR.
  - General purpose 0 - Through the GP0 Master port the PS controls the PL memory mapped registers (such as DMA registers, the autoencoder registers, etc.)
  - Clocks are resets - The PS produces a single asynchronous reset and a 100MHz clock. The entire system works in this single clock domain.
- Processor System Reset unit - Generates the reset and clock of the AXI interconnect units.
- AXI smart connect - multiplexes between the DMA MM2S (Memory mapped to Stream) read from DDR channel, and the DMA S2MM (Stream to Memory Mapped) write to DDR channel.
- Autoencoder subsystem - The heart of the system. Includes the HLS implementation of the DNN and the DMA.

Figure 5: Top level



6.2 Autoencoder Subsystem

The subsystem includes the following components:

- AXI DMA - Controlled by the PS, this unit is responsible for reading the transmitted data from the DDR, sending it to the autoencoder, and writing it back to the DDR as the received data.
- Encoder Decoder unit - the autoencoder.
- Input and output FIFOs - to control the data flow and allow back-pressure from the autoencoder and the AXI Smart Connect to the HP0 interface.

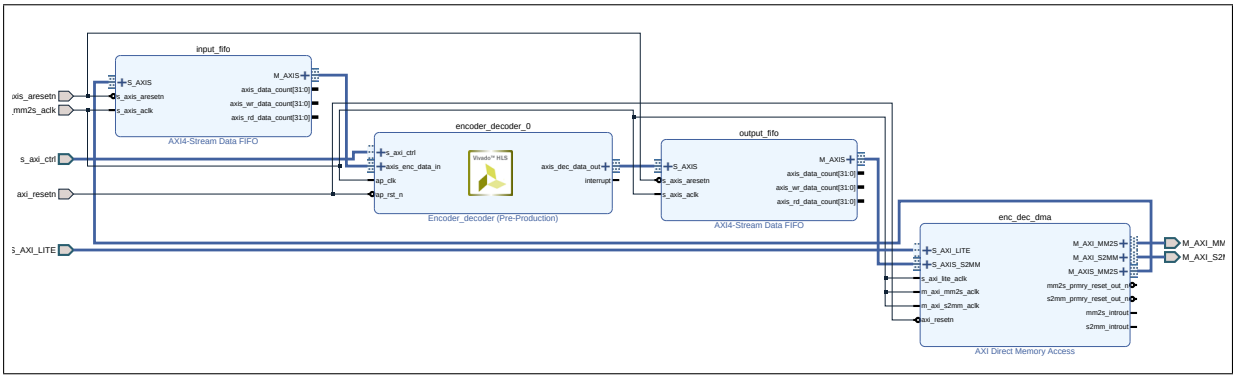


Figure 6: Autoencoder subsystem

6.3 Autoencoder HLS IP

The autoencoder written in HLS contains the following components:

- AWGN - Additive White Gaussian Noise. This Vivado HLS IP is used to generate a random Gaussian distributed noise to model the medium channel noise (distortion, delay, etc.). This IP is integrated inside the autoencoder from convenience of implementation considerations. The alternative to pass the transmitted data through another DMA, calculate the noise outside the PL and transmit it back was not worth the trouble.
- Encoder - the transmitter. The MM2S AXI Stream from the DMA is connected to its inputs, and after 2 dense layers and 1 normalization layer (see Chapter 3) the 2x32 bits IQ data is produced.
- Decoder - the receiver. This unit interprets the noisy data after the channel impairments. Its inputs are the 2x32 IQ data and it outputs a one hot vector of the decoded data to the AXIS S2MM channel

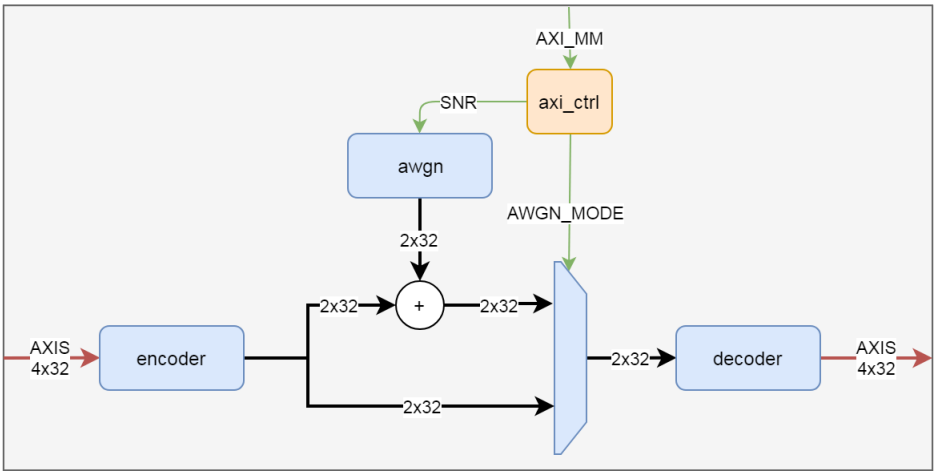


Figure 7: Autoencoder HLS IP

Table 2 shows the registers map of the Autoencoder IP. Note that the DMA register map is not described, as the Overlay package provides drivers that hide the registers from the user. That way, re-creating and re-configuring the DMA in Vivado is transparent to the user.

Address	Name	Width	Description
0x00000000	`CTRL`	8 bits	bit 0 - ap start (Read/Write/COH) bit 1 - ap done (Read/COR) bit 2 - ap idle (Read) bit 3 - ap ready (Read) bit 7 - auto restart (Read/Write)
0x0000004 - 0x000000c	`Reserved`		Unused.
0x00000010	`SNR_REG`	8 bits	Unsigned values in range [0.0, 16.0) in steps of 1/16 decibel
0x00000014	`Reserved`		Unused.
0x00000018	`AWGN_MODE`	1 bit	0x0 = AWGN enabled 0x1 = AWGN disabled (loopback mode)

Table 2: Register Map

6.4 Software

The PYNQ arrives with a pre-built image of Ubuntu 16.04 and a Jupyter Notebook server. The only concern the developer has is to build the application and upload the FPGA bitstream on the PL.
In SW side of things the flow is quite simple:

1. Load the bitstream using PYNQ's Overlay package.
2. Config the Autoencoder IP by:
   (a) Write the value 0x81 to the `CTRL` register(bit 0 = start; bit 7 = auto restart when done)
   (b) Write to `AWGN_MODE` register (0x0 = use AWGN; 0x1 = Loopback)
   (c) Write the desired SNR value to `SNR` register.
3. Config the DMA by:
   (a) Prepare the input buffer for write data to the Autoencoder IP, and output buffer for the read data from the Autoencoder IP.
   (b) Start the DMA receiver channel and DMA transmit channel.
   (c) Read the received data from output buffer.

7 Development Procedure

7.1 Autoencoder IP

In order to have a bit accurate model of the HLS IP to the TensorFlow, I ran C-simulations in HLS environment.
The simulations included recording the outputs of each layer in the encoder and the decoder and comparing the results to the TensorFlow corresponding outputs.
After the simulations were bit-accurate, I ran a loopback simulation (controlled by the SW register `AWGN_MODE = 1`) where the outputs of the encoder were connected to the inputs of the decoder to confirm the symbols are decoded correctly.

7.2 Proving on the Hardware

Using Vivado I built a block diagram (bd) of the top level design. In order to confirm data is transferred correctly, I first created a loopback between the DMA write channel and the DMA read channel, just to make sure the Python code running on the PS is correct. After the DMA mechanism worked as expected, I connected the HLS IP and made sure the encoder-decoder loopback works as expected.
When the loopback mode worked on the board, I started verifying the BER vs. SNR performance of the network. The greatest difficulty I came across was when I found out the BER vs. SNR performance graph does not match what I expected:

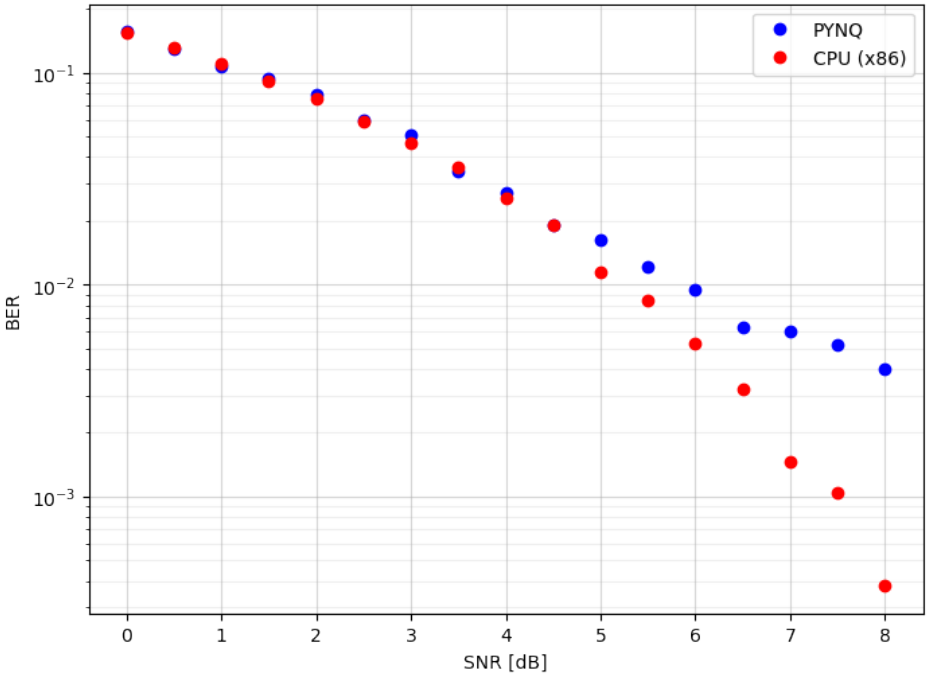


Figure 8: BER vs. SNR with bug in Softmax action

As seen from Figure 8, even though the SNR increases, the BER does not improve much. After debugging, I found out that the Softmax activation of the decoder acts weird. A result of a given output (first row) and the resulting output (second row) is shown in Figure 9. Marked in yellow is a result greater than 1, which is not feasible for Softmax by definition.

```
-1.37617 13.741 -17.847 -14.9386
0.015625 1 1.33301 0
```

Figure 9: Softmax error

More debugging proved that the Softmax exponent LUT values were overflowed. The values were represented as `ap_fixed<18,8>`. 8 bits for the integer part allowing the range [-128,127) was not enough bits in some cases, so the softmax value produced (sometimes) values greater than one.
The solution was to increase the integer part of the representation, using `ap_fixed<32,12>`. It is possible that the number of bits can be reduced, but optimization were not done as for now.

8 Results

In order to compare the results, the best thing to do was to run the same NN on the PYNQ's ARM CPU in it's plain SW form vs. running it using the FPGA NN implementation. Unfortunately, TensorFlow and Keras packages are not supported yet on PYNQ. To have an some idea of what could be achieved on the PYNQ, the comparison was done vs. a Quad Core Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz.

8.1 BER vs. SNR

In communications systems, the performance measure is BER (Bit Error Rate) vs. SNR (Signal to Noise Ratio). In order to verify the FPGA implementation does not degrade the TensorFlow implementation, 50,000 random symbols were transmitted with varying SNR (0.0 [dB] to 8.0 [dB]). The results are shown in Figure 10.

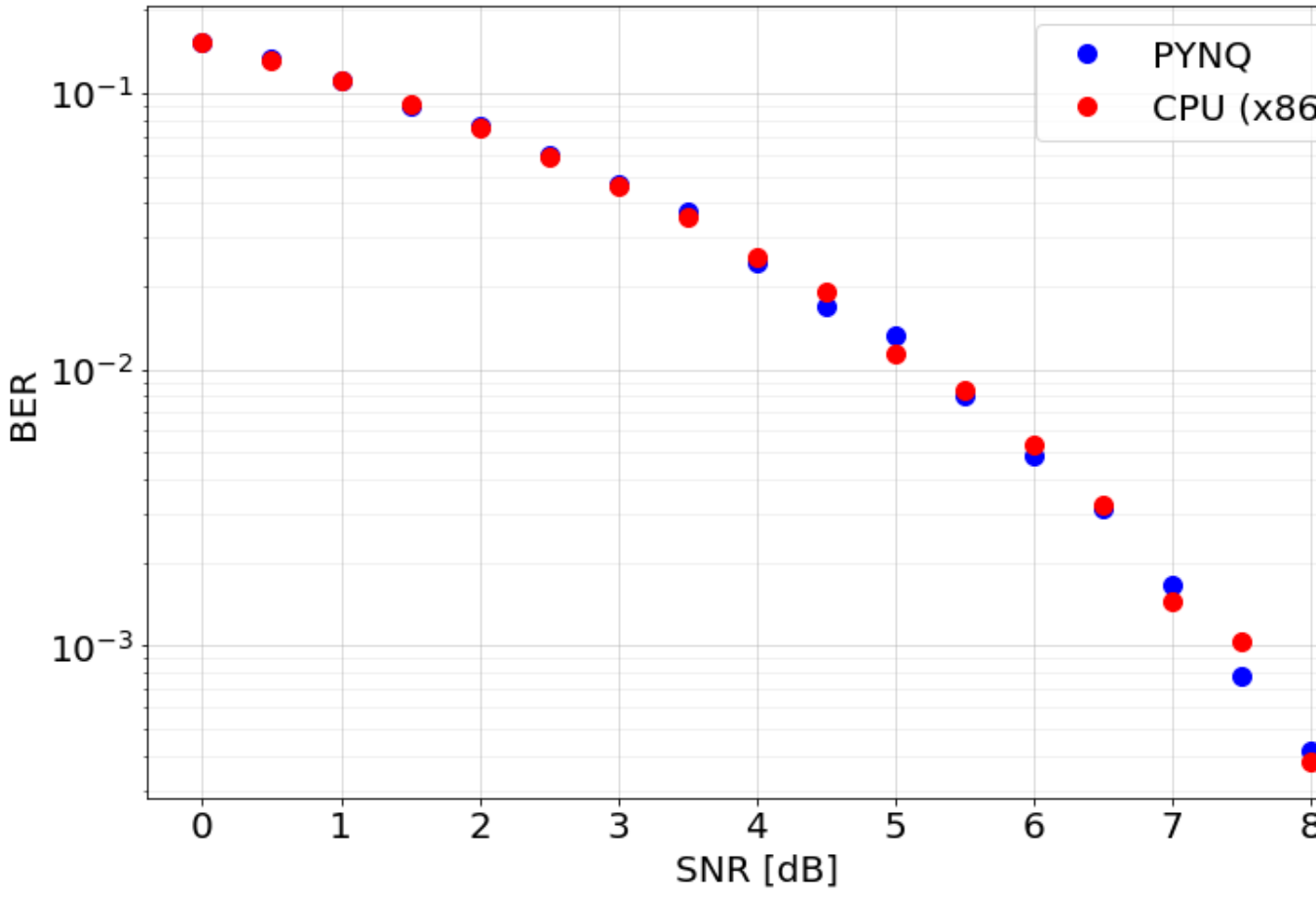


Figure 10: BER vs. SNR

8.2 Acceleration Ratio

To analyze whether it's worth using the FPGA implementation over the TensorFlow or not, 20 independent runs of increasing sizes of bytes were analyzed. From Figure 11 we can see that up to 200x16=3200 Bytes the execution time is the same. That is because of the overhead it costs to prepare the data to the DMA.
For more than 3200 Bytes, increasing the number of bytes is linear with execution time.

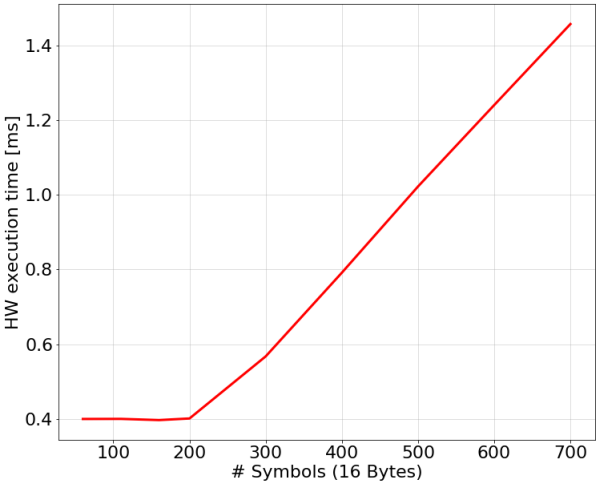


Figure 11: DMA overhead

Figure 12 shows the execution time on PYNQ and on CPU vs. the number of symbols transmitted and received. Each symbol is 16 Bytes. Figure 13 illustrates that speedup is achieved regardless of the amount of data.

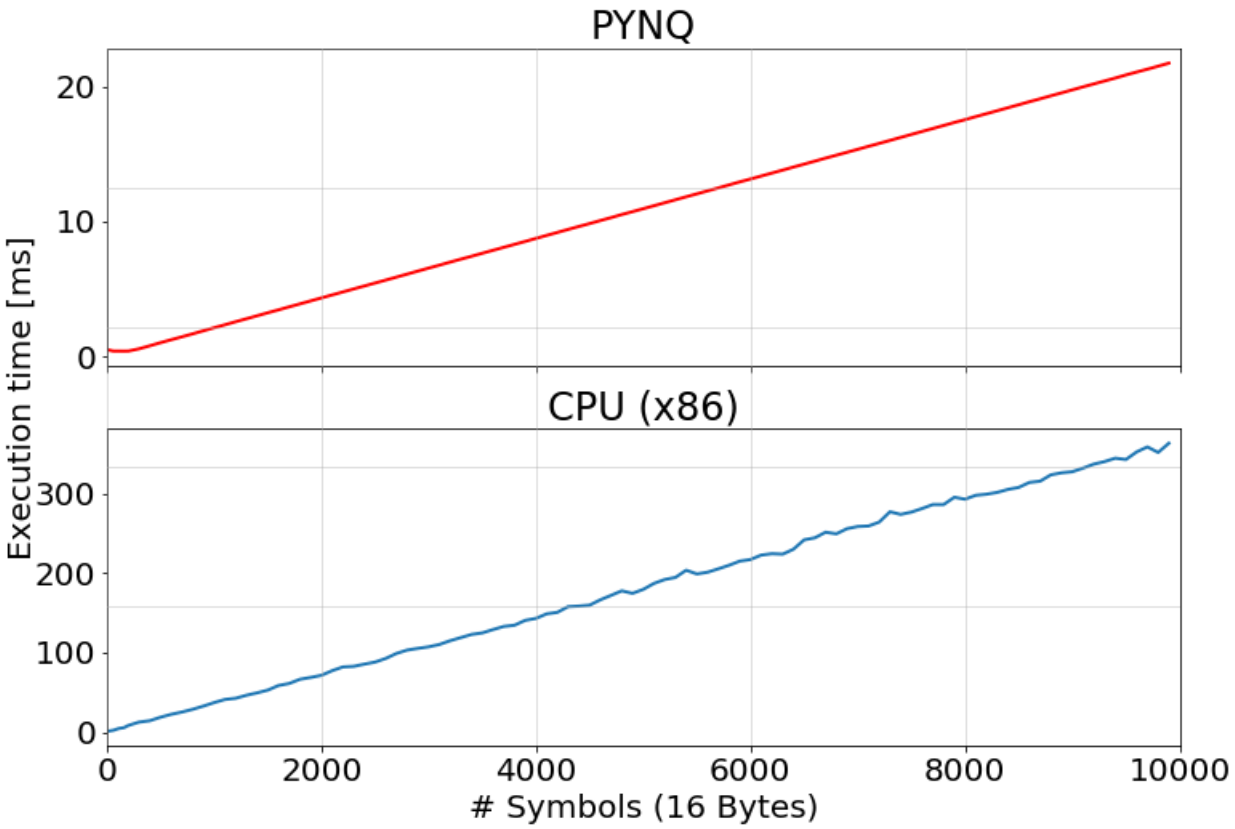


Figure 12: PYNQ vs. CPU execution time

Figure 14 shows the ratio between the SW execution time and HW execution time. On average, we can see there is an x16.5 acceleration.

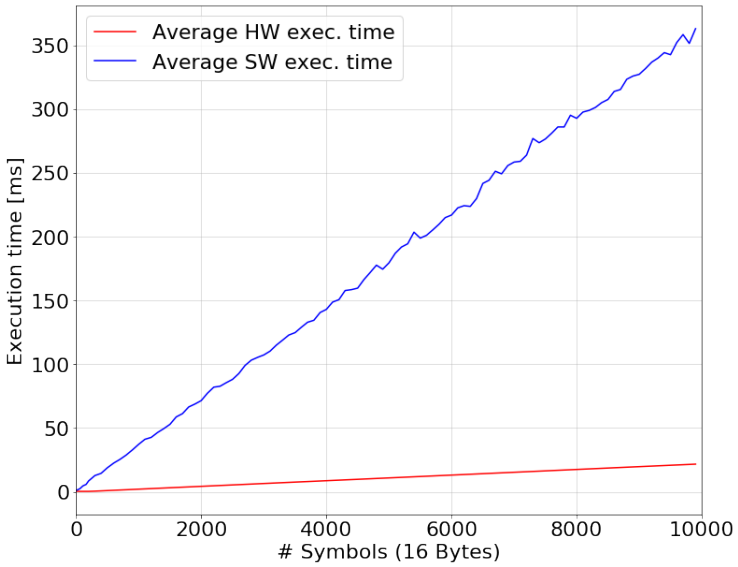


Figure 13: PYNQ vs. CPU execution time

9 Conclusions and Future Work

9.1 Acceleration

The main conclusion from this work is that accelerating the inference phase of a NN can be achieved using an FPGA.

9.2 Quantization errors

Quantization errors appear in this project due to several reasons:

- 64 bit floating point arithmetic in TensorFlow is now done in 32 bit fixed point.
- Functions such as division, exponents etc. are done using LUTs rather than calculating the true values.

Although quantization errors do exist, they did not affect the performance of the system (see Figure 10. We can safely conclude from Figure 13 that accelerating this NN over FPGA is the correct choice for any number of bytes.

9.3 Possible Improvements

1. Increase clock frequency. Current clock frequency is 100 MHz.
2. Optimize pipeline implementation in HLS.
3. Optimizing the fixed point representation. The entire HLS system is `ap_fixed<32,8>`, meaning 8 bits of integer (1 of them is the sign bit) and 24 bits of fraction. Perhaps after further analysis it is possible to reduce the number of bits, making the computation more efficient both utilization-wise and power consumption-wise.

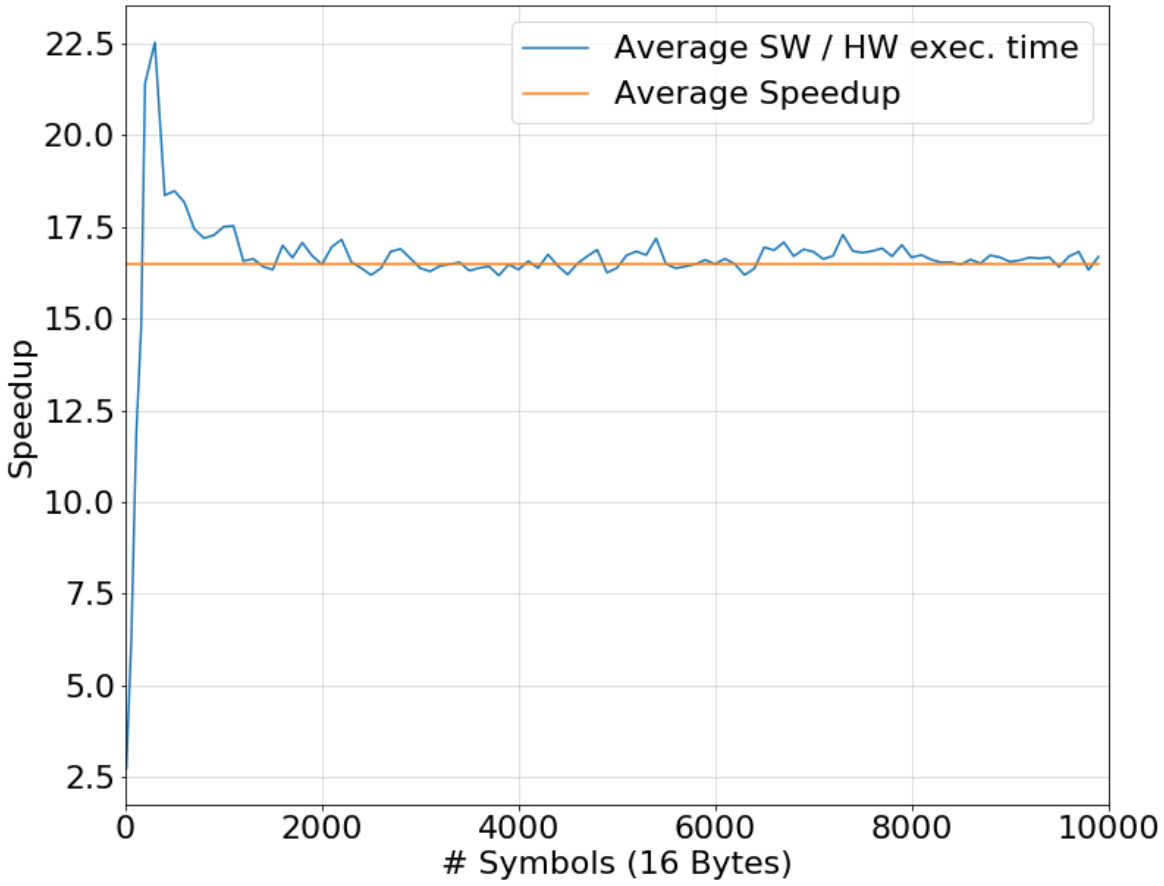


Figure 14: SW / HW acceleration ratio

References

[1] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and Z. Wu. Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation*, 13:P07027, July 2018.

[2] Timothy O'Shea and Jakob Hoydis. An introduction to deep learning for the physical layer. *IEEE Transactions on Cognitive Communications and Networking*, 3(4):563–575, 2017.

[3] Patel Dipkumar P. AutoEncoder Based Communication System. `https://github.com/immortal3/AutoEncoder-Based-Communication-System`. [Online; last accessed Oct. 2018].

[4] TUL. PYNQ User Manual. `https://d2m32eurp10079.cloudfront.net/Download/pynqz2_user_manual_v1_0.pdf`. [Online; last accessed Oct. 2018].

[5] Xilinx. CHaiDNN. `https://github.com/Xilinx/CHaiDNN`. [Online; last accessed Oct. 2018].

[6] Xilinx. PYNQ. https://www.pynq.io. [Online; last accessed Oct. 2018].

[7] Xilinx. Vivado Design Suite HLS User Guide (UG902). https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug902-vivado-high-level-synthesis.pdf. [Online; last accessed Oct. 2018].