MSOC

Dec. 11, 2020

Neural Network Acceleration Using HLS Solution

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

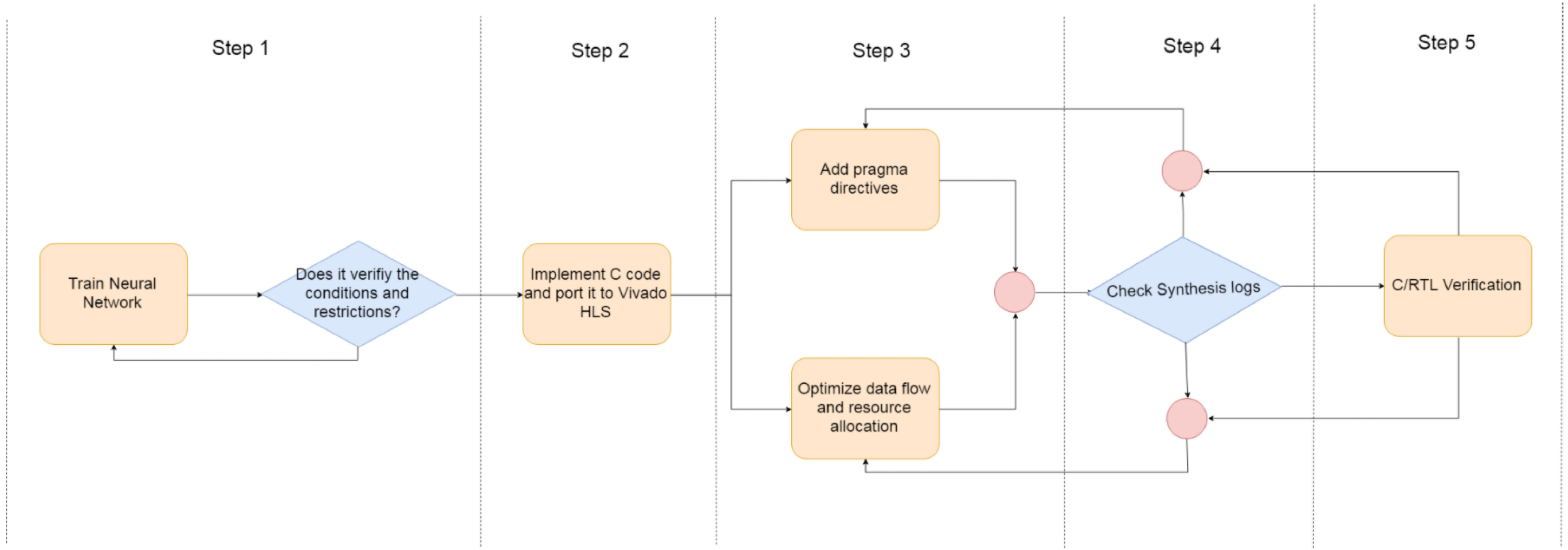
Team#: 1

Outline

- Introduction
- Design Optimization
- Challenges and Solutions
- HLS Debugging Insights
- Vitis acceleration on U50

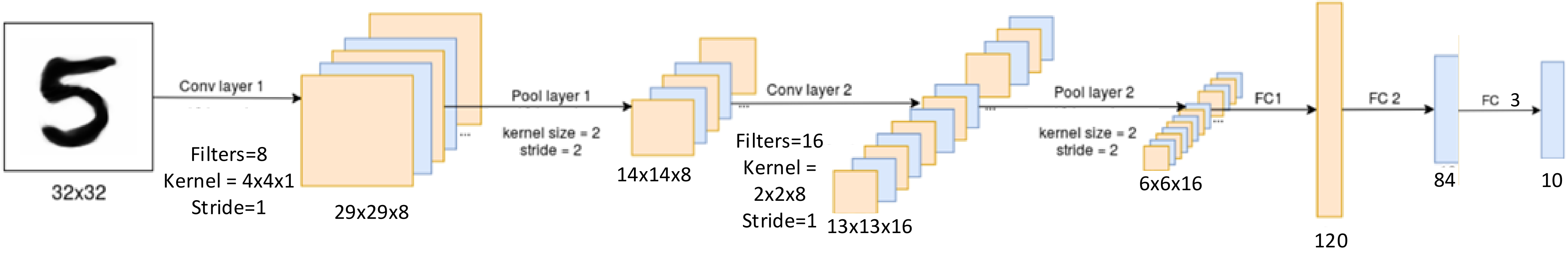
Introduction

- Implement a Deep Learning algorithm for handwritten digit recognition using HLS
- Workflow overview



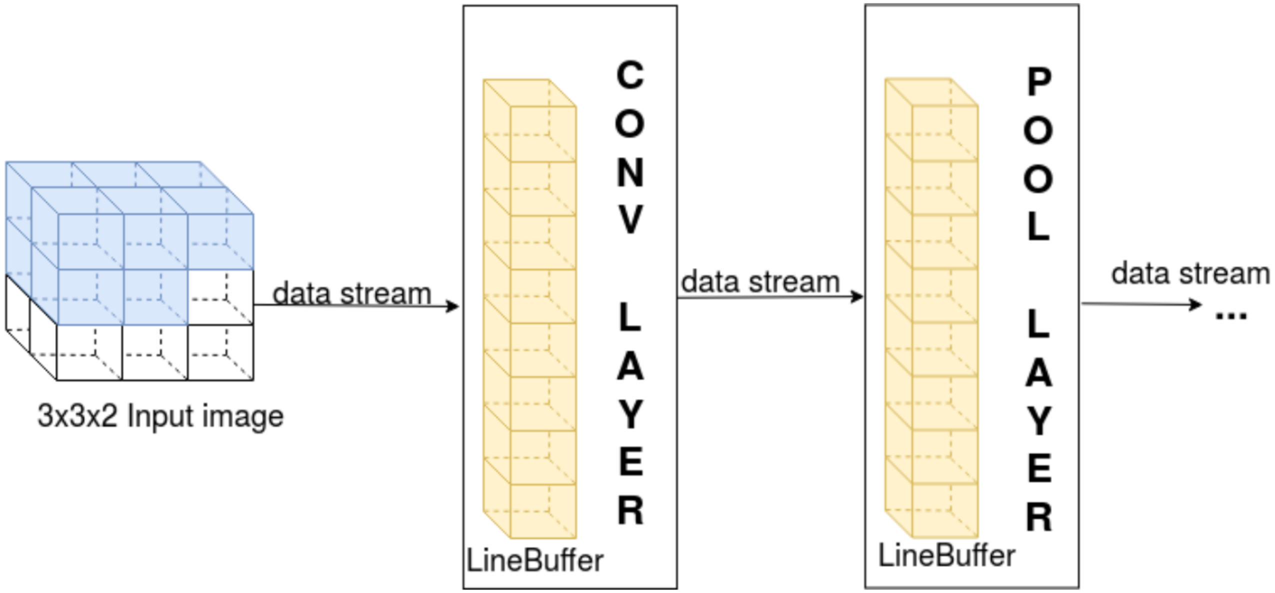
Neural Network architecture

- Two Convolution Layers
- Two Pooling Layers
- Two Fully-connected Layers



Dataflow pipelining

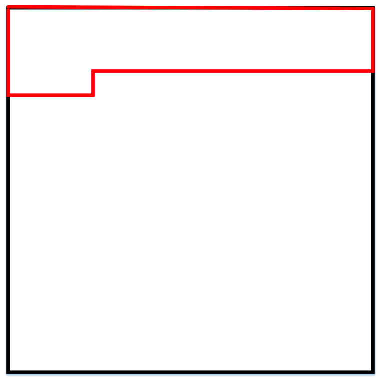
- There are three main functions, and they are called sequentially
  - Convolution, Pooling, Fully-connected
- Parallelizing the execution of convolutional and pooling layers
  - Don’t need to wait until convolution completes
  - Data streams were added between the functions
  - PINGPONG buffers were added in each layer



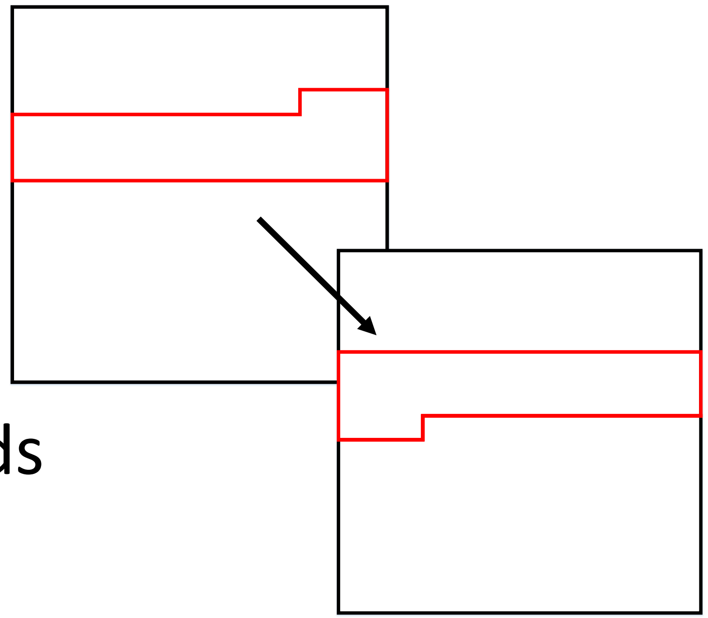
Conv. Layer

```
for (i = 0; i < CONV1_BUFFER_SIZE; i++) {
    check_read(in, placeholder);
    conv_buff.shift_up(0);
    conv_buff.insert_top(placeholder, 0);
}

for (i = 0; i < (IMAGE_SIZE - CONV1_KERNEL_SIZE + 1); i += CONV1_STRIDE){
    for (j = 0; j < (IMAGE_SIZE - CONV1_KERNEL_SIZE + 1); j += CONV1_STRIDE) {
        #pragma HLS PIPELINE II=20
        for (filter = 0; filter < CONV1_FILTERS; filter++) {
            // #pragma HLS UNROLL complete
            sum = 0;
            for (row_offset = 0; row_offset < CONV1_KERNEL_SIZE; row_offset++){
                #pragma HLS UNROLL complete
                for (col_offset = 0; col_offset < CONV1_KERNEL_SIZE; col_offset++){
                    #pragma HLS UNROLL complete
                    for (channel_offset = 0; channel_offset < CONV1_CHANNELS; channel_offset++) {
                        #pragma HLS UNROLL complete
                        int t1, t2;
                        static float24_t val1, val2;
                        t1 = row_offset * IMAGE_SIZE * IMAGE_CHANNELS;
                        t2 = col_offset * IMAGE_CHANNELS;
                        val1 = conv_buff.getval(t1 + t2 + channel_offset, 0);
                        val2 = weight[row_offset][col_offset][channel_offset][filter];
                        sum += val1 * val2;
                    }
                }
            }
            float24_t out_val = sum + bias[filter];
            out_val = (out_val > float24_t(0)) ? out_val : float24_t(0);
            out <<out_val;

        if ((j + CONV1_STRIDE < (IMAGE_SIZE - CONV1_KERNEL_SIZE + 1))) {
            for (int p = 0 ; p < IMAGE_CHANNELS; p++){
                // #pragma HLS UNROLL complete
                check_read(in, placeholder);
                conv_buff.shift_up(0);
                conv_buff.insert_top(placeholder, 0);
            }

        else if ((i + CONV1_STRIDE < (IMAGE_SIZE - CONV1_KERNEL_SIZE + 1))
                && (j + CONV1_STRIDE >= (IMAGE_SIZE - CONV1_KERNEL_SIZE + 1))) {
            for (int p = 0; p < CONV1_KERNEL_SIZE * IMAGE_CHANNELS; p++){
                // #pragma HLS UNROLL complete
                check_read(in, placeholder);
                conv_buff.shift_up(0);
                conv_buff.insert_top(placeholder, 0);
            }

    }
}
```

Read the initial data into line buffer



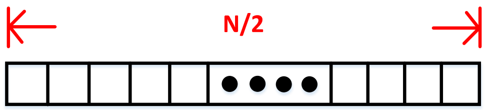
Do filtering

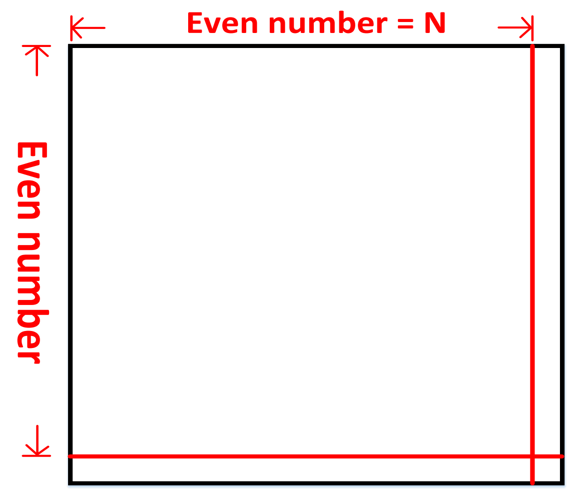


Pre-read data into line buffer if row ends

Pooling layer

```
void pool_layer1(hls::stream<float24_t>& out, hls::stream<float24_t>& in) {

    int i, j, k, l, m;
    float24_t read;
    // hls::LineBuffer<POOL1_BUFFER_SIZE, 1, float24_t> pool_buff;
    float24_t pool_buff[POOL1_BUFFER_SIZE];

    for (i = 0; i < P1_SIZE; i++){
        for (l = 0; l < P1_KERNEL_SIZE; l++) {
            for (j = 0; j < P1_SIZE; j++){
                // #pragma HLS UNROLL complete
                for (m = 0; m < P1_KERNEL_SIZE; m++){
                    // #pragma HLS UNROLL complete
                    for (k = 0 ; k < P1_CHANNELS; k++) {
                        // #pragma HLS UNROLL complete
                        check_read(in, read);
                        if (l == 0 && m == 0)
                            pool_buff[j * P1_CHANNELS + k] = read;

                        else
                            pool_buff[j * P1_CHANNELS + k] =
                                    pool_buff[j * P1_CHANNELS + k] > read ?
                                            pool_buff[j * P1_CHANNELS + k] : read;

                        if (l == (P1_KERNEL_SIZE - 1) && m == (P1_KERNEL_SIZE - 1))
                            out << pool_buff[j * P1_CHANNELS + k];
                    }
                }
            }
            for (int skip = P1_SIZE * P1_STRIDE ; skip < A1_SIZE; skip++){
                // #pragma HLS UNROLL complete
                for (int channel = 0 ; channel < P1_CHANNELS; channel++){
                    // #pragma HLS UNROLL complete
                    check_read(in, read);
                }
            }
        }
    }
    for (int skip_row = P1_SIZE * P1_STRIDE; skip_row < A1_SIZE; skip_row++){
        // #pragma HLS PIPELINE
        for (int skip_col = 0; skip_col < A1_SIZE; skip_col++){
            // #pragma HLS UNROLL complete
            for (int skip_channel = 0; skip_channel < A1_CHANNELS; skip_channel++){
                // #pragma HLS UNROLL complete
                check_read(in, read);
            }
        }
    }
}
```





Compare the input with the data in buffer and store back

Read extra cols

Read extra rows

Fully Connected Layer

```
void fc_layer1(hls::stream<float24_t> &out, hls::stream<float24_t> &in,
        float24_t weight[FC1_WEIGHTS_H][FC1_WEIGHTS_W],
        float24_t bias[FC1_BIAS_SIZE]) {
    float24_t read;
    float24_t output[FC1_ACT_SIZE] = { 0 };

    // #pragma HLS ARRAY_PARTITION variable=weight complete

    check_read(in, read);
    for (int i = 0; i < FC1_WEIGHTS_W; i++)
        output[i] = weight[0][i] * read;

    for (int j = 1; j < FC1_WEIGHTS_H; j++) {
        check_read(in, read);
        for (int i = 0; i < FC1_WEIGHTS_W; i++) {
            #pragma HLS UNROLL complete
            output[i] += weight[j][i] * read;
        }
    }
    for (int i = 0; i < FC1_WEIGHTS_W; i++){
        float24_t out_val = output[i] + bias[i];
        out_val = (out_val > float24_t(0)) ? out_val : float24_t(0);
        out <<out_val;
    }
}
```

Accumulate partial products

Output results

Word length Reduction

- Fixed point operations are less precise, use less hardware and won’t affect the global accuracy of the Neural Network

```
#include "ap_fixed.h"
typedef ap_fixed<WIDTH, INT_WIDTH> fixed_p;
```

- Fixed point arithmetic uses significantly fewer resources for basic operations

- Original design use float width of 32 and int width of 12 for implementation

```
#define EXP_WIDTH   32
#define INT_WIDTH   12
```

- Reduce word length under allowable accuracy loss, e.g. (16,4) is enough

Optimize FIFO size

- Config dataflow command

```
config_dataflow -default_channel fifo -fifo_depth 150
```

- If not specified, it is default fulfill by PINGPONG buffer
- FIFO depth is a tradeoff between hardware resource and latency
- Set FIFO depth to 10

Parallelize and Pipeline

- Convolution operations UNROLL

```
for (filter = 0; filter < CONV1_FILTERS; filter++) {
    // #pragma HLS UNROLL complete
    sum = 0;
    for (row_offset = 0; row_offset < CONV1_KERNEL_SIZE; row_offset++){
        #pragma HLS UNROLL complete
        for (col_offset = 0; col_offset < CONV1_KERNEL_SIZE; col_offset++){
            #pragma HLS UNROLL complete
            for (channel_offset = 0; channel_offset < CONV1_CHANNELS; channel_offset++) {
                #pragma HLS UNROLL complete
                int t1, t2;
                static float24_t val1, val2;
                t1 = row_offset * IMAGE_SIZE * IMAGE_CHANNELS;
                t2 = col_offset * IMAGE_CHANNELS;
                val1 = conv_buff.getval(t1 + t2 + channel_offset, 0);
                val2 = weight[row_offset][col_offset][channel_offset][filter];
                sum += val1 * val2;
```

- Convolution operations PIPELINE

```
for (row_offset = 0; row_offset < CONV1_KERNEL_SIZE; row_offset++){
    // #pragma HLS UNROLL complete
    for (col_offset = 0; col_offset < CONV1_KERNEL_SIZE; col_offset++){
        // #pragma HLS UNROLL complete
        for (channel_offset = 0; channel_offset < CONV1_CHANNELS; channel_offset++) {
            #pragma HLS PIPELINE
            int t1, t2;
            static float24_t val1, val2;
            t1 = row_offset * IMAGE_SIZE * IMAGE_CHANNELS;
            t2 = col_offset * IMAGE_CHANNELS;
            val1 = conv_buff.getval(t1 + t2 + channel_offset, 0);
            val2 = weight[row_offset][col_offset][channel_offset][filter];
            sum += val1 * val2;
```

Comparison

	Original	UNROLL filter operations	Word length Reduction FIFO=10	PIPELINE
Latency	539, 684	256, 786	184, 204	447, 166
BRAM (%)	14	13	8	6
DSP (%)	9	589	265	3
FF (%)	29	59	17	15
LUT (%)	14	113	25	13

HLS Debugging

- Csim
  - Large array size initialization leads to compile error

```
float24_t fc_layer1_weights[576][120] = {0.075783, 0.076267, 0.132234, -0.163968, 0.119929, -0.127198, -0.053471, -0.101884, -0.017365, -0.096139, -0.135449, 0.067226,
0.051957, -0.142047, -0.166744, 0.159165, -0.080644, 0.011216, -0.164717, -0.131069, -0.119039, -0.156155, -0.045792, 0.019286, 0.015486, -0.000664, -0.049182, -0.
148958, -0.076115, -0.068917, -0.049044, 0.036929, -0.134843, -0.097139, 0.005907, 0.029125, 0.022657, 0.157794, 0.070759, 0.017073, 0.143593, -0.079946, -0.162674, -0.
101664, -0.164730, -0.169723, -0.155077, -0.092246, 0.114661, 0.056274, 0.134112, -0.017957, -0.095586, 0.061802, 0.049811, 0.039654, -0.079547, -0.136659, 0.010998, 0.
101880, -0.058070, -0.095349, 0.019299, 0.076037, -0.132955, 0.115853, -0.136334, 0.170343, 0.011617, -0.070295, -0.058128, -0.138204, -0.045440, -0.094381, 0.084882,
-0.169541, -0.021891, 0.137458, 0.125114, 0.068998, 0.122854, -0.006618, -0.154524, 0.169891, -0.104682, 0.116913, 0.153831, -0.155533, 0.150546, -0.062595, -0.021405,
```

- Synth
  - Set directives with non-exist name
  - Set wrong directives (PIPELINE for # filters in CNN2)
- Cosim
  - Deadlock situation

HLS Debugging

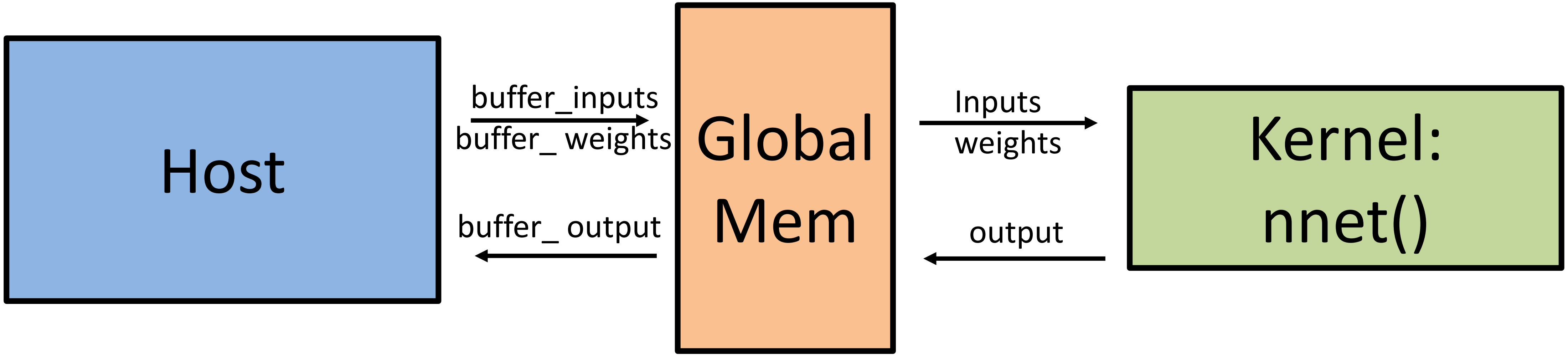
- Cosim mismatch
  - Pass Csim & No warning at Synth
  - If empty occurs, pixel will loss!

```
for (i = 0; i < CONV1_BUFFER_SIZE; i++) {
    if (in.empty() == 0) {
        in >> placeholder;
        conv_buff.shift_up(0);
        conv_buff.insert_top(placeholder, 0);
    }
}
```

```
void check_read(hls::stream<float24_t>& in, float24_t& out_val){
    // #pragma
    while(true){
        if(in.empty() == 0){
            in >> out_val;
            break;
        }
    }
}
```

```
for (i = 0; i < CONV1_BUFFER_SIZE; i++) {
    check_read(in, placeholder);
    conv_buff.shift_up(0);
    conv_buff.insert_top(placeholder, 0);
}
```

Vitis Acceleration



Fixed Point Arithmetic

- Fixed point operations are less precise, use less hardware and won’t affect the global accuracy of the Neural Network

```
#include "ap_fixed.h"
typedef ap_fixed<WIDTH, INT_WIDTH> fixed_p;
```

- Fixed point arithmetic uses significantly fewer resources for basic operations

- We use word length of 16 and int width of 4 for implementation

```
#define EXP_WIDTH   16
#define INT_WIDTH   4
```

Host program

- Create context, command, queue, program and kernel

```
cl::Context context(device);
cl::CommandQueue q(context, device, CL_QUEUE_PROFILING_ENABLE);


cl::Program::Binaries bins = xcl::import_binary_file(args.binary_file);
devices.resize(1);
cl::Program program(context, devices, bins);
cl::Kernel convolve_kernel(program, args.kernel_name);
```

- Buffer allocation for all input activations & weights

```
cl::Buffer buffer_input(context, CL_MEM_READ_ONLY, frame_bytes, NULL);
cl::Buffer buffer_output(context, CL_MEM_WRITE_ONLY, frame_bytes, NULL);
cl::Buffer buffer_coefficient(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, coefficient_size_bytes, filter_coeff.data());


convolve_kernel.setArg(0, buffer_input);
convolve_kernel.setArg(1, buffer_output);
convolve_kernel.setArg(2, buffer_coefficient);
convolve_kernel.setArg(3, coefficient_size);
convolve_kernel.setArg(4, args.width);
convolve_kernel.setArg(5, args.height);
```

Host program

- Writing buffers to FPGA memory

```
q.enqueueMigrateMemObjects({buffer_coefficient}, 0);
q.enqueueWriteBuffer(buffer_input, CL_FALSE, 0, frame_bytes, inFrame.data());
```

- Launch the kernel

```
q.enqueueTask(convolve_kernel);
q.finish();
```

- Reading buffers from FPGA memory

```
q.enqueueReadBuffer(buffer_output, CL_TRUE, 0, frame_bytes, outFrame.data());
q.finish();
```

- Compute software results

```
test(inFrame, outFrame, coefficients, coefficient_size, 512, 10);
```

SW Emulation

```
[###################################] 100 %rgs.nframes 1


Processed 0.02 MB in 8.743s (0.00 MBps)

cmp ../build/baseline/sw_emu/output.mp4 ../golden_out_small.mp4; RETVAL=$?; \
if [ $RETVAL -eq \0 ];  then \
  echo "PASS : Output Video Matches the Golden Output Results " ; \
else \
  echo "FAIL : Output Video Corrupt" ; \
fi
PASS : Output Video Matches the Golden Output Results
```