Neural Network Inference Acceleration on FPGA

Itay Zach

Technion

itay.zach@campus.technion.ac.il

October 2018

Outline

Abstract

- The increasing use of Deep Neural Networks in many applications has driven great development in FPGA vendors tools.
- FPGAs 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 Xilinxs Vivado HLS tool.
- This work shows the performance acceleration gained by using FPGA.

Application Overview

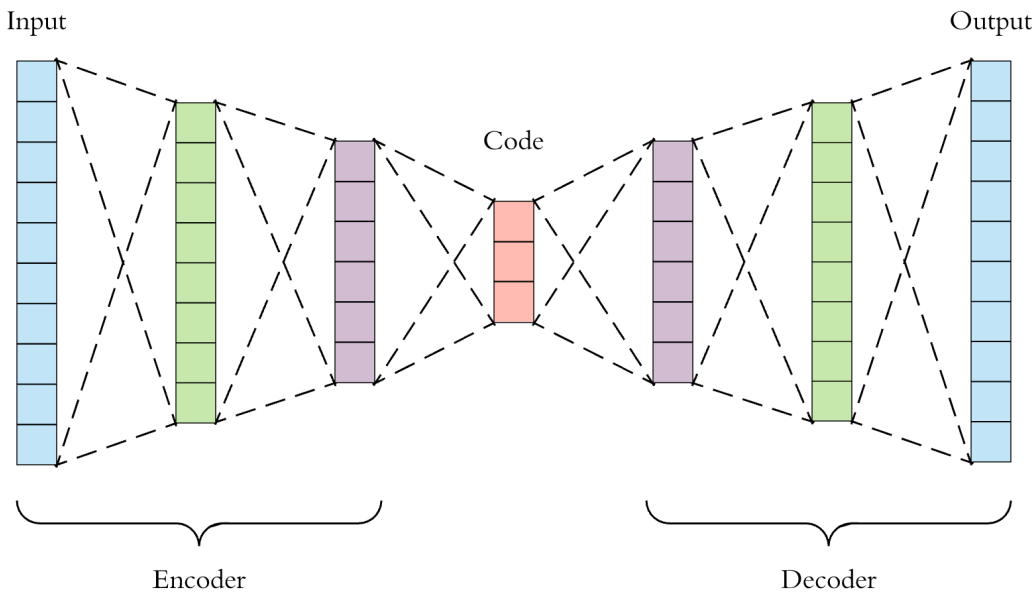
- The application chosen is an implementation of the Physical Layer of a communication system
- In classic digital communication each function is optimized **individually**.
- The idea here is to optimize the entire system **as a whole**.

Digital Communications Theoretical Overview

- Transmit one out of M possible messages $\boldsymbol{s} \in \mathcal{M} = \{1, 2, ..., M\}$ with n discrete uses of the channel.
- The transmitter applies $f : \mathcal{M} \to \mathbb{R}^n$ such that $\boldsymbol{x} = f(\boldsymbol{s})$.
- The channel imposes imperfections to the transmitted signal and modeled as AWGN.
  - i.e. attenuation, noise, fading, distortion, etc.
- The receiver then applies $g : \mathbb{R}^n \to \mathcal{M}$ and produces the estimated version of $\boldsymbol{s}$, $\hat{\boldsymbol{s}}$.

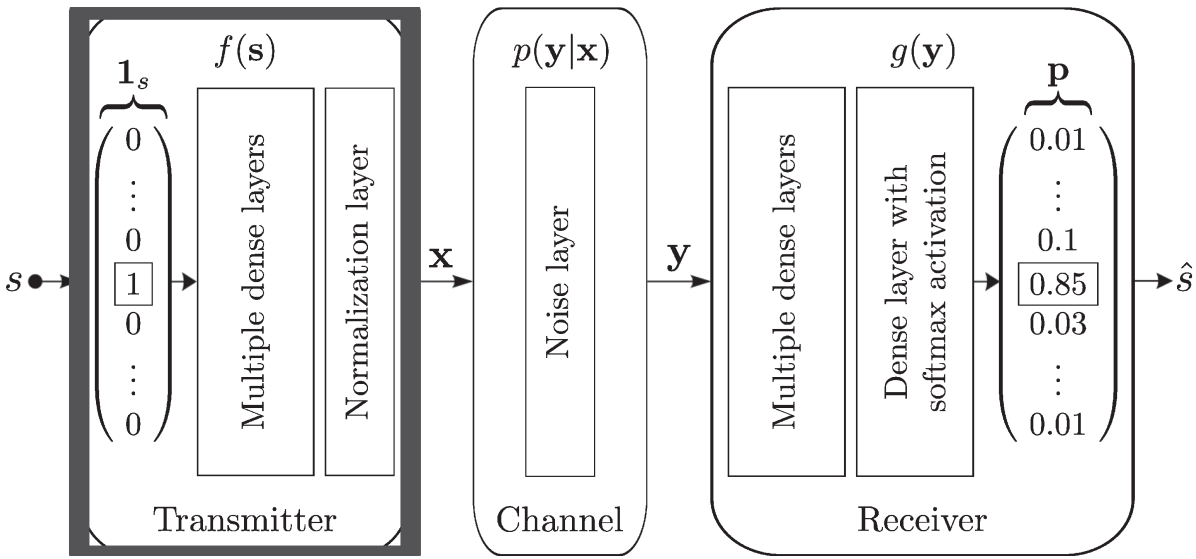
Deep Learning Point of View

- From DL point of view, this system is an **Autoencoder**.
- A typical autoencoder finds a lower dimension, non-linear representation of an input, and reconstruct it with minimal error.
- Here, the encoder will learn a representation $\boldsymbol{x}$ of the input signal $\boldsymbol{s}$ that is robust to the channel impairments.
- The recovered signal should have the minimal probability of error.



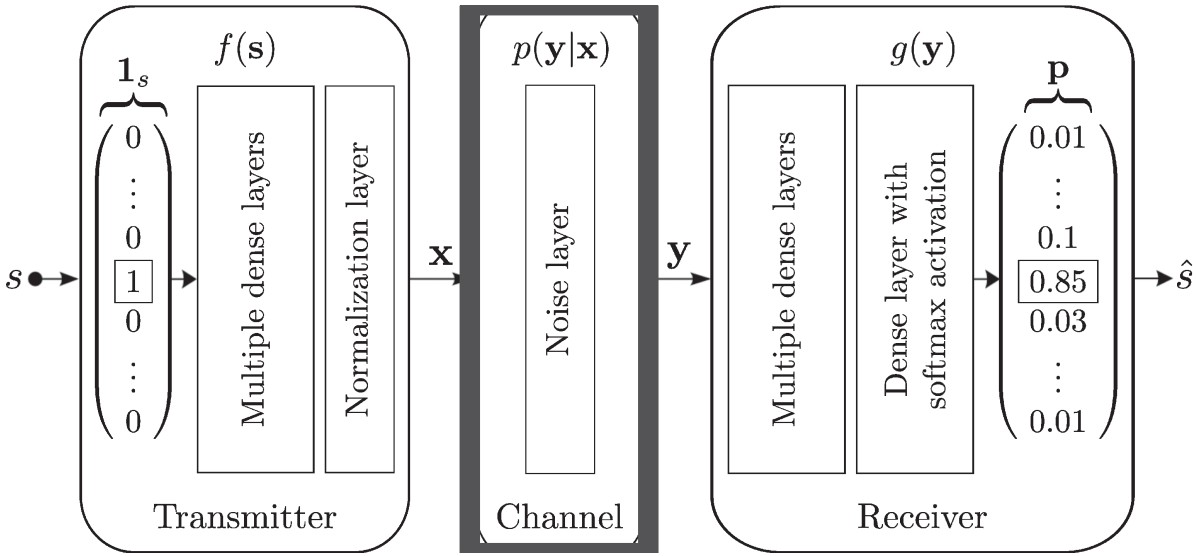
Deep Learning Point of View - The Transmitter

- Implements $f(\boldsymbol{s})$ with 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:
  - Energy constraint $||\boldsymbol{x}||_2^2 \leq n$
  - Amplitude constraint $|x_i| \leq 1, \forall i$
  - Average power constraint $\mathbb{E}[|x_i|]^2 \leq 1, \forall i$
- Note that the transmitter input is a one-hot vector $\mathbf{1}_{\boldsymbol{s}} \in \mathbb{R}^M$.



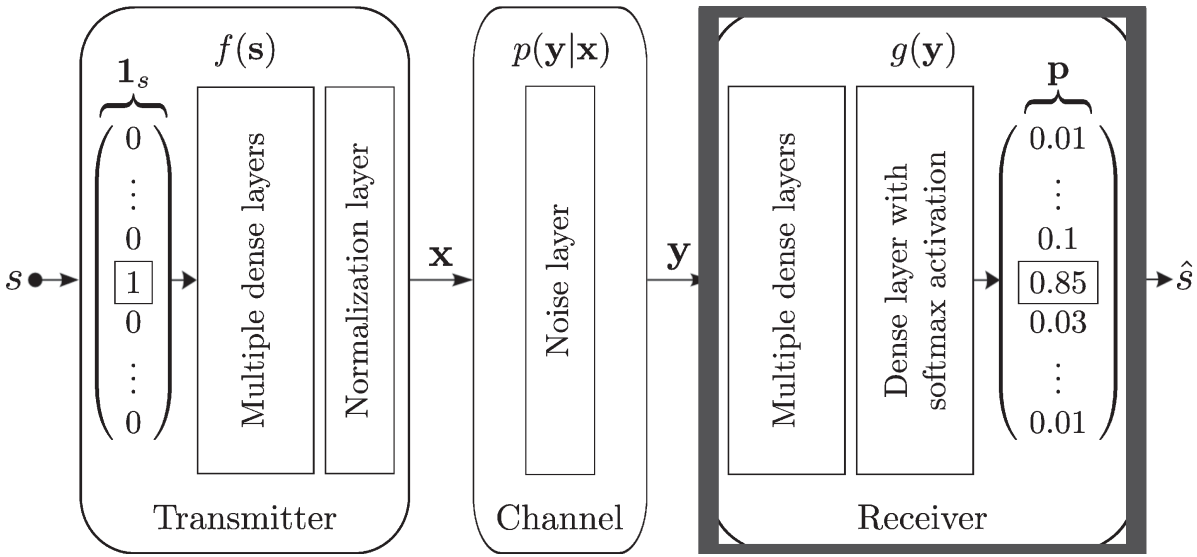
Deep Learning Point of View - The Channel

- The channel consists of an additive Gaussian noise layer with a fixed variance.



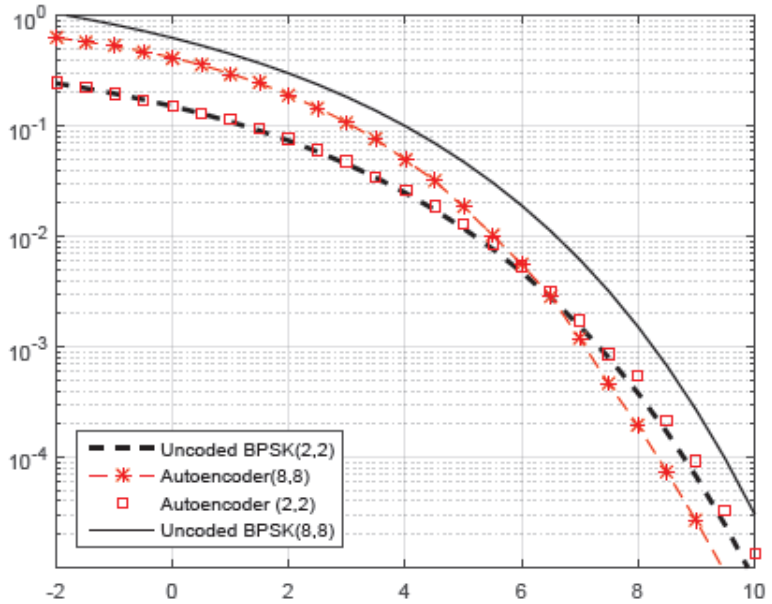
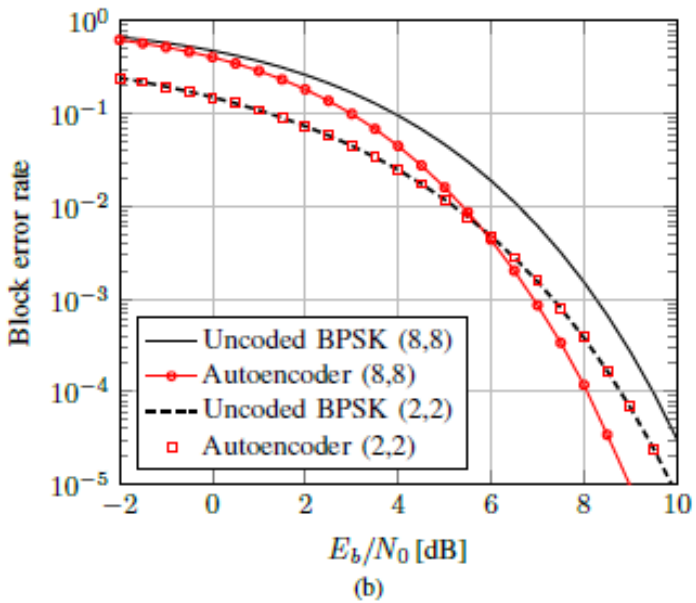
Deep Learning Point of View - The Receiver

- Implements $g(\boldsymbol{y})$ with several dense layers and a softmax layer.
- Softmax 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 $\hat{\boldsymbol{s}}$.



TensorFlow Implementation

- The reference design of the system was implemented using TensorFlow by Patel Dipkumar P. [3].
- The following table shows that the TF implementation achieves the same results as the paper does. [2]

Re-generated	Paper
	

TensorFlow Implementation

NN Structure

```
# 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*EbNo_train)))(enc2)
```

TensorFlow Implementation

NN Structure

```
# 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_
```

High Level Synthesis

- 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 the following figure. 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.

HLS Methodology

- 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.

HLS Methodology

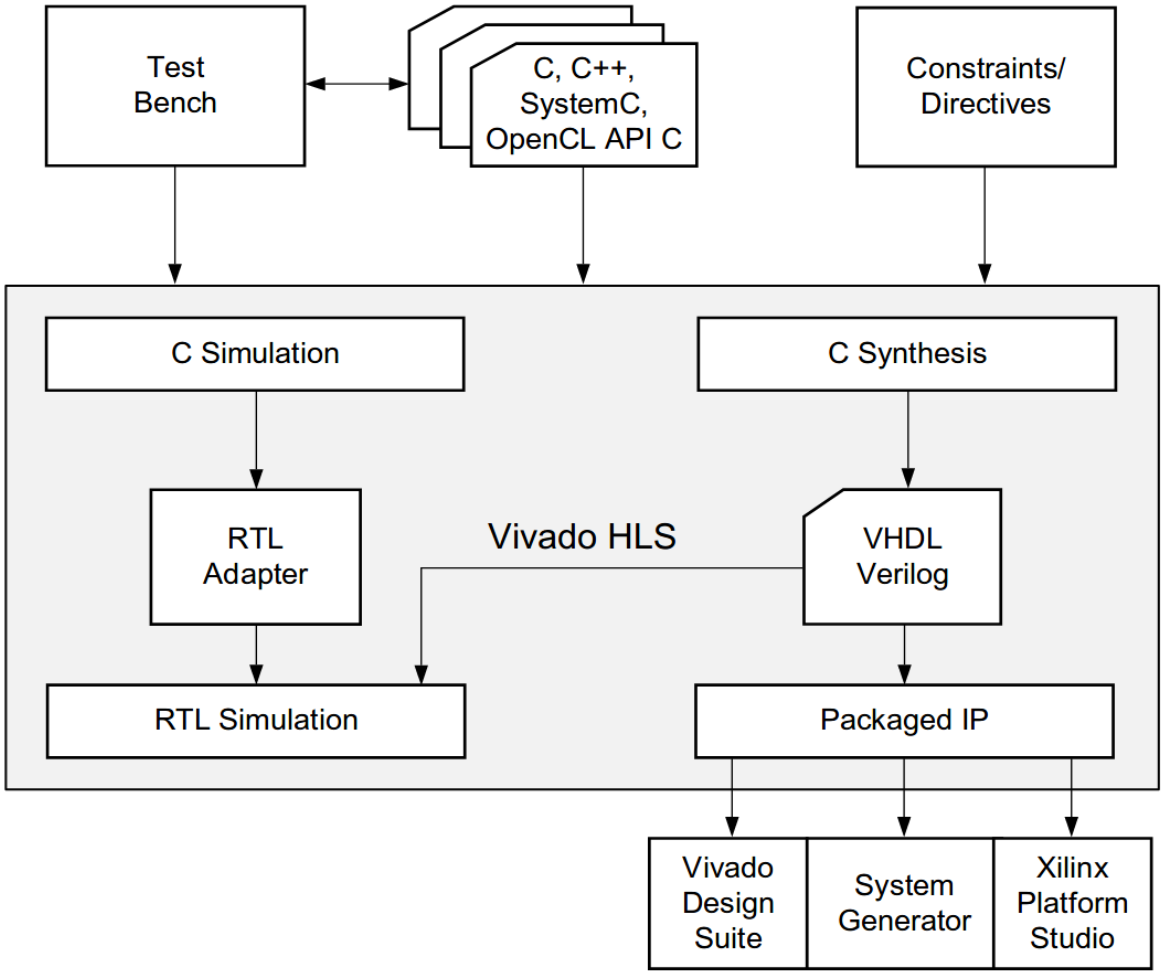


Figure: HLS Design Flow

HLS Packages

Several tools and packages were considered during this work.

1. **CHaiDNN** [5] - *"CHaiDNN is a Xilinx Deep Neural Network library for acceleration of deep neural networks on Xilinx UltraScale MPSoCs"*.
2. **hls4ml** [1] - 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.

hls4ml

- **hls4ml** package was selected in this project.
- Open source, easy to use, hardware-oriented implementation of common layers.
- For example, Softmax activation

$$\mathbb{P}(y = j|\boldsymbol{x}) = \frac{e^{\boldsymbol{x}^T \boldsymbol{w}_j}}{\sum_{k=1}^{K} e^{\boldsymbol{x}^T \boldsymbol{w}_k}}$$

was implemented using a Look-up Tables of 1024 entries for the exponent value and for the inverse values.

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.

PYNQ (Python Productivity for ZYNQ) Overview

- An open-source project from Xilinx that makes it easy to design embedded systems with Xilinx Zynq Systems on Chips (SoCs) [6].
- The selected board is TUL's **PYNQ Z2 board**.
- Comes with a pre-built image of Ubuntu 16.04 and board file for Vivado.

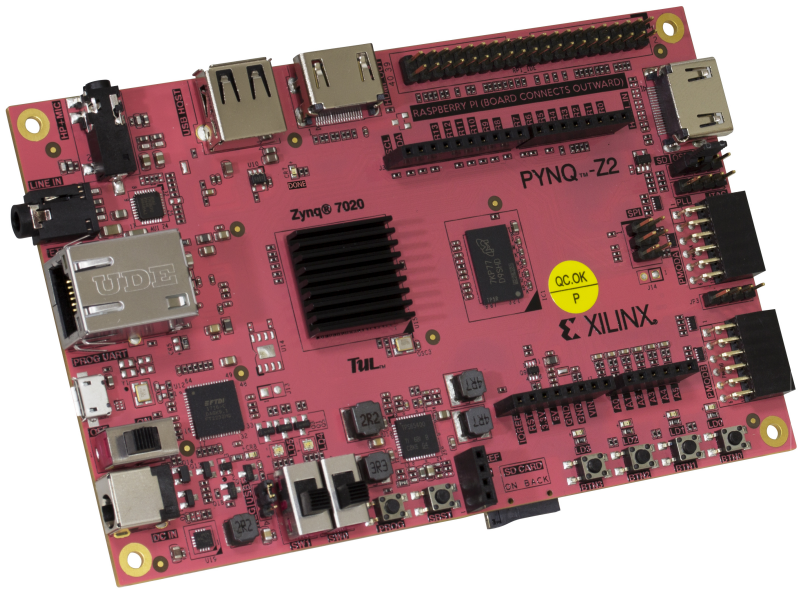


Figure: PYNQ Z2 board

PYNQ 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 compatible identifier
  - MicroSD slot

Top Level Overview

- 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.

Top Level Overview

The two main components in the top level of the design are:

- **ZYNQ** - The Processing System that includes the ARM processors. The main interfaces are:
  - High performance 0 - Connection to the DDR.
  - General purpose 0 - Control the PL memory mapped registers.
  - Clocks are resets - The entire systems works with a single clock of 100MHz and a single async reset.
- **Autoencoder subsystem** - The heart of the system. Includes the HLS implementation of the DNN and the DMA.

Top Level Overview

There are also:

- **Processor System Reset** - 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.

Top Level Overview

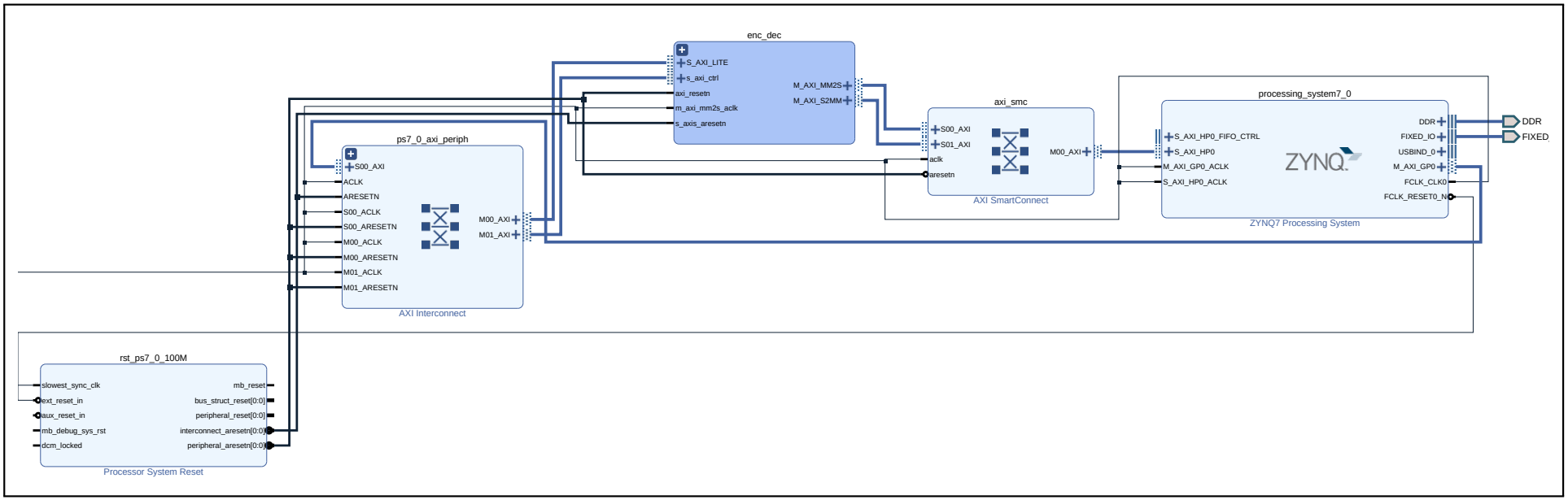


Figure: Top level

Autoencoder Subsystem

The subsystem includes the following components:

- **AXI DMA** - Controlled by the PS. Reads data from DDR - sends to Autoencoder IP - Writes data back to DDR.
- **Autoencoder HLS IP** - the autoencoder.
- **Input and output FIFOs** - To control data flow.

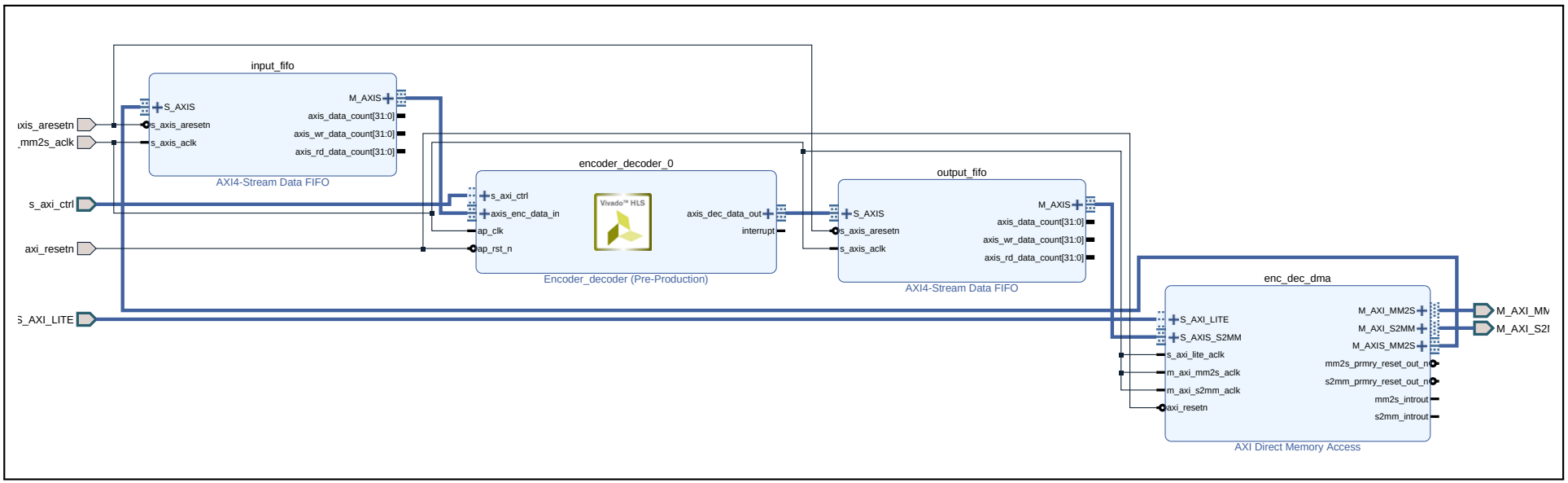


Figure: Autoencoder Subsystem

Autoencoder Subsystem

The autoencoder written in HLS contains the following components:

- **AWGN** - Additive White Gaussian Noise (Vivado HLS IP).
- **Encoder** - The transmitter.
- **Decoder** - The receiver.

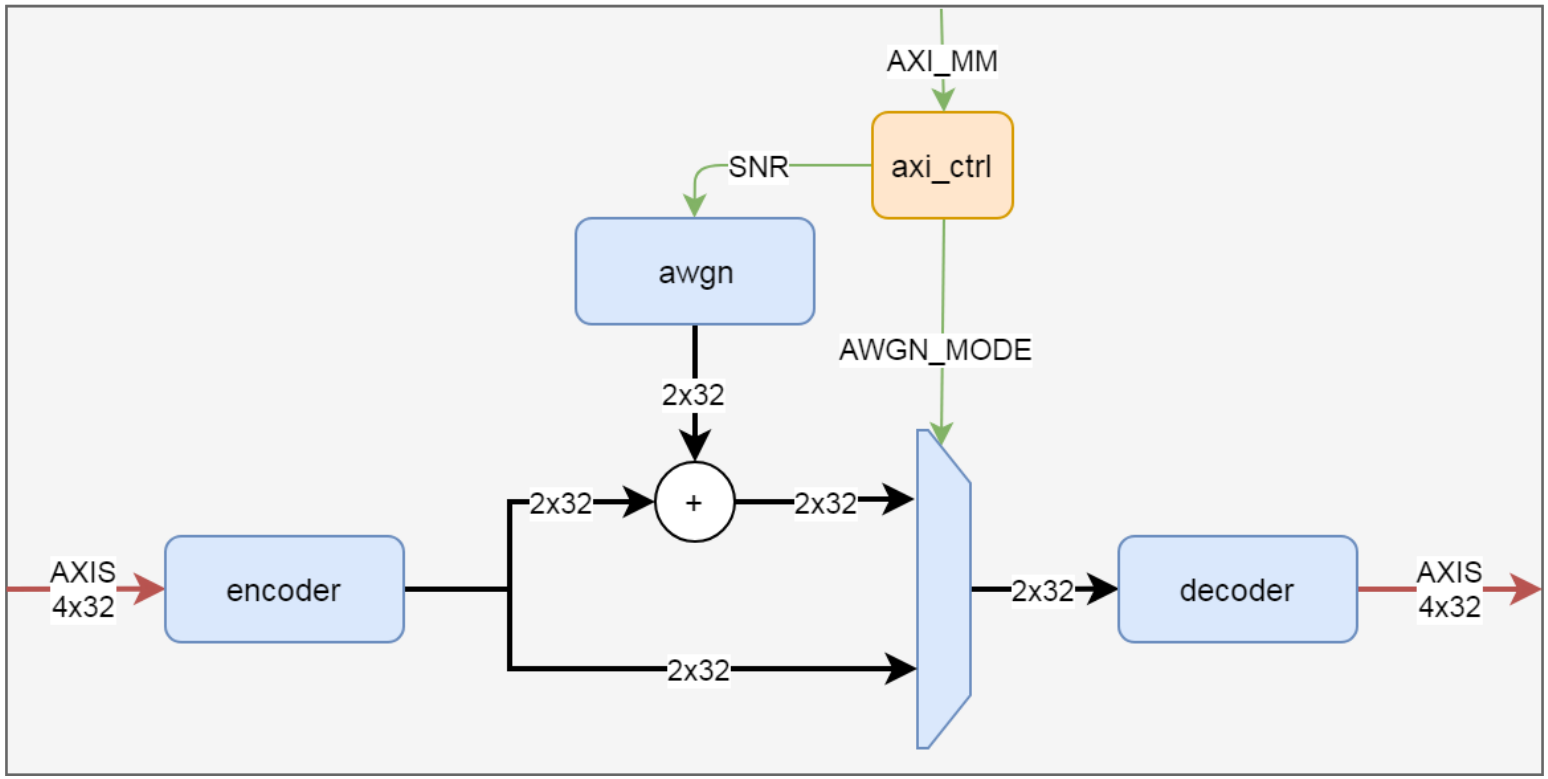


Figure: Autoencoder HLS IP

Autoencoder Subsystem Address Space

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)

SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake

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.

Software Flow

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

Development Procedure - Training

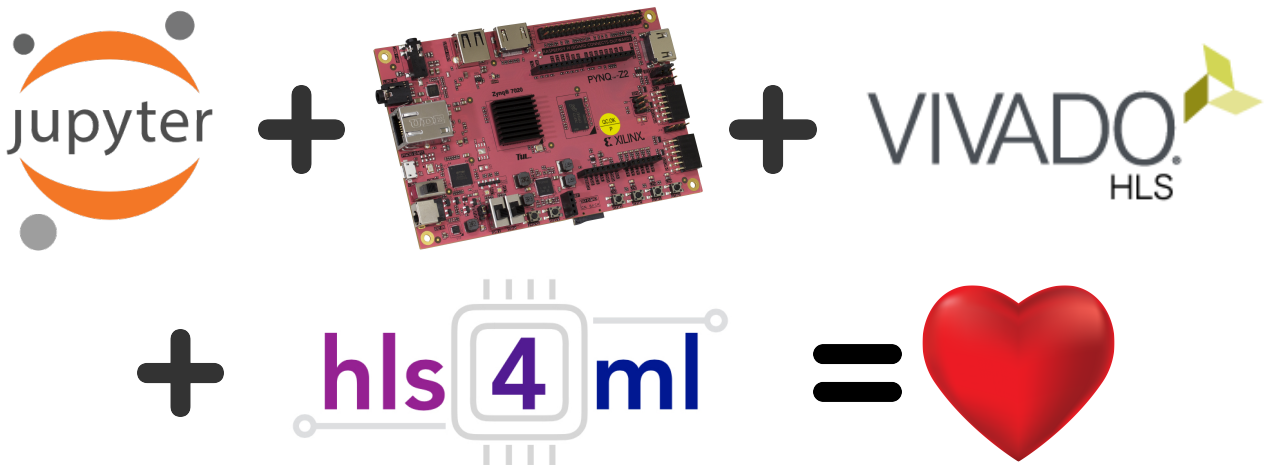
- The TensorFlow model was trained using $SNR = 7dB$
- The optimizer is Adam with learning rate of 0.01.
- The model was trained with 8000 samples.
- Training was done with 45 epochs and batch size of 32 samples.
- The training loss was 0.0044.
- **The weights** were imported from the model to the Autoencoder BRAM cells.

Development Procedure - Simulations in HLS

- **Bit accuracy** - In order to have a bit accurate model of the HLS IP to the TensorFlow model, two types of simulations were run and compared to the TensorFlow test phase:
  - C-simulations.
  - CO-simulations.
- **Loopback simulation** - After the simulations were bit-accurate, 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.

Development Procedure - Proving on the Hardware

- **DMA loopback** - DMA write channel to DMA read channel.
- **Autoencoder loopback** - Encoder to Decoder.
- **AWGN enabled** - Complete system.
  - But at first the BER vs. SNR performance graph wasn't as expected... (further details in following slides)

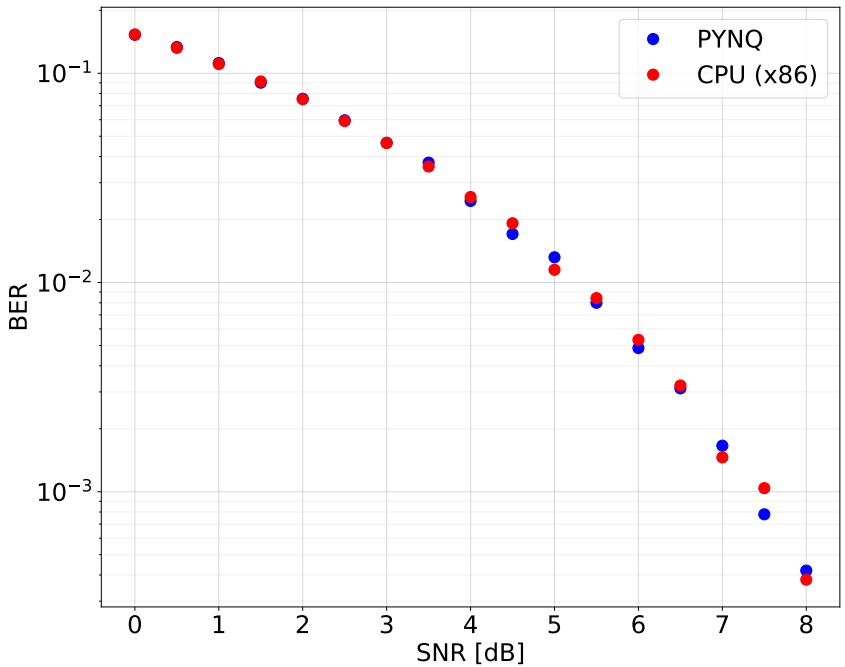


Results

- 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.

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]).



Interesting Bug

- During the development, an interesting bug was found.
- As seen from the figure, even though the SNR increases, the BER does not improve much.

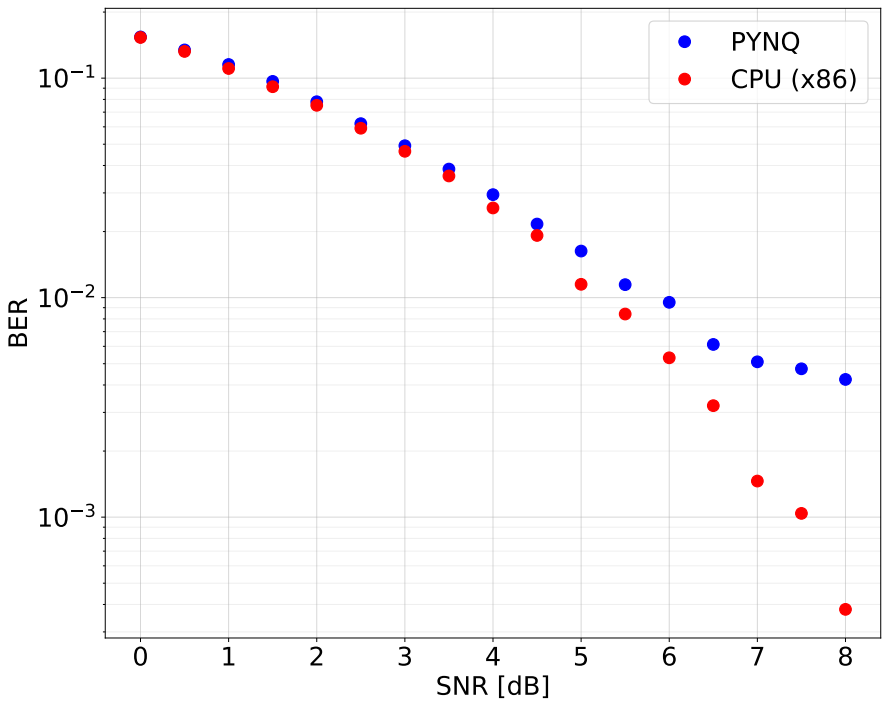


Figure: BER vs. SNR with bug

Softmax Bug

- After debugging, it seemed like the Softmax activation of the decoder acts weird. A result of a given output (first row) and the resulting output (second row) is shown below.
- 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: Softmax error

Softmax Bug

- 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.

Acceleration Ratio

- 20 independent runs of increasing sizes of bytes were analyzed.
- For 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.

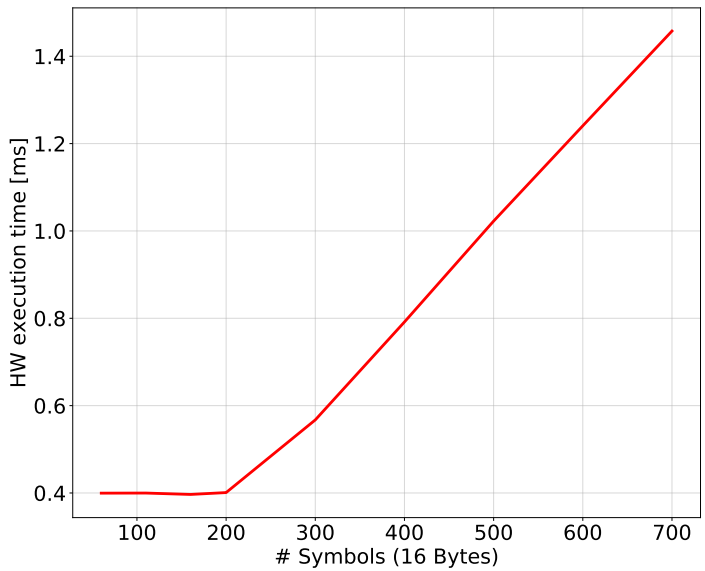


Figure: DMA overhead

Acceleration Ratio

- The execution time on PYNQ and on CPU vs. the number of symbols transmitted and received is shwon below.
- Each symbol is 16 Bytes.

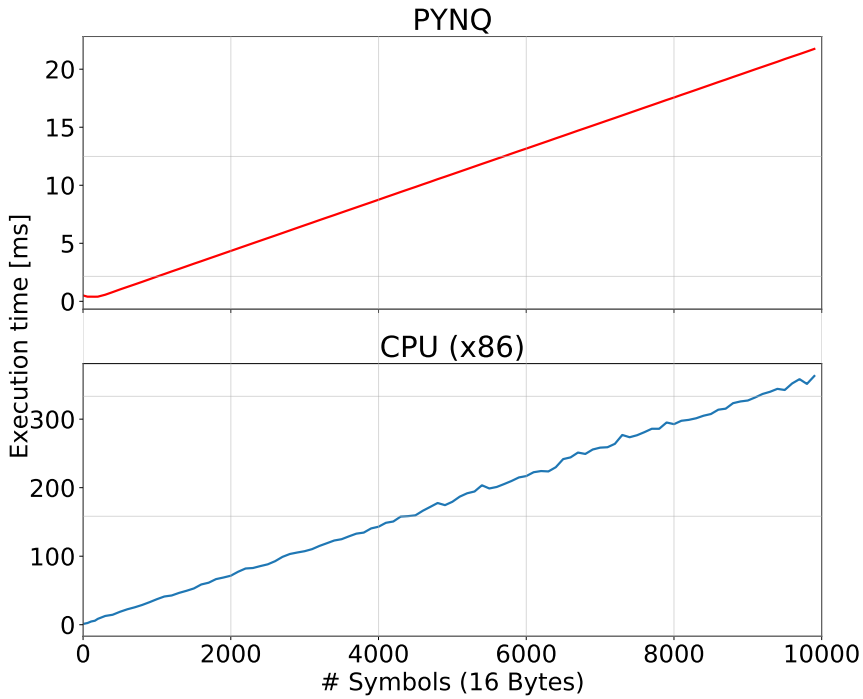


Figure: PYNQ vs. CPU execution time

Acceleration Ratio

- The figure illustrates that speedup is achieved regardless of the amount of data.

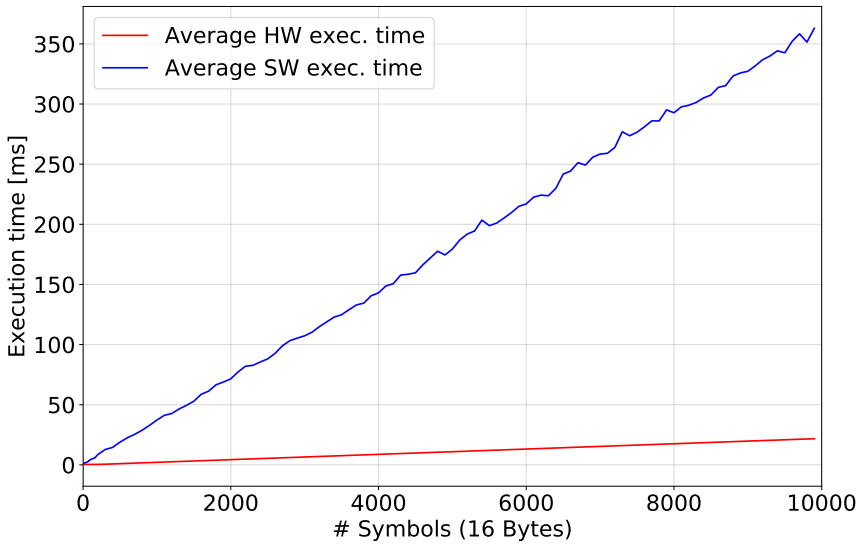


Figure: PYNQ vs. CPU execution time

Acceleration Ratio

- The following figure shows the ratio between the SW execution time and HW execution time.
- On average, we can see there is an **x16.5 acceleration**.

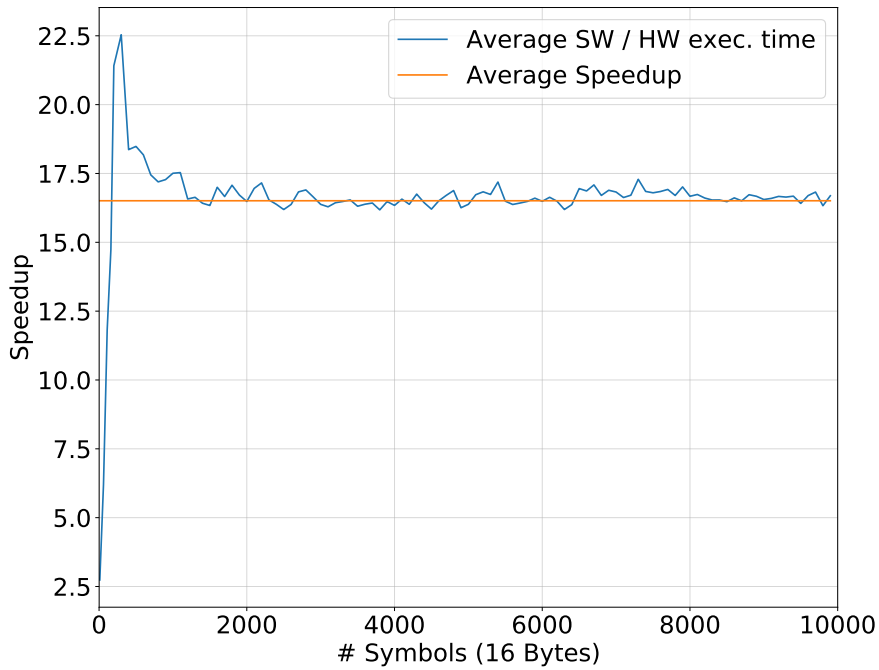


Figure: SW / HW acceleration ratio

Conclusions

Acceleration

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

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.
- Complicated 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

Conclusions

Bottom line

We can safely conclude that accelerating this NN over FPGA is the correct choice for any number of bytes.

Possible Improvements to This System

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.

References I


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.


Timothy OShea and Jakob Hoydis.
An introduction to deep learning for the physical layer.
IEEE Transactions on Cognitive Communications and Networking, 3(4):563–575, 2017.


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

References II

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

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

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

References III

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].

The End

Thank you