MSoC Final Project

January 22, 2021

Reconstruction of Compressively Sensed Diagnostic Images Using HLS Solution

Presenter: Ting-Yang Chen, Yu-Cheng Lin, I-Hsuan Liu

Team#: 1

Graduate Institute of Electronics Engineering, National Taiwan University

https://github.com/linkingmon/ReconNet-hls

https://github.com/tingyungchen/ReconNet

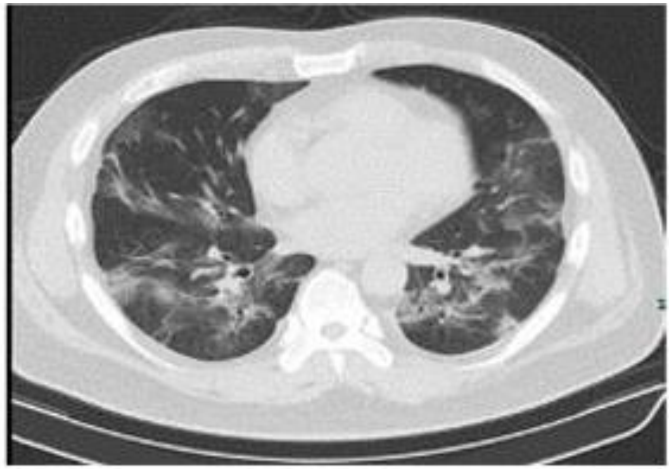
Outline

- Introduction
  - CT Images
  - Compressive sensing
  - ReconNet
- Software simulation
- HLS implementation
- PYNQ implementation
- Summary & Future Work

Diagnostic Images

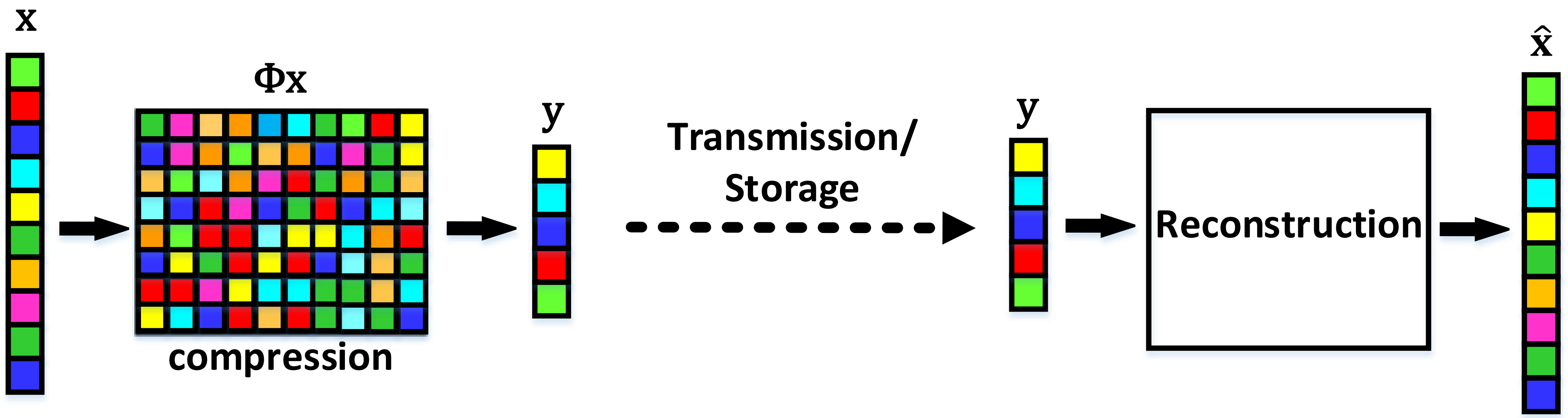
- **Diagnostic images nowadays**
  – CT, radiography and MRI images
  – Poor image quality produced in radiographic examinations
  – Much radiation exposure to patients through repeated radio graphic examinations, loss of diagnostic information

- **Solutions**
  – Compressive sensing to reconstruct images from underdetermined linear systems
  – Reducing patient dose and loss of information



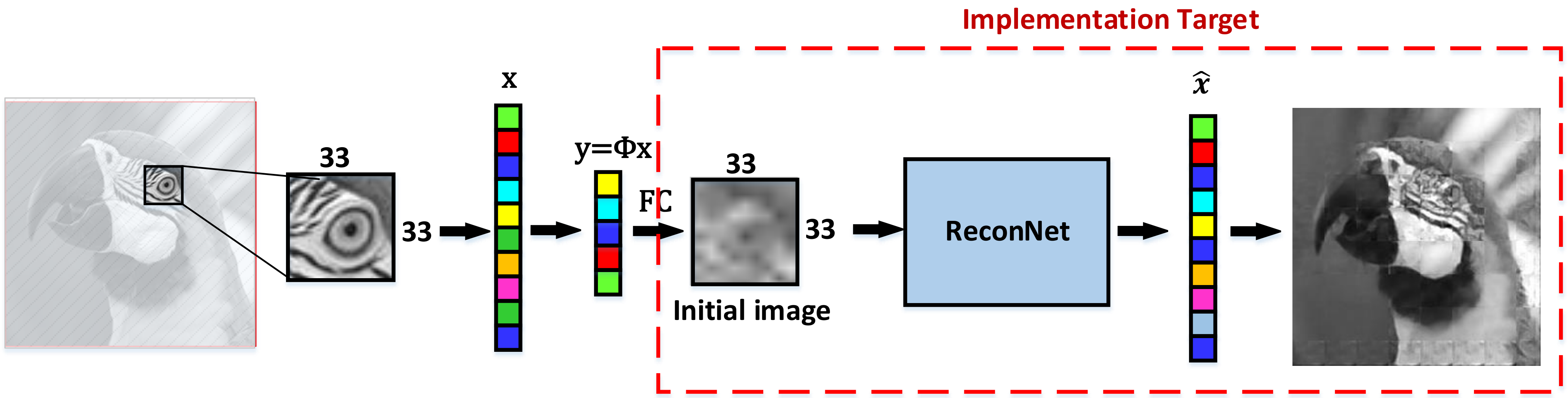
Compressive sensing

- **Compressive sensing**
  - The matrix Φ reduces the dimension of the signal x
- **Reconstruction**
  - Underdetermined linear systems
  - Solved by algorithms or neural network



Reconstruction with ReconNet

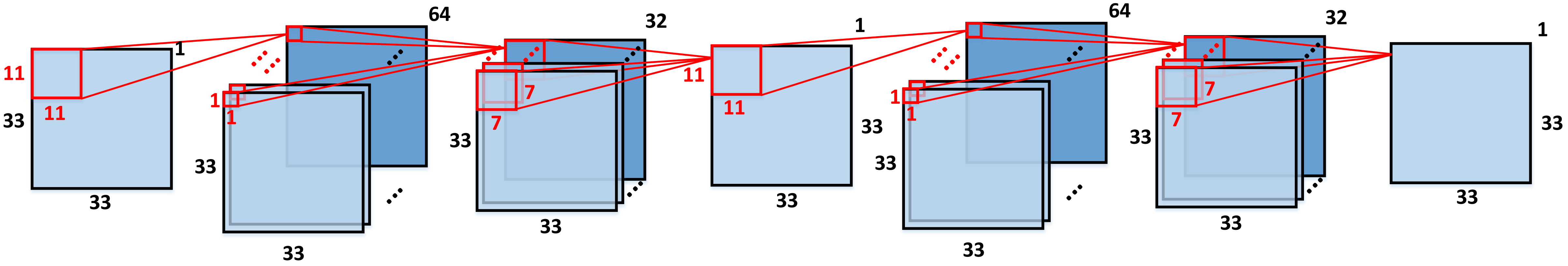
- **CS measurements**
  - Measurement matrix Φ is a random Gaussian matrix of appropriate size
- **Initial image estimate**
  - A FC layer maps the CS measurements vector to a 2D array
  - Due to large size, patch-based method is applied
- ReconNet used to solve the image reconstruction problem



[1] S. Lohit, *et al.*, *Trans. Computational Imaging*, 2018.

ReconNet

- **Three convolutional layers**
  - All intermediate feature maps are 33×33
  - ReconNet Unit
    - first convolutional layer - 11×11, generates 64 feature maps
    - second convolutional layer - 1×1, generates 32 feature maps
    - third convolutional layer - 7×7, generates the output block
  - ReLU is followed after each convolutional layer



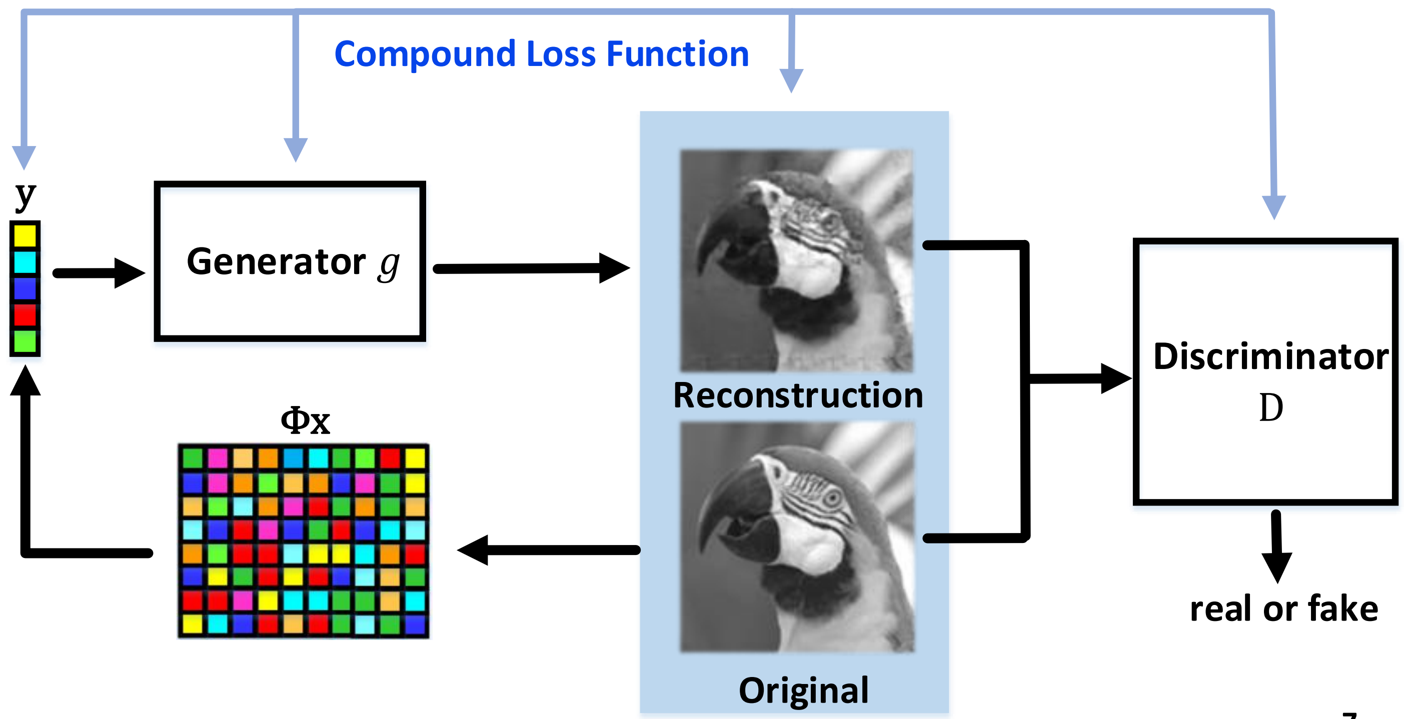
Loss Function of ReconNet

- **Euclidean Loss**
  - mini-batch gradient descent with backpropagation

$$L(\Theta) = \frac{1}{B}\sum_{i=1}^{B}\left\| f(\mathbf{y}_i, \Theta) - \mathbf{x}_i \right\|^2$$

- **Euclidean + Adversarial Loss**
  - ReconNet units acts as generator g
  - Discriminator D outputs the probability of the input being a real image block
  - Parameters of g & D update in alternating fashion

$$L_g = \frac{\lambda_{rec}}{B}\sum_{i=1}^{B}\left\| g(\mathbf{y}_i) - \mathbf{x}_i \right\|^2 + \frac{\lambda_{adv}}{B}\sum_{i=1}^{B} L_{CE}(D(g(\mathbf{y}_i)),1)$$

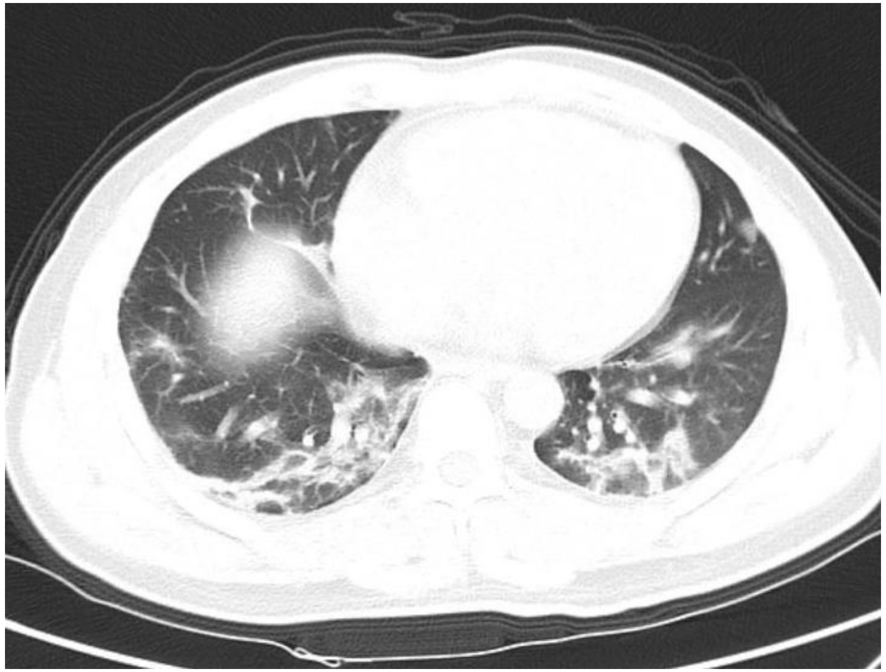


Outline

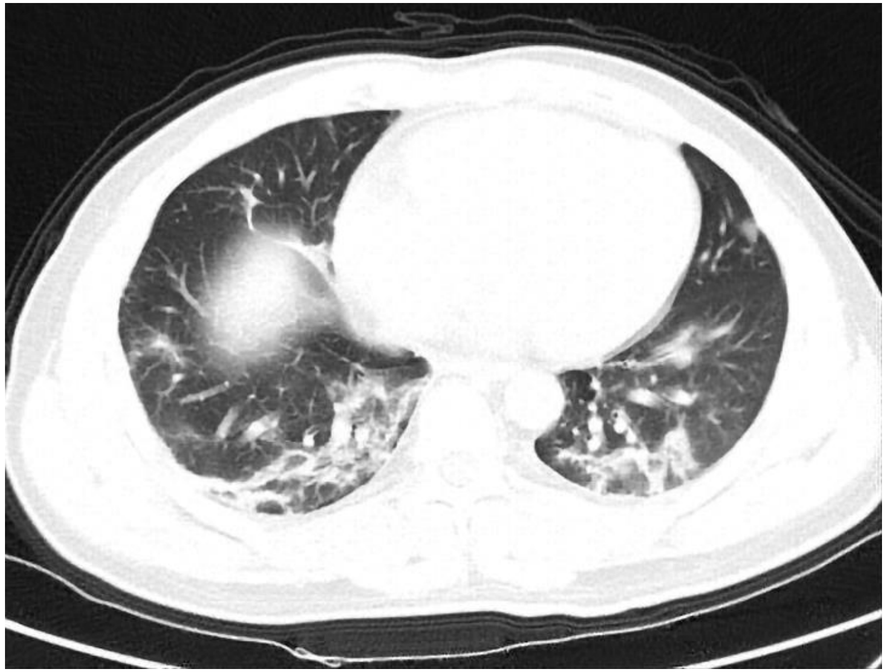
- Introduction
  - CT Images
  - Compressive sensing
  - ReconNet
- Software simulation
- HLS implementation
- PYNQ implementation
- Summary & Future Work

ReconNet Reconstruction (4x Reduction)

- Euclidean + adversarial with Learned Φ, trained with Adam optimizer
  - MR = 0.25, mean PSNR is 30.43 (total 346)



Original



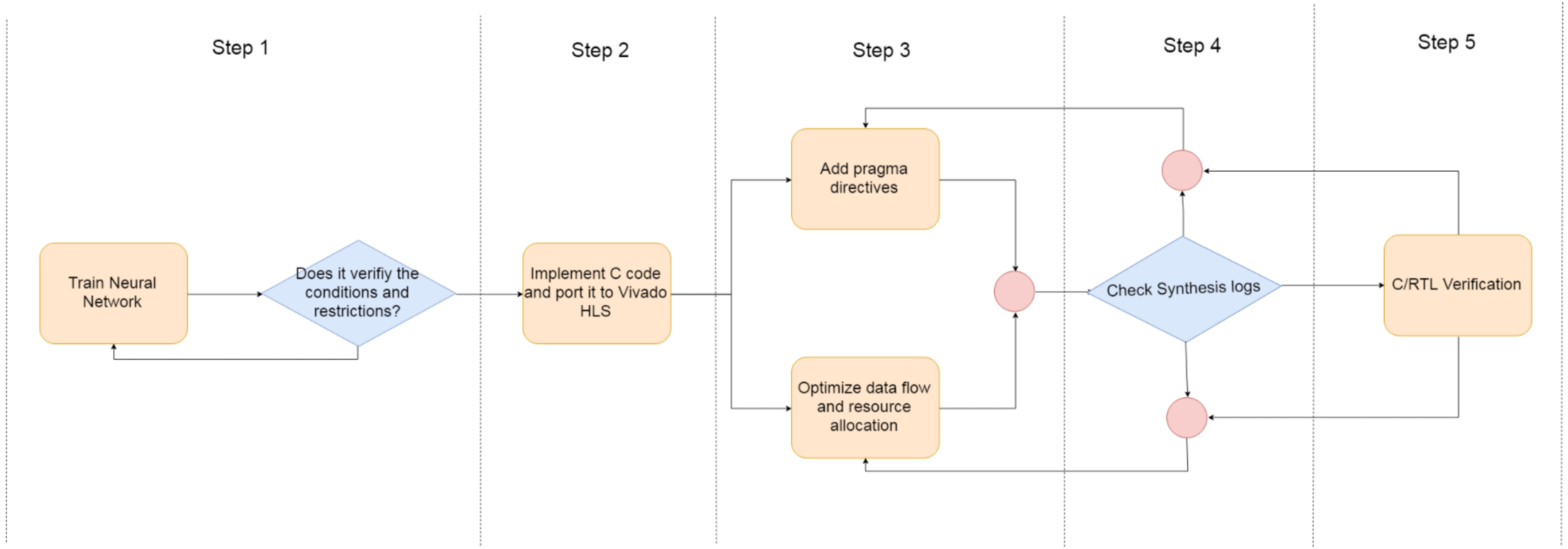
Reconstructed

Outline

- Introduction
  - Compressive sensing
  - ReconNet
- Software simulation
- HLS implementation
  - HLS Workflow
  - Model quantization
  - Dataflow, streaming and line-buffer
  - Comparison Table
- PYNQ implementation
- Summary & Future Work

Workflow

- Cosim cost several days, so we skip this step
- All the works are our original work



Model Quantization (1/2)

- Fixed point arithmetic uses significantly fewer resources for basic operations and less memory usage
- However, range of model weights and feature map varies from layers (Even 24 bit fixed point get poor results)
- We use float-16 (1 signed, 5 exp, 10 mantissa) for implementation (Software verified)

```
#include "hls_half.h"
typedef half data_type;
```

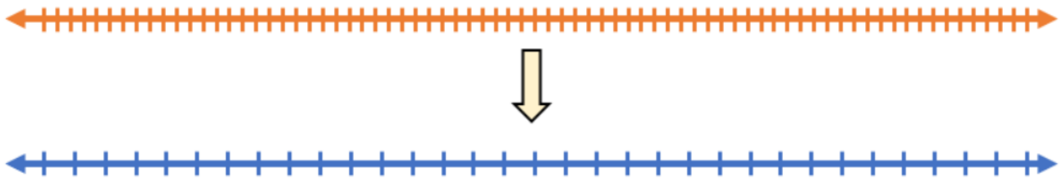
Float32

Float16

Original

Model Quantization (2/2)

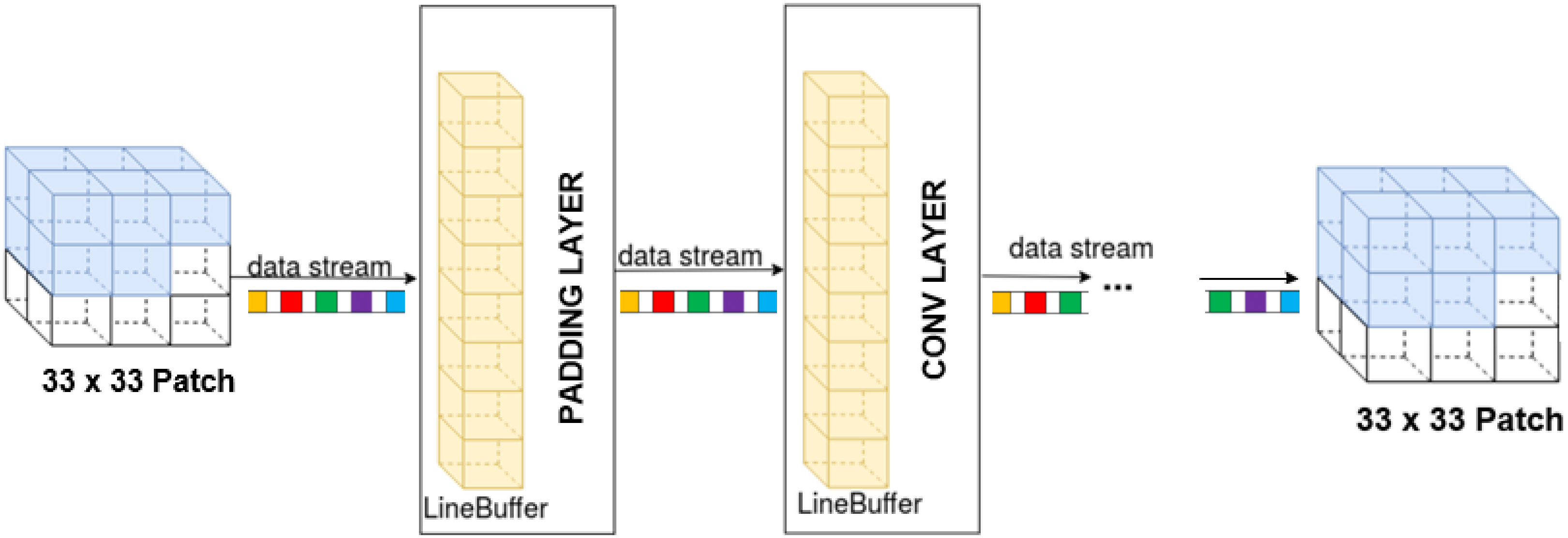
- **Post-training quantization**
  - Reduce ReconNet model size
  - Reduce hardware resources
- **Simple linear equation applied**
  - r is the real value (usually *float32*)
  - q is its quantized representation as a B-bit integer(*uint8*, *uint32*, etc.)
  - S (*float32*) and z (*uint*) are the factors by which we scaling and shifting
  - z is the quantized zero-point



$$r = S(q - z)$$

Dataflow pipelining

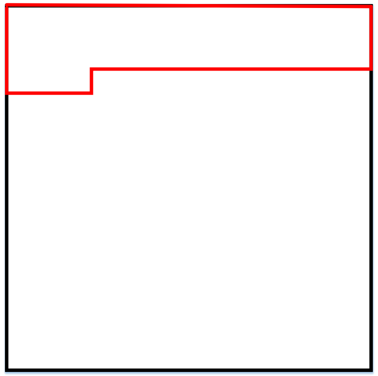
- There are two main functions: Zero-padding & Convolution 2d
- Dataflow pipelining the execution of convolutional
  – Don’t need to wait until all convolution completes
  – Data buffers (FIFO or PingPong buffer) were added in each layer
  – We choose FIFO for higher throughput and lower resources



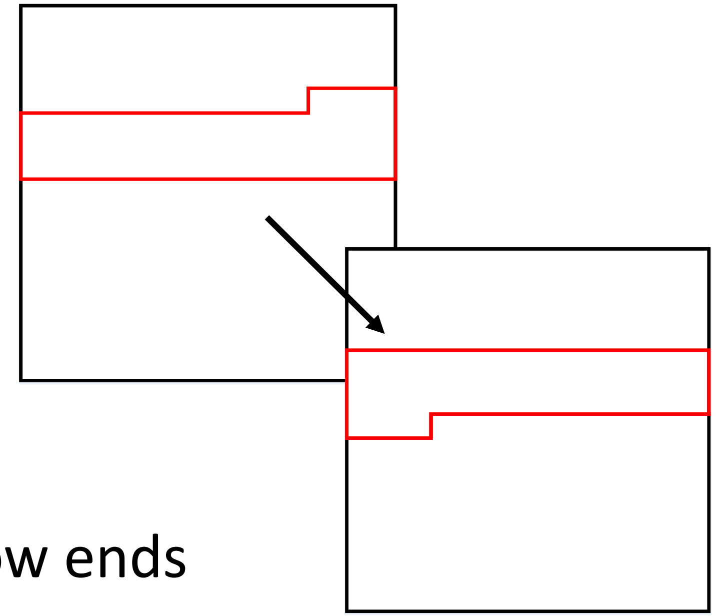
Line buffer for convolution operation

```
for(i = 0 ; i < INPUT_SIZE * (KERNEL_SIZE -1) + KERNEL_SIZE ; i++) {
    for(int p = 0 ; p < INPUT_CHANNELS ;  p++){
        check_read(in, placeholder);
        conv_buff.shift_up(p);
        conv_buff.insert_top(placeholder, p);
    }
}
for (i = 0 ; i < (INPUT_SIZE - KERNEL_SIZE + 1); i += STRIDE)
    for (j = 0 ; j < (INPUT_SIZE - KERNEL_SIZE + 1); j += STRIDE){
        for (filter = 0 ; filter < FILTERS ; filter++) {

            for (row_offset = 0 ; row_offset < KERNEL_SIZE; row_offset++)
                for (col_offset = 0 ; col_offset < KERNEL_SIZE; col_offset++)
                    for (channel_offset = 0 ; channel_offset < INPUT_CHANNELS ; channel_offset++) {
                        #pragma HLS pipeline
                        int t1, t2;
                        static data_type val1, val2;
                        t1 = row_offset * INPUT_SIZE;
                        t2 = col_offset;
                        val1 = conv_buff.getval(t1 + t2, channel_offset);
                        val2 = weight[row_offset][col_offset][channel_offset][filter];
                        sum += val1 * val2;
                    }
            out << relu(sum/* + bias[filter]*/);
        }

        if ((j + STRIDE < (INPUT_SIZE - KERNEL_SIZE + 1))) {
            for (int p = 0 ; p < INPUT_CHANNELS ; p++){
                check_read(in, placeholder);
                conv_buff.shift_up(p);
                conv_buff.insert_top(placeholder, p);
            }
        }
        else if ((i + STRIDE < (INPUT_SIZE - KERNEL_SIZE + 1)) && (j + STRIDE >= (INPUT_SIZE - KERNEL_SIZE + 1))){
            for (int k = 0 ; k < KERNEL_SIZE ; k++){
                for (int p = 0 ; p < INPUT_CHANNELS ; p++){
                    check_read(in, placeholder);
                    conv_buff.shift_up(p);
                    conv_buff.insert_top(placeholder, p);
                }
            }
    }
```

Read the initial data into line buffer



Do filtering (Pipeline in the inner loop)



Pre-read data into line buffer if row ends

2D Line buffer Library

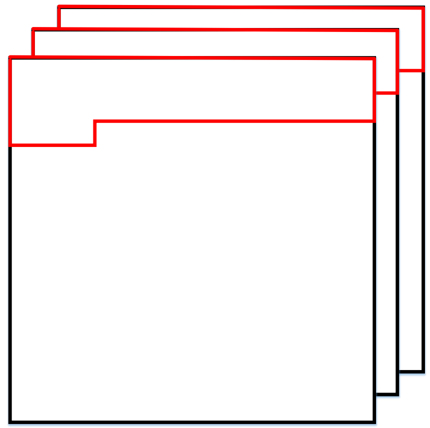
- If the line buffer is specify as 1D (**7712 x 1**), the array partition number of layer2 (32 channel kernel 7)exceeds 1024 (error!)

```
hls::LineBuffer<INPUT_SIZE * INPUT_CHANNELS * (KERNEL_SIZE -1) + KERNEL_SIZE * INPUT_CHANNELS, 1, data_type> conv_buff;
```

- We modified the Linebuffer to 2D (**241 x 32**), then the array partition number satisfies, however, the channel will be non-parallizable

```
hls::LineBuffer<INPUT_SIZE * (KERNEL_SIZE -1) + KERNEL_SIZE , INPUT_CHANNELS, data_type> conv_buff;
```

```
template<int ROWS, int COLS, typename T>
class LineBuffer<ROWS, COLS, T, 0> {
public:
    LineBuffer() {
#pragma HLS array_partition variable=val dim=1 complete
#pragma HLS dependence variable=val inter false
#pragma HLS dependence variable=val intra false
    };
    /* LineBuffer main APIs */
    void shift_pixels_up(int col);
    void shift_pixels_down(int col);
    void insert_bottom_row(T value, int col);
    void insert_top_row(T value, int col);
    void get_col(T value[ROWS], int col);
    T& getval(int row, int col);
    T& operator ()(int row, int col);
```



2D Line buffer Resource usage

- The HLS results of 32 bit data type is as follows

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	50	-
FIFO	50	-	1275	1175	-
Instance	491	23	84683	41123	0
Memory	44	-	0	0	0
Multiplexer	-	-	-	815	-
Register	-	-	51	-	-
Total	585	23	86009	43163	0
Available	280	220	106400	53200	0
Utilization (%)	208	10	80	81	0

* Memory:

Memory	Module	BRAM_18K	FF	LUT	URAM	Words	Bits	Banks	W*Bits*Banks
conv_buff_val_0_U	conv_layer_1_convjbC	1	0	0	0	32	32	1	1024
conv_buff_val_1_U	conv_layer_1_convjbC	1	0	0	0	32	32	1	1024
conv_buff_val_2_U	conv_layer_1_convjbC	1	0	0	0	32	32	1	1024
⋮	⋮								⋮
conv_buff_val_237_U	conv_layer_1_convjbC	1	0	0	0	32	32	1	1024
conv_buff_val_238_U	conv_layer_1_convjbC	1	0	0	0	32	32	1	1024
conv_buff_val_239_U	conv_layer_1_convjbC	1	0	0	0	32	32	1	1024
conv_buff_val_240_U	conv_layer_1_convjbC	1	0	0	0	32	32	1	1024
kernel3_weight_0_U	conv_layer_1_kernibs	4	0	0	0	1568	32	1	50176
Total		245	0	0	0	9280	7744	242	296960

- By changing the datatype to half (float16), the non-parallel part is then realized by FF.

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	50	-
FIFO	25	-	875	975	-
Instance	5	18	55162	40192	0
Memory	22	-	64	16	0
Multiplexer	-	-	-	815	-
Register	-	-	51	-	-
Total	52	18	56152	42048	0
Available	280	220	106400	53200	0
Utilization (%)	18	8	52	79	0

Memory	Module	BRAM_18K	FF	LUT	URAM	Words	Bits	Banks	W*Bits*Banks
conv_buff_val_0_U	conv_layer_1_convjbC	0	32	8	0	32	16	1	512
conv_buff_val_1_U	conv_layer_1_convjbC	0	32	8	0	32	16	1	512
conv_buff_val_2_U	conv_layer_1_convjbC	0	32	8	0	32	16	1	512
⋮	⋮								⋮
conv_buff_val_239_U	conv_layer_1_convjbC	0	32	8	0	32	16	1	512
conv_buff_val_240_U	conv_layer_1_convjbC	0	32	8	0	32	16	1	512
kernel3_weight_0_U	conv_layer_1_kernibs	2	0	0	0	1568	16	1	25088
Total		2	7712	1928	0	9280	3872	242	148480

Comparison table

- Final resources

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	50	-
FIFO	25	-	875	975	-
Instance	5	18	55162	40192	0
Memory	22	-	64	16	0
Multiplexer	-	-	-	815	-
Register	-	-	51	-	-
Total	52	18	56152	42048	0
Available	280	220	106400	53200	0
Utilization (%)	18	8	52	79	0

- Latency

* Summary:

Latency (cycles)		Latency (absolute)		Interval		Pipeline
min	max	min	max	min	max	Type
147182011	147193990	0.736 sec	0.736 sec	146086593	146098572	dataflow

Clock	Target	Estimated	Uncertainty
ap_clk	5.00 ns	4.322 ns	0.62 ns

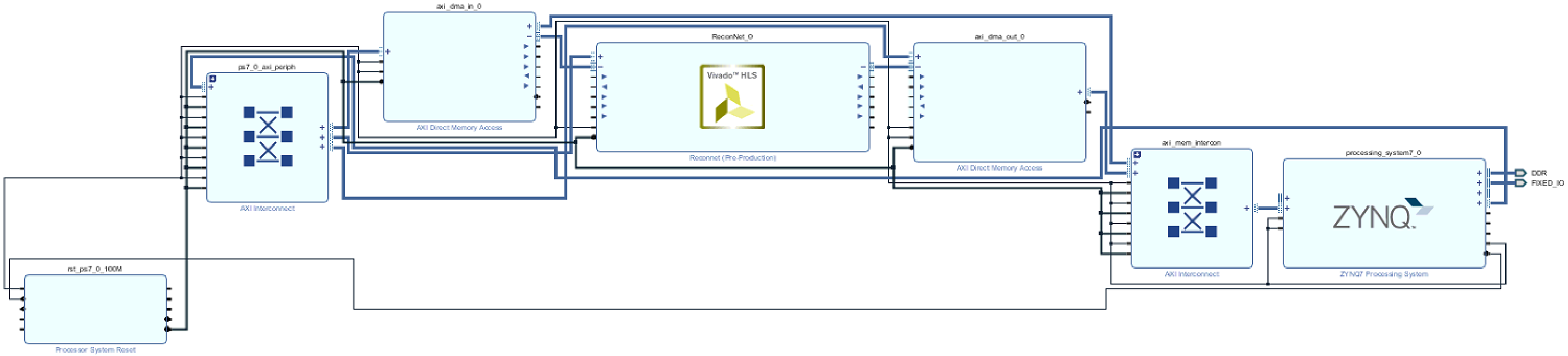
Outline

- Introduction
  - Compressive sensing
  - ReconNet
- Software implementation
- HLS implementation
- PYNQ implementation
  - Top level architecture
  - Host program
  - Demo
- Summary & Future Work

Top level architecture

- Apply AXI stream as interface

```
#pragma HLS INTERFACE axis port=AXI_video_stream_in bundle=VIDEO_IN
#pragma HLS INTERFACE axis port=AXI_video_stream_out bundle=VIDEO_OUT
#pragma HLS INTERFACE s_axilite port=return bundle=CONTROL
```



Host program

```python
ol = Overlay("/home/xilinx/IPBitFile/yclin/ReconNet2.bit")
ip_ReconNet = ol.ReconNet_0
ipDMAIn = ol.axi_dma_in_0
ipDMAOut = ol.axi_dma_out_0
```

Read bit stream

```python
n_row = 33
n_col = 33
# n_col = 256+144
numSamples = n_row*n_col
image = cv2.imread('barbara.tif',cv2.IMREAD_UNCHANGED)
print("Shape of image_reshape", image.shape)
```

Read input feature map

```python
xlnk = Xlnk()
inBuffer0 = xlnk.cma_array(shape=(numSamples,), dtype=np.single)
outBuffer0 = xlnk.cma_array(shape=(numSamples,), dtype=np.single)


for i in range(n_row):
    for j in range(n_col):
        inBuffer0[i*n_col+j] = image[i][j]
```

Store input feature map to Xlnk

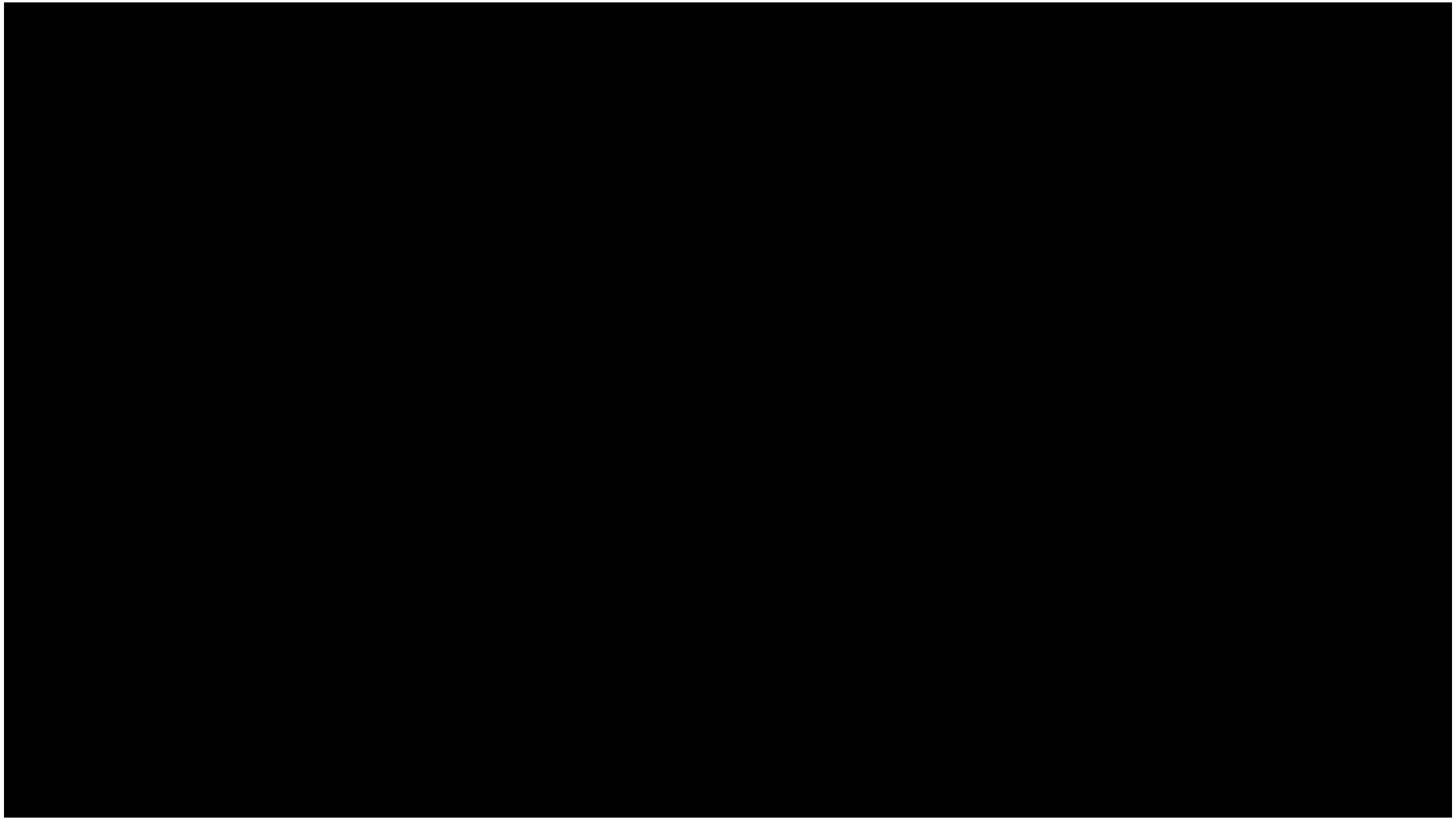
```python
ip_ReconNet.write(0x00, 0x01)
ipDMAIn.sendchannel.transfer(inBuffer0)
ipDMAOut.recvchannel.transfer(outBuffer0)
ipDMAIn.sendchannel.wait()
ipDMAOut.recvchannel.wait()
```

Start Kernel and wait till DMA_out receive all values

```python
print("==========================")
print("Exit process")
```

Demo

- Reconstruction of 256x256 image (4x transmission reduction)



Final Performance

- Performance
  - Reach a reconstruction of PSNR over 30(dB) on CT image (Human cannot tell whether the image is real or fake by eye)

- Hardware implemetation
  - The network require memory to store the weight (Weight stationary)
  - The image is streaming into and out of the submodule (streaming)
  - Utilize half data type to reduce resources while maintaining performance

Summary

- Algorithms
  - Construct high resolution image by ReconNet
  - Slice full image into patches to reduce model size
  - Model Quantization make the solution more hardware efficient
- HLS
  - Dataflow pipelining with data streaming
  - Utilize half data type to reduce resources while maintaining performance
  - Loop pipelining to speed up
- FPGA implementation

Future Work

- Real time high resolution video streaming (v.s. H.264)
- Motion vector aided bypassing
  - Find Motion vector by motion estimation algorithms on low resolution picture and predict on high resolution picture
  - Enhance frame rate
- Make the model more scalable

Reference

[1] S. Lohit, K. Kulkarni, R. Kerviche, P. Turaga and A. Ashok, "Convolutional Neural Networks for Noniterative Reconstruction of Compressively Sensed Images," in *IEEE Transactions on Computational Imaging*, vol. 4, no. 3, pp. 326-340, Sept. 2018