FPGA-based NN inference accelerator

Presented by **Ting-An Chen**

Outline

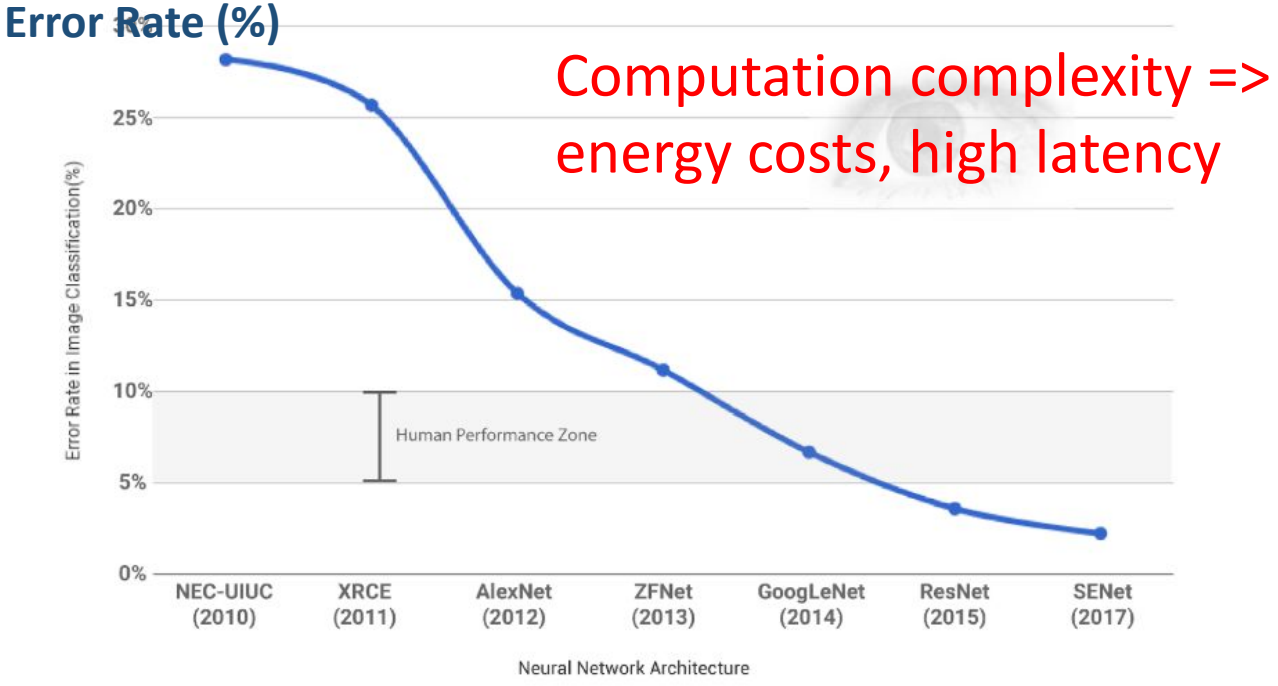
- Introduction & Preliminaries
  - Evolution
  - FPGA design and criteria
  - CNN – properties
- Optimization methodologies of CNN
  - Quantization
  - Weight reduction
- Conclusion
- From Vivado-HLS to PYNQ-Z2
- Demo – PYNQ-BNN
- Reference

Introduction & Preliminaries

Evolution

Phase 1. Performance driven

ILSVR（ImageNet Large Scale Visual Recognition Competition）



Source: https://chtseng.wordpress.com/2017/11/20/ilsvrc-%E6%AD%B7%E5%B1%86%E7%9A%84%E6%B7%B1%E5%BA%A6%E5%AD%B8%E7%BF%92%E6%A8%A1%E5%9E%8B/

Phase 2. Complexity driven – NN sizes

[1]	ResNet152	MobileNet	ShuffleNet
Year	2016	2017	2017
# Param	57M	4.2M	2.36M
# Operation	22.6G	1.1G	0.27G
Top-1-Accuracy	79.3%	70.6%	67.6%

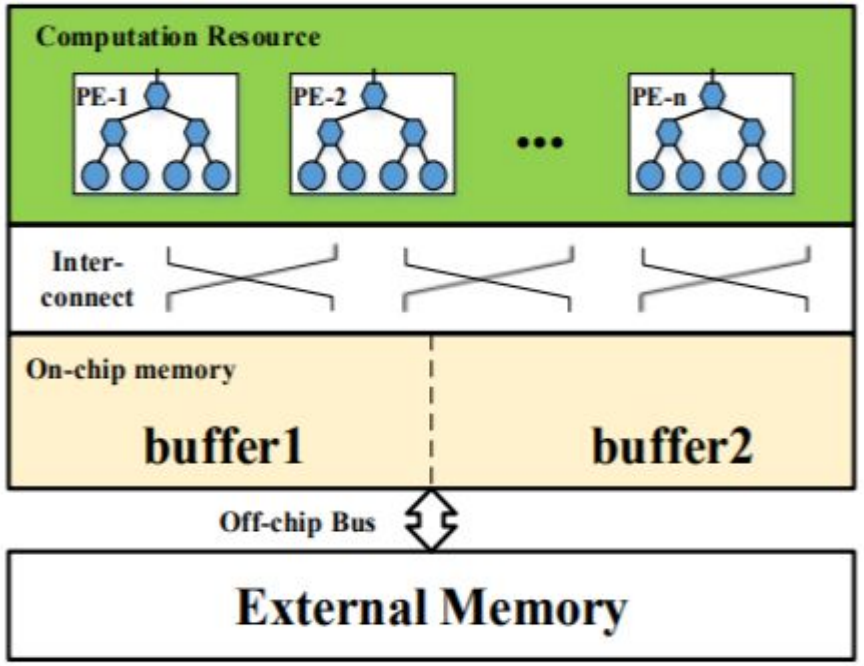
Phase 3. Complexity driven – Computation platform + Optimization

FPGA-based NN accelerator

>> Economical, energy efficient, reconfigurable

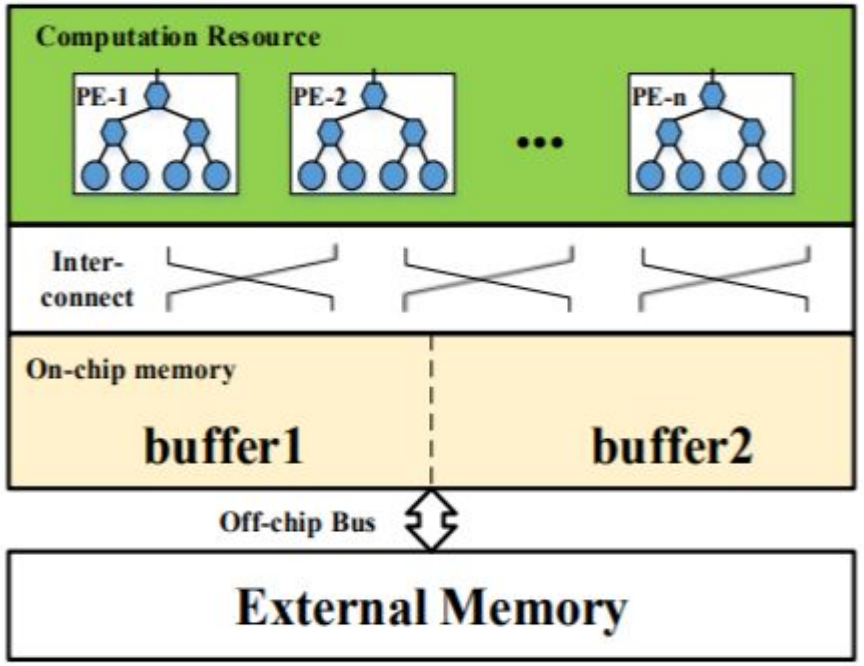
An accelerator design on FPGA [4]

- Processing elements (PEs) – computation unit
- On-chip buffer
- External memory
- On-/off-chip inter-connection



How can it be efficient?

- **Data allocation on PEs**
  - Data reuse (corresponding to algorithm and model)
  - E.g. weight stationary, output stationary, row stationary [2]
- **Data paths**
  - Bandwidth
  - Buffers to PEs
  - PE to PE, e.g. [2]
- **Model compression**



Design criteria

↑ • **Throughput** (1/s) – inferences per sec.

↓ • **Latency** (s) – time in an inference

↑ • **Workload** (GOP) – #MACs

• **Peak performance** (GOP/s)

• **Run-time performance** (GOP/s)

• **Utilization ratio*** (%) – average utilization ratio of PEs

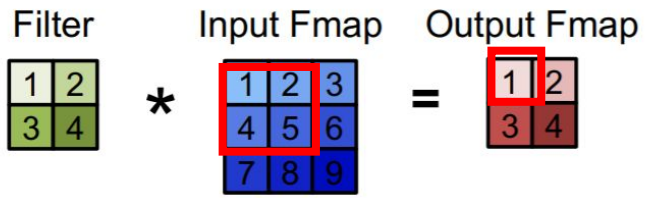
• **Energy efficiency** (GOP/J)

*Resources used ↓ => Utilization ratio ↓ ??

Utilization ratio ↓ => Less computations dealt with per clock

CNN (Convolutional Neural Network) [2][3]

- **Convolution layer** – computational-centric



For- loop + Unrolling

- **Fully connected layer** – memory-centric

Matrix-vector multiplication
(Feature map x weight)

Dimension reduction, e.g. PCA
Matrix decomposition

Other optimization methods?

Source: http://www.rle.mit.edu/eems/wp-content/uploads/2019/06/Tutorial-on-DNN-04-Kernel-Computations.pdf

Optimization methodologies of CNN

Optimization - previous works [3],[5-10]

Reduce size of operands for storage/compute **Reduce number of operations for storage/compute**

<table>
<tr><th colspan="3">Quantization - mapping data to a smaller set of levels</th><th>Weight Reduction</th></tr>
<tr><th></th><th>Static</th><th>Dynamic
(layer-wise adjustment)</th><td rowspan="3">• Pruning
- (Approx.) zero-weight pruning
- Regularization, e.g. Lasso

• Low-rank
(Matrix decomposition)
- SVD on FC layer [3]
- Optimization process on Conv. Layer [10]</td></tr>
<tr><td>Linear (fixed-point)</td><td>- Binarization [5]
- K-bit weights/activations</td><td>- Different fractional bit-widths [3]
- End layers to large bit-widths; middle layers to binary [6]
- Fine-tuning [7]</td></tr>
<tr><td>Non-linear</td><td>- Hash function [8]
- Clustering [9]</td><td>- …</td></tr>
</table>

Quantization – BNN [5]

BNN [*Umuroglu et al., FPGA'16*] [5]

- Objective – image classification
- NN – ConvNet (Conv*6, MaxPool*3, FC*3)
- Dataset – Cifar-10
- Quantization – 1 or 2 bit
- Platform – PYNQ-Z2 (512MB)
- Training – input image pixels scaled to [-1, 1] => floating-point parameters
- Inference – trained parameters & activated values => 1 or 2 bits

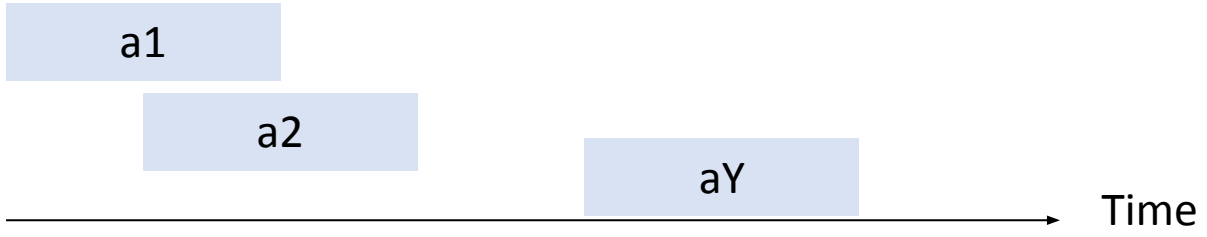
Model	Weight Bits	Activation Bits	Test Error
CNV	1	1	20.46%
CNV	1	2	16.37%
CNV	2	2	15.20%

Quantization – BNN [5] (Cont.)

BNN [*Umuroglu et al., FPGA'16*] [5]

- Conv. Layer – Pop-count substitutes for dot-product
- Batch norm – Find threshold τ_k s.t. $\text{BatchNorm}(a_k, \Theta_k) = \gamma_k \cdot (a_k - \mu_k) \cdot i_k + B_k = 0$
- Max pooling + Activation – $(\text{Max}(a_1, a_2, \ldots a_Y) > \tau^+)$ is replaced with

$$(a_1 > \tau^+) \vee (a_2 > \tau^+) \ldots \vee (a_Y > \tau^+)$$



Quantization – comparison [1]

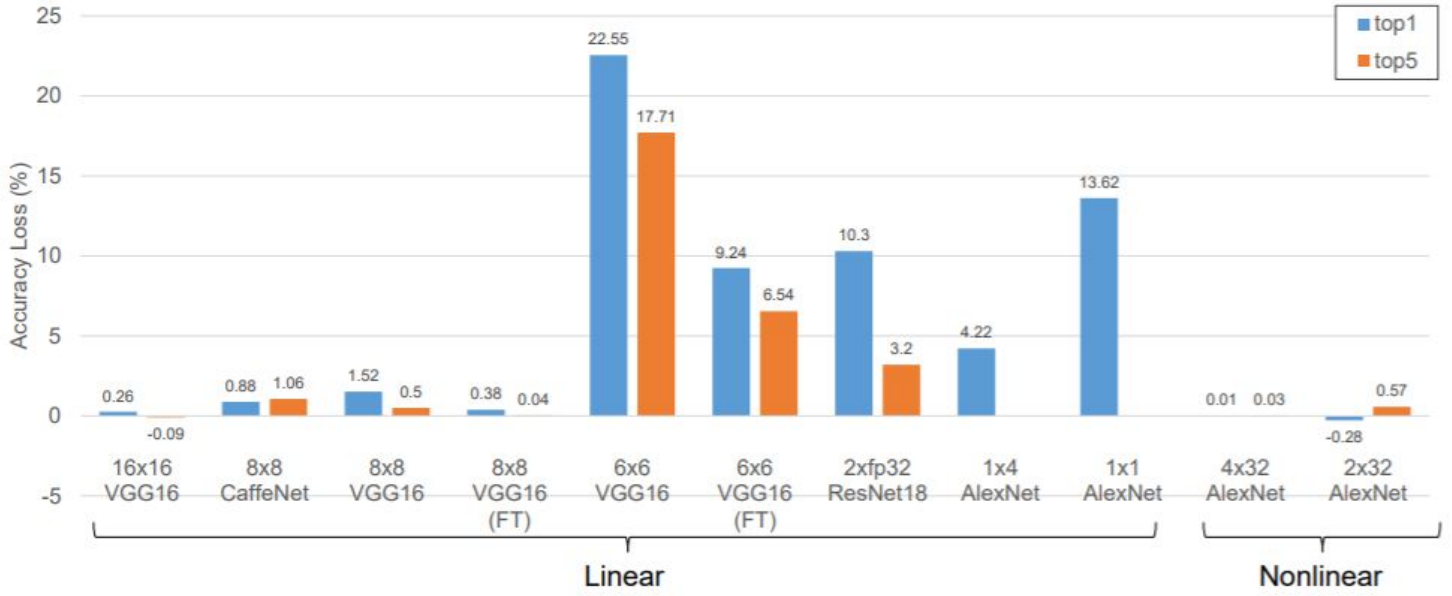


Fig. 3. Comparison between different quantization methods from [17, 20, 29, 49, 82, 83]. The quantization configuration is expressed as (weight bit-width)×(activation bit-width). The "(FT)" denotes that the network is fine-tuned after a linear quantization.

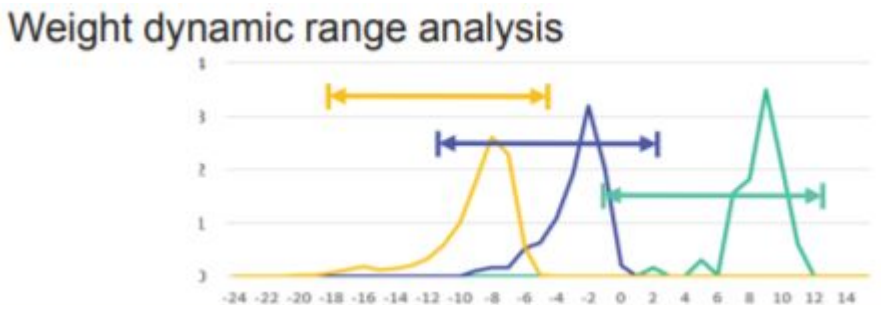
Pros – low memory, low computation complexity, energy efficiency

Cons – sacrifice to accuracy

Quantization – [3]

[*Qiu et al., FPGA'16*] [3]

- Objective – image classification
- NN – VGG16
- Dataset – ImageNet
- Quantization – Dynamic-precision
- Platform – Xilinx Zynq ZC706 (1G)



- Motivation – **vary ranges of weights** in different layer with **fixed reduced precision** can cause **overflow or underflow**
- Objective – variable reduced precision for each layer

Quantization – [3] (Cont.)

[*Qiu et al., FPGA'16*] [3]

- Quantization – Dynamic-precision
- Fixed-point to floating-point

$$n = \sum_{i=0}^{bw-1} B_i \cdot 2^{-f_l} \cdot 2^i$$

- **Weight quantization phase** – as <u>initialization</u> to avoid overflow / underflow

$$f_l = \underset{f_l}{\operatorname{argmin}} \sum |W_{float} - W(bw, f_l)|$$

- **Data quantization phase** – as <u>fine-tuning</u> to diminish the inference gap between fixed-point and floating-point model

$$f_l = \underset{f_l}{\operatorname{argmin}} \sum |x^+_{float} - x^+(bw, f_l)|$$

Quantization + SVD – [3] (Cont.)

[*Qiu et al., FPGA'16*] [3]

- Quantization – Dynamic-precision
- SVD

Table 2: The Memory footprint, Computation Complexities, and Performance of the VGG16 model and its SVD version.

Network	FC6	# of total weights	# of operations	Top-5 accuracy
VGG16	25088×4096	138.36M	30.94G	88.00%
VGG16-SVD	$25088 \times 500 + 500 \times 4096$	50.18M	30.76G	87.96%

Precision v.s. Accuracy

Table 3: Exploration of different data quantization strategies with state-of-the-art CNNs.

Network	CaffeNet			VGG16							VGG16-SVD		
Experiment	**Exp 1**	**Exp 2**	**Exp 3**	**Exp 4**	**Exp 5**	**Exp 6**	**Exp 7**	**Exp 8**	**Exp 9**	**Exp 10**	**Exp 11**	**Exp 12**	**Exp 13**
Data Bits	Single-float	16	8	Single-float	16	16	8	8	8	8	Single-float	16	8
Weight Bits	Single-float	16	8	Single-float	16	8	8	8	8	8 or 4	Single-float	16	8 or 4
Data Precision	N/A	Dynamic	Dynamic	N/A	2^{-2}	2^{-2}	Not available	2^{-5} or 2^{-1}	Dynamic	Dynamic	N/A	Dynamic	Dynamic
Weight Precision	N/A	Dynamic	Dynamic	N/A	2^{-15}	2^{-7}	Not available	2^{-7}	Dynamic	Dynamic	N/A	Dynamic	Dynamic
Top 1 Accuracy	53.90%	53.90%	53.02%	68.10%	68.02%	62.26%	Not available	28.24%	66.58%	66.96%	68.02%	64.64%	64.14%
Top 5 Accuracy	77.70%	77.12%	76.64%	88.00%	87.94%	85.18%	Not available	49.66%	87.38%	87.60%	87.96%	86.66%	86.30%

[1] The weight bits "8 or 4" in Exp10 and Exp13 means 8 bits for CONV layers and 4 bits for FC layers.
[2] The data precision "2^{-5} or 2^{-1}" in Exp8 means 2^{-5} for feature maps between CONV layers and 2^{-1} for feature maps between FC layers.

Summary

- Performance driven => Computation complexity => Energy efficiency
- FPGA design and metrics
- CNN with basic optimization methods
- Quantization / Weight reduction

- Data allocation
- Data paths
  - #pragma hls
  - #pragma sds

Conclusion

- ***What has not been mentioned?***
  - Quantization – layer-wise adjustment leads to ***accumulated error***?
  - ***Convolution's parallel algorithm*** – which depends on ***data reuse*** and what else
  - ***Network-level optimization*** – e.g. Scheduling (inter-layer parallelism), which highly relates to resource allocation
  - Out of quantization & weight reduction
  - Large NN architecture
  - …

- ***Next…***
  - Small NN to PYNQ-Z2
  - Parallel algorithm & current optimization methods
  - Resource allocation

  - Larger NN
  - 2018-2019 works

From Vivado-HLS to PYNQ-Z2

Workflow - overview

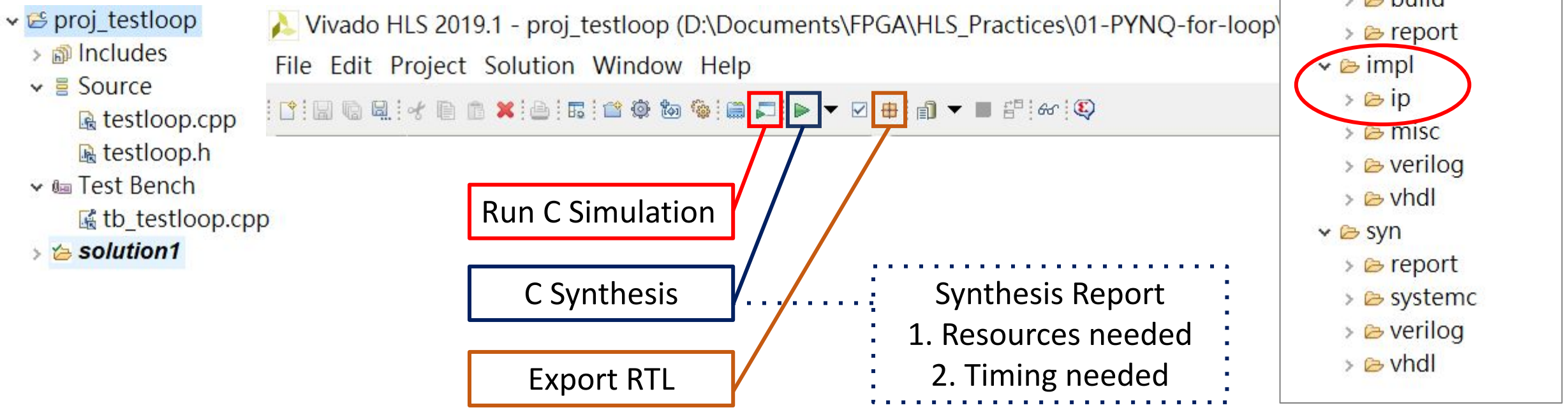
HLS (C/C++) >> IP (RTL)

IP + Zynq PS >> Overlay files (.bit & .tcl)

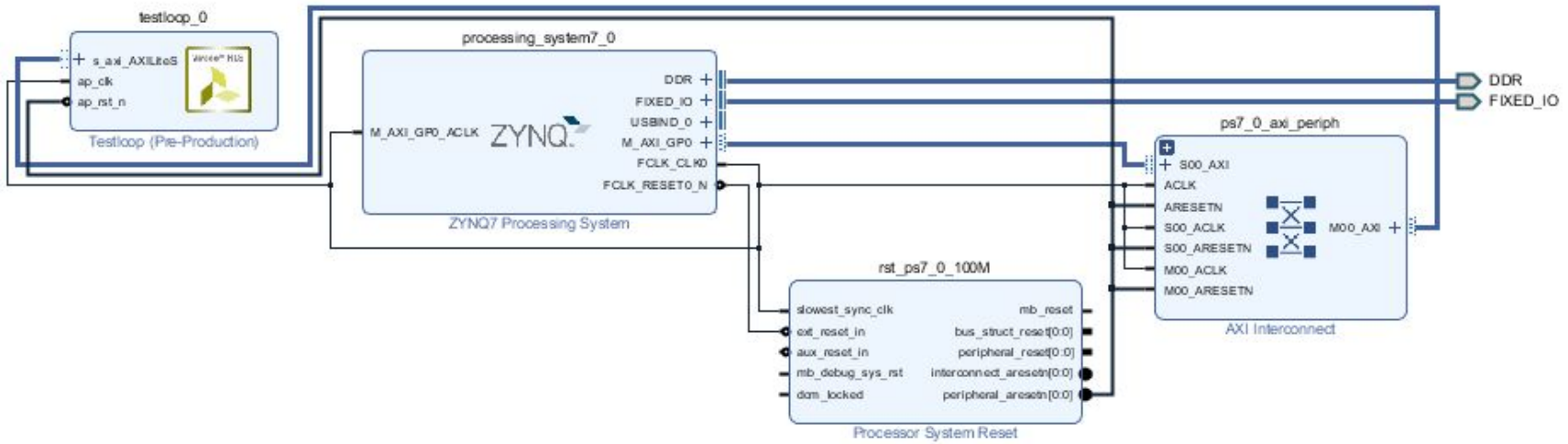
Overlay files (.bit) >> Python API

Workflow – I. Vivado HLS

HLS (C/C++) >> IP (RTL)



Workflow – II. Vivado



Create Block Design

1. Add in ZYNQ7 PS (Processing System)
2. Import HLS project (IP)

Run Connection Automation

Validate Design

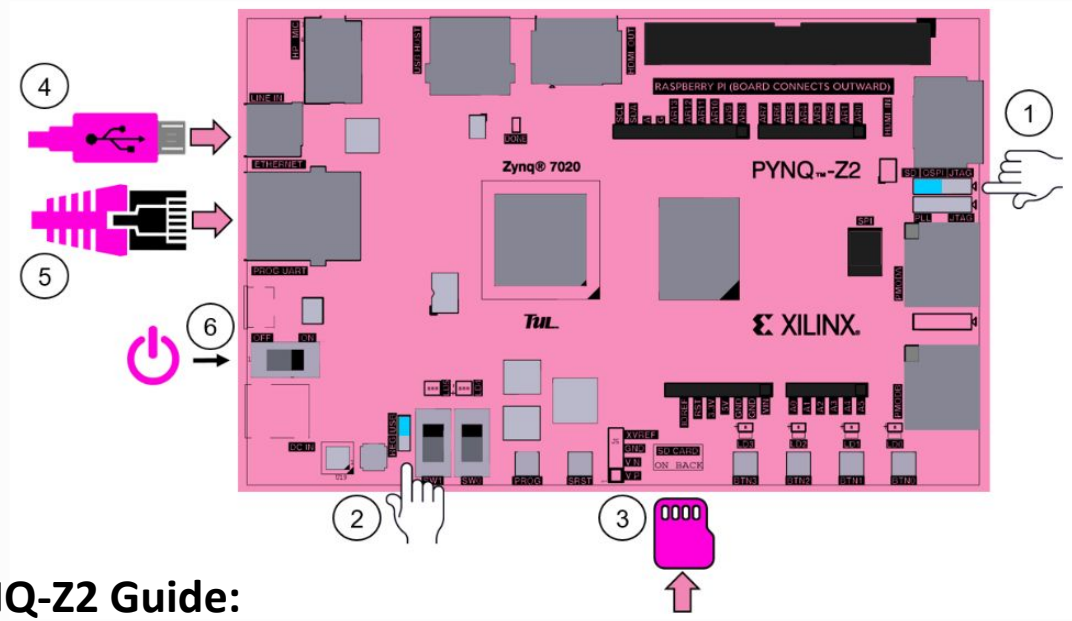
Create HDL Wrapper

Generate bitstream => .bit & .tcl

Workflow – III. Overlay

Overlay files (.bit) >> Python API

Board Setup



192.168.2.99:9090

```
In [1]: from pynq import Overlay
        overlay = Overlay("PATH/loop.bit")

In [2]: overlay?
```

```
IP Blocks
----------
testloop_0           : pynq.overlay.DefaultIP
```

PYNQ-Z2 Guide:
https://pynq.readthedocs.io/en/v2.4/getting_started/pynq_z2_setup.html

Demo – PYNQ-BNN

PYNQ-BNN

- Work - slide p.10

5. Open image to be classified

Download a JPEG image of a deer and place it in a valid directory. The image can then be loaded and displayed through the notebook.

In [12]:
```
from PIL import Image
import numpy as np

im = Image.open('/home/xilinx/jupyter_notebooks/BNN-PYNQ-master/notebooks/pictures/dog2.jpg') # snoopy
# Image.open('/home/xilinx/jupyter_notebooks/BNN-PYNQ-master/notebooks/pictures/deer.jpg')
im
```

Out[12]:

PYNQ-BNN (Cont.)

4. Launching BNN in hardware

The image is passed into the PL and the inference is performed. The Python API takes care of resizing the image to the format required by the network (Cifar-10 format) and transferring the image between hardware and software.

```
class_out=hw_classifier.classify_image(im)
print("Class number: {0}".format(class_out))
print("Class name: {0}".format(hw_classifier.class_name(class_out)))
```

```
Inference took 1581.00 microseconds
Classification rate: 632.51 images per second
Class number: 5
Class name: Dog
```

1/1000x

5. Launching BNN in software

As a comparison, the same image can be classified using a software implementation of the algorithm.

```
class_out = sw_classifier.classify_image(im)
print("Class number: {0}".format(class_out))
print("Class name: {0}".format(sw_classifier.class_name(class_out)))
```

```
Inference took 1587377.00 microseconds
Classification rate: 0.63 images per second
Class number: 5
Class name: Dog
```

Appendix A. – HLS Pragmas

Table 1. Vivado HLS Pragmas by Type

Type	Attributes
Kernel Optimization	• pragma HLS allocation • pragma HLS clock • pragma HLS expression_balance • pragma HLS latency • pragma HLS reset • pragma HLS resource • pragma HLS top
Function Inlining	• pragma HLS inline • pragma HLS function_instantiate
Interface Synthesis	• pragma HLS interface • pragma HLS protocol
Task-level Pipeline	• pragma HLS dataflow • pragma HLS stream

Source: https://www.xilinx.com/html_docs/xilinx2017_2/sdaccel_doc/topics/pragmas/concept-Intro_to_HLS_pragmas.html

Appendix B. – SDS Pragmas

Table 1. SDS Pragmas by Type

Type	Pragmas
Data Access Patterns	• pragma SDS data access_pattern
Data Transfer Size	• pragma SDS data copy • pragma SDS data zero_copy
Memory Attributes	• pragma SDS data mem_attribute
Data Mover Type	• pragma SDS data data_mover
SDSoC Platform Interfaces to External Memory	• pragma SDS data sys_port

Source: https://www.xilinx.com/html_docs/xilinx2017_2/sdaccel_doc/topics/pragmas/concept-Intro_to_SDS_pragmas.html

Reference

[1] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. 2017. A survey of FPGA based neural network accelerator. ACM Trans. Reconfig. Technol. Syst. 9, 4 (2017).

[2] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. Proceedings of the IEEE, 105(12):2295–2329, 2017.

[3] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. Going deeper with embedded fpga platform for convolutional neural network. In ACM International Symposium on FPGA, 2016.

[4] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks", Proc. FPGA, pp. 161-170, 2015.

Reference

[5] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In ACM International Symposium on FPGA, 2016.

[6] Junsong Wang, Qiuwen Lou, Xiaofan Zhang, Chao Zhu, Yonghua Lin, and Deming Chen. 2018. Design Flow of Accelerating Hybrid Extremely Low Bit-width Neural Network in Embedded FPGA. arXiv preprint arXiv:1808.04311(2018).

[7] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. 2017. Angel-Eye: A Complete Design Flow for Mapping CNN onto Embedded FPGA. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2017).

Reference

[8] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. 2015. Compressing neural networks with the hashing trick. In International Conference on Machine Learning. 2285–2294

[9] Song Han, Huizi Mao, and William J Dally. 2016. Deep compression: Compressing deep neural networks with pruning, trained quantization and human coding. ICLR.

[10] Xiangyu Zhang, Jianhua Zou, Xiang Ming, Kaiming He, and Jian Sun. 2015. Efficient and accurate approximations of nonlinear convolutional networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.1984–1992