FINN Verification

Lecturer: Hua-Yang Weng

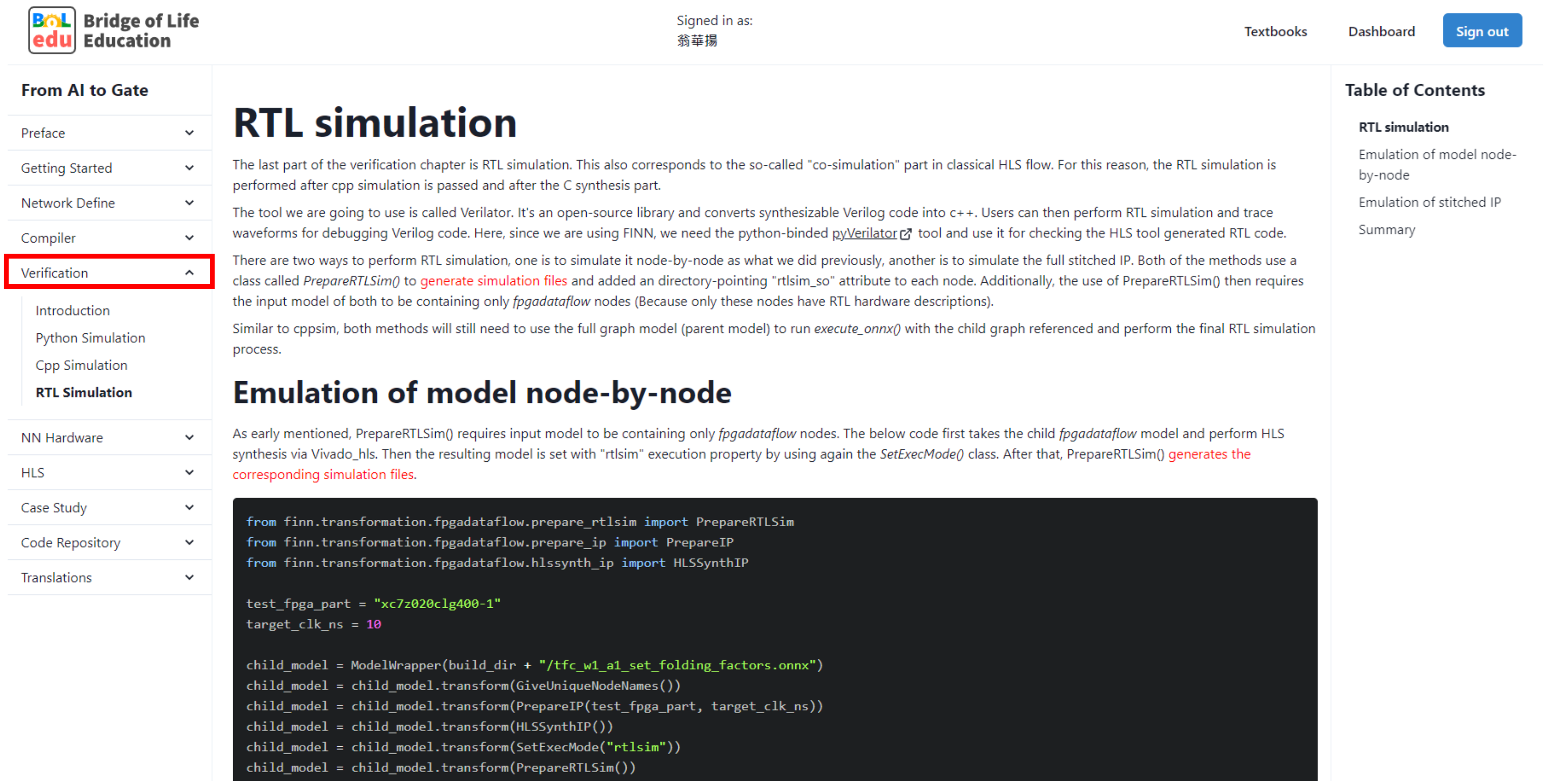
Date: 2022/08/28

https://github.com/Xilinx/finn/blob/main/notebooks/end2end_example/bnn-pynq/tfc_end2end_verification.ipynb

https://github.com/Xilinx/finn-hlslib/tree/master/tb

©BOLEDU

From AI to Gate Textbook



Bridge of Life Education

Signed in as:
翁華揚

Textbooks Dashboard Sign out

From AI to Gate

RTL simulation

The last part of the verification chapter is RTL simulation. This also corresponds to the so-called "co-simulation" part in classical HLS flow. For this reason, the RTL simulation is performed after cpp simulation is passed and after the C synthesis part.

The tool we are going to use is called Verilator. It's an open-source library and converts synthesizable Verilog code into c++. Users can then perform RTL simulation and trace waveforms for debugging Verilog code. Here, since we are using FINN, we need the python-binded pyVerilator tool and use it for checking the HLS tool generated RTL code.

There are two ways to perform RTL simulation, one is to simulate it node-by-node as what we did previously, another is to simulate the full stitched IP. Both of the methods use a class called *PrepareRTLSim()* to generate simulation files and added an directory-pointing "rtlsim_so" attribute to each node. Additionally, the use of PrepareRTLSim() then requires the input model of both to be containing only *fpgadataflow* nodes (Because only these nodes have RTL hardware descriptions).

Similar to cppsim, both methods will still need to use the full graph model (parent model) to run *execute_onnx()* with the child graph referenced and perform the final RTL simulation process.

Emulation of model node-by-node

As early mentioned, PrepareRTLSim() requires input model to be containing only *fpgadataflow* nodes. The below code first takes the child *fpgadataflow* model and perform HLS synthesis via Vivado_hls. Then the resulting model is set with "rtlsim" execution property by using again the *SetExecMode()* class. After that, PrepareRTLSim() generates the corresponding simulation files.

```
from finn.transformation.fpgadataflow.prepare_rtlsim import PrepareRTLSim
from finn.transformation.fpgadataflow.prepare_ip import PrepareIP
from finn.transformation.fpgadataflow.hlssynth_ip import HLSSynthIP

test_fpga_part = "xc7z020clg400-1"
target_clk_ns = 10

child_model = ModelWrapper(build_dir + "/tfc_w1_a1_set_folding_factors.onnx")
child_model = child_model.transform(GiveUniqueNodeNames())
child_model = child_model.transform(PrepareIP(test_fpga_part, target_clk_ns))
child_model = child_model.transform(HLSSynthIP())
child_model = child_model.transform(SetExecMode("rtlsim"))
child_model = child_model.transform(PrepareRTLSim())
```

Overview

- Introduction
- Python Simulation
- Cpp Simulation
- RTL Simulation
- HLS Testbench

©BOLEDU

Overview

- Introduction
- Python Simulation
- Cpp Simulation
- RTL Simulation
- HLS Testbench

©BOLEDU

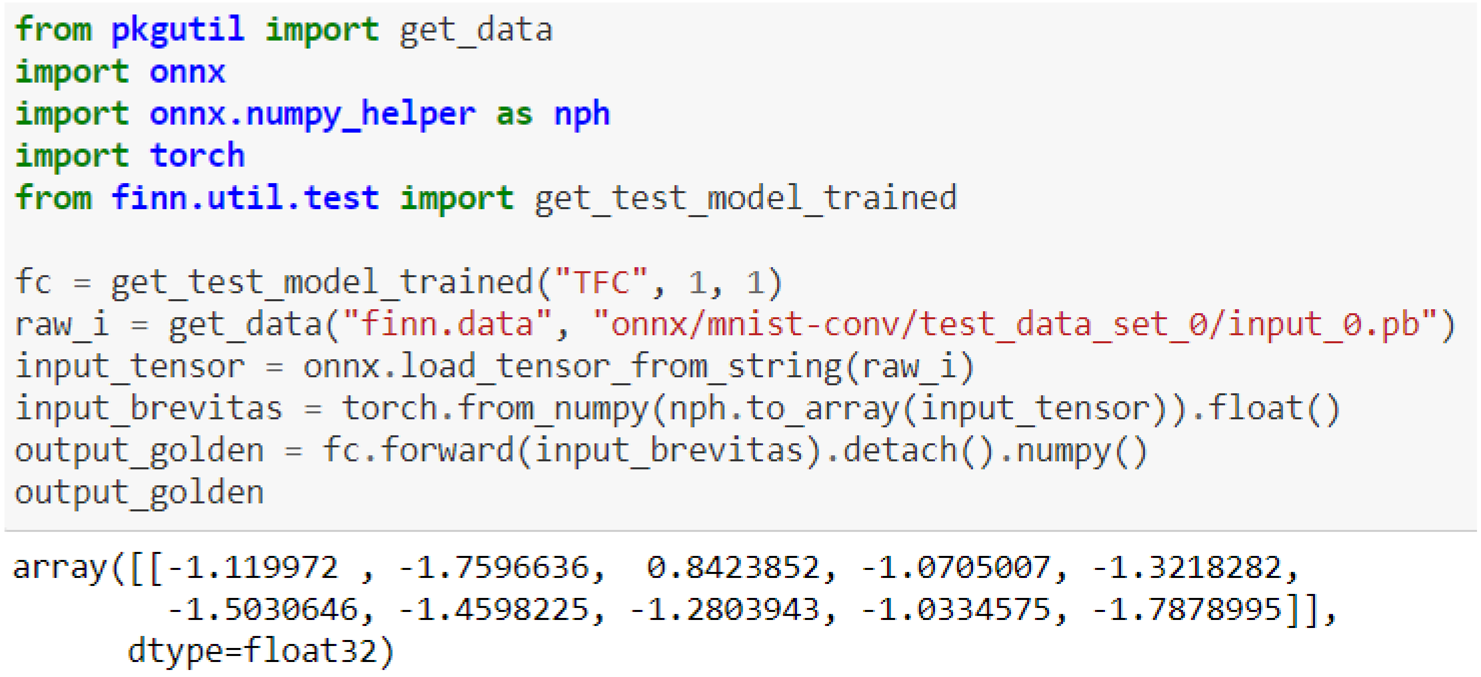
Introduction (1/3)

- **Python Simulation:** (Behavior-Level)
  - verification executed within FINN compiler.
  - Focus on python language
  - For python executed code.
- **Cpp Simulation:** (Behavior-Level)
  - C/C++ executed code.
  - Enables us to check the HLS codes for hardware.
- **RTL Simulation:** (Register-Transfer-Level)
  - Performs cycle accurate tests and verifies the final hardware HDL implementation.

©BOLEDU

Introduction (2/3): Golden

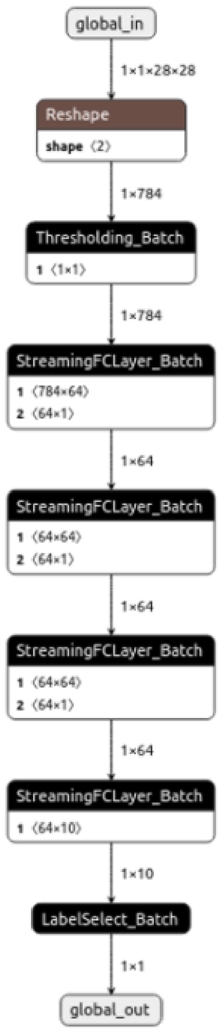
- Calculated directly from the Brevitas
  - Running some example data from the MNIST



```
from pkgutil import get_data
import onnx
import onnx.numpy_helper as nph
import torch
from finn.util.test import get_test_model_trained

fc = get_test_model_trained("TFC", 1, 1)
raw_i = get_data("finn.data", "onnx/mnist-conv/test_data_set_0/input_0.pb")
input_tensor = onnx.load_tensor_from_string(raw_i)
input_brevitas = torch.from_numpy(nph.to_array(input_tensor)).float()
output_golden = fc.forward(input_brevitas).detach().numpy()
output_golden
```

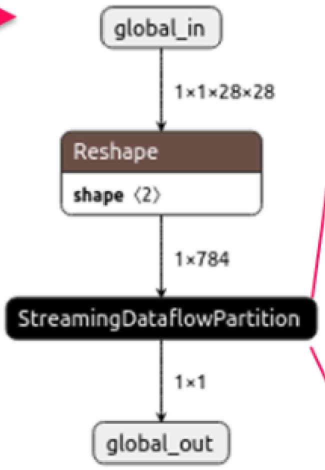
```
array([[-1.119972 , -1.7596636,  0.8423852, -1.0705007, -1.3218282,
        -1.5030646, -1.4598225, -1.2803943, -1.0334575, -1.7878995]],
      dtype=float32)
```

Introduction (3/3)

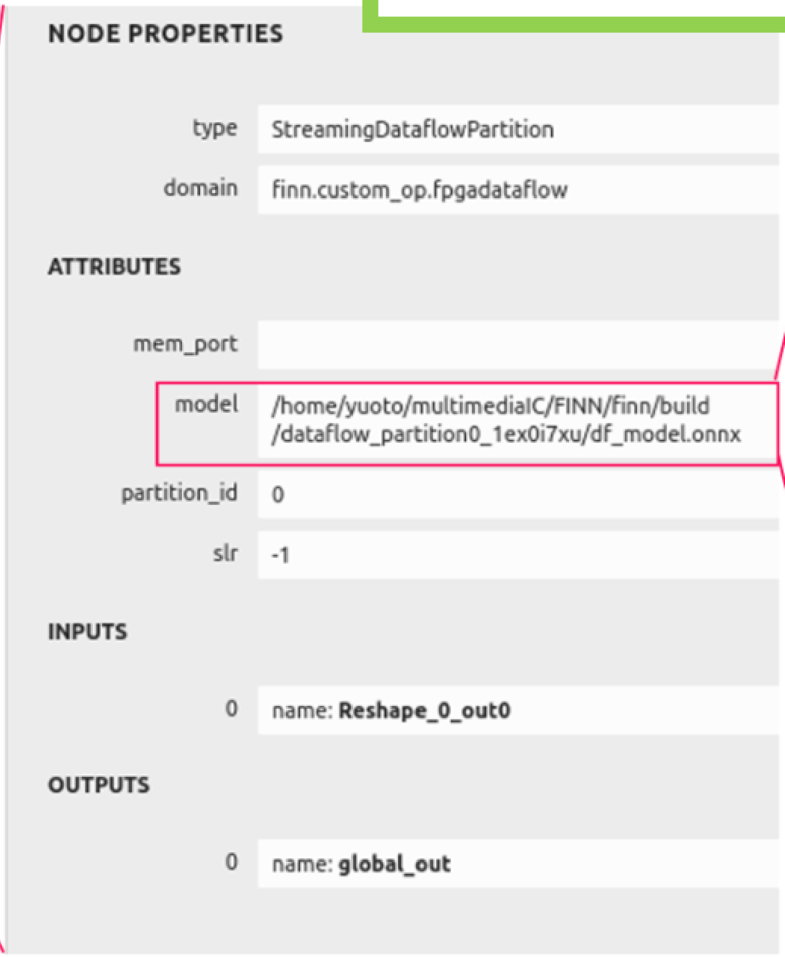


Parent Node

create dataflow partition



Child node



NODE PROPERTIES

type	StreamingDataflowPartition
domain	finn.custom_op.fpgadataflow

ATTRIBUTES

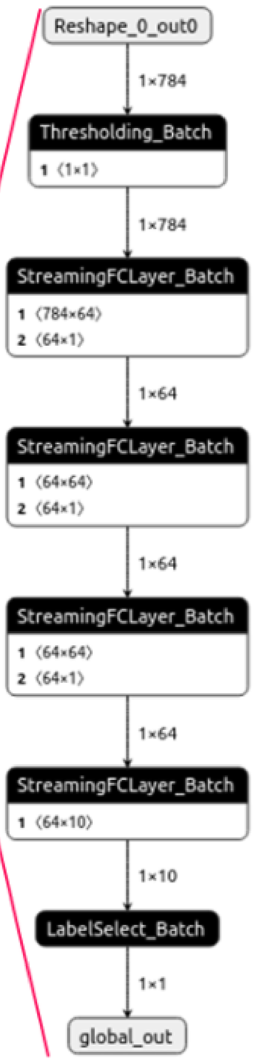
mem_port	
model	/home/yuoto/multimedialC/FINN/finn/build /dataflow_partition0_1ex0i7xu/df_model.onnx
partition_id	0
slr	-1

INPUTS

0	name: **Reshape_0_out0**

OUTPUTS

0	name: **global_out**

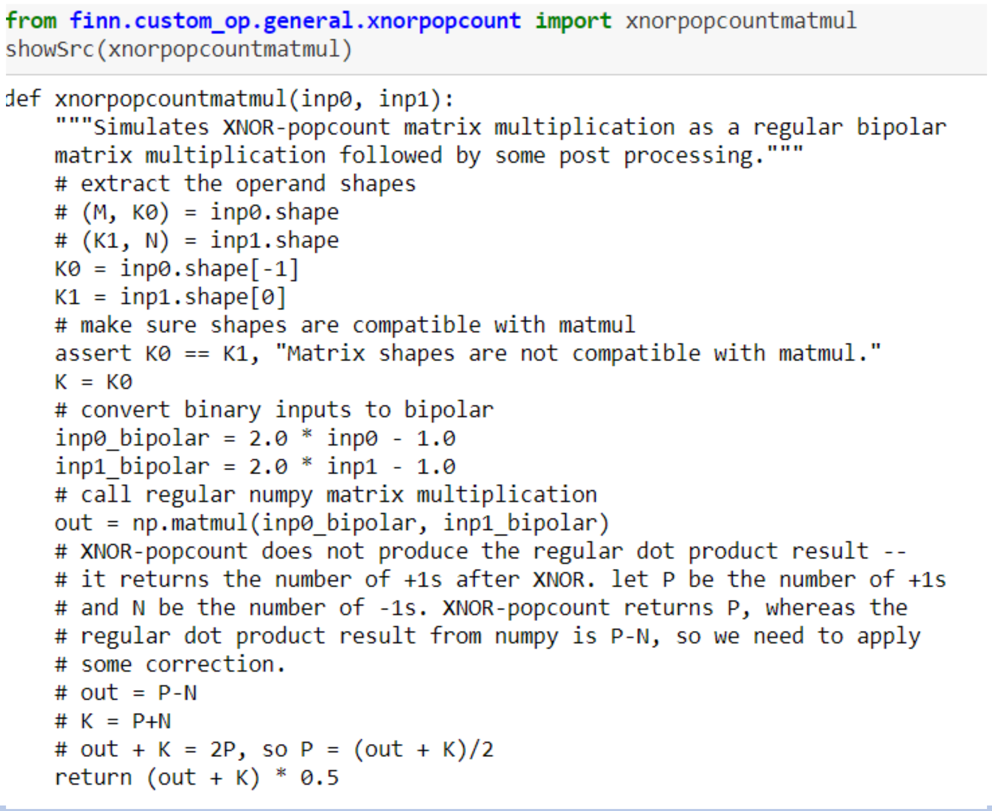


Overview

- Introduction
- Python Simulation
- Cpp Simulation
- RTL Simulation

Python Simulation (1/3)

- Functionality check for operations that do not belong to the **fpgadataflow** backend attribute



```python
from finn.custom_op.general.xnorpopcount import xnorpopcountmatmul
showSrc(xnorpopcountmatmul)
```

```python
def xnorpopcountmatmul(inp0, inp1):
    """Simulates XNOR-popcount matrix multiplication as a regular bipolar
    matrix multiplication followed by some post processing."""
    # extract the operand shapes
    # (M, K0) = inp0.shape
    # (K1, N) = inp1.shape
    K0 = inp0.shape[-1]
    K1 = inp1.shape[0]
    # make sure shapes are compatible with matmul
    assert K0 == K1, "Matrix shapes are not compatible with matmul."
    K = K0
    # convert binary inputs to bipolar
    inp0_bipolar = 2.0 * inp0 - 1.0
    inp1_bipolar = 2.0 * inp1 - 1.0
    # call regular numpy matrix multiplication
    out = np.matmul(inp0_bipolar, inp1_bipolar)
    # XNOR-popcount does not produce the regular dot product result --
    # it returns the number of +1s after XNOR. let P be the number of +1s
    # and N be the number of -1s. XNOR-popcount returns P, whereas the
    # regular dot product result from numpy is P-N, so we need to apply
    # some correction.
    # out = P-N
    # K = P+N
    # out + K = 2P, so P = (out + K)/2
    return (out + K) * 0.5
```

- Example of the execution function of a XNOR popcount node.

- Contains descriptions of the behavior in Python and calculate the result of the node.

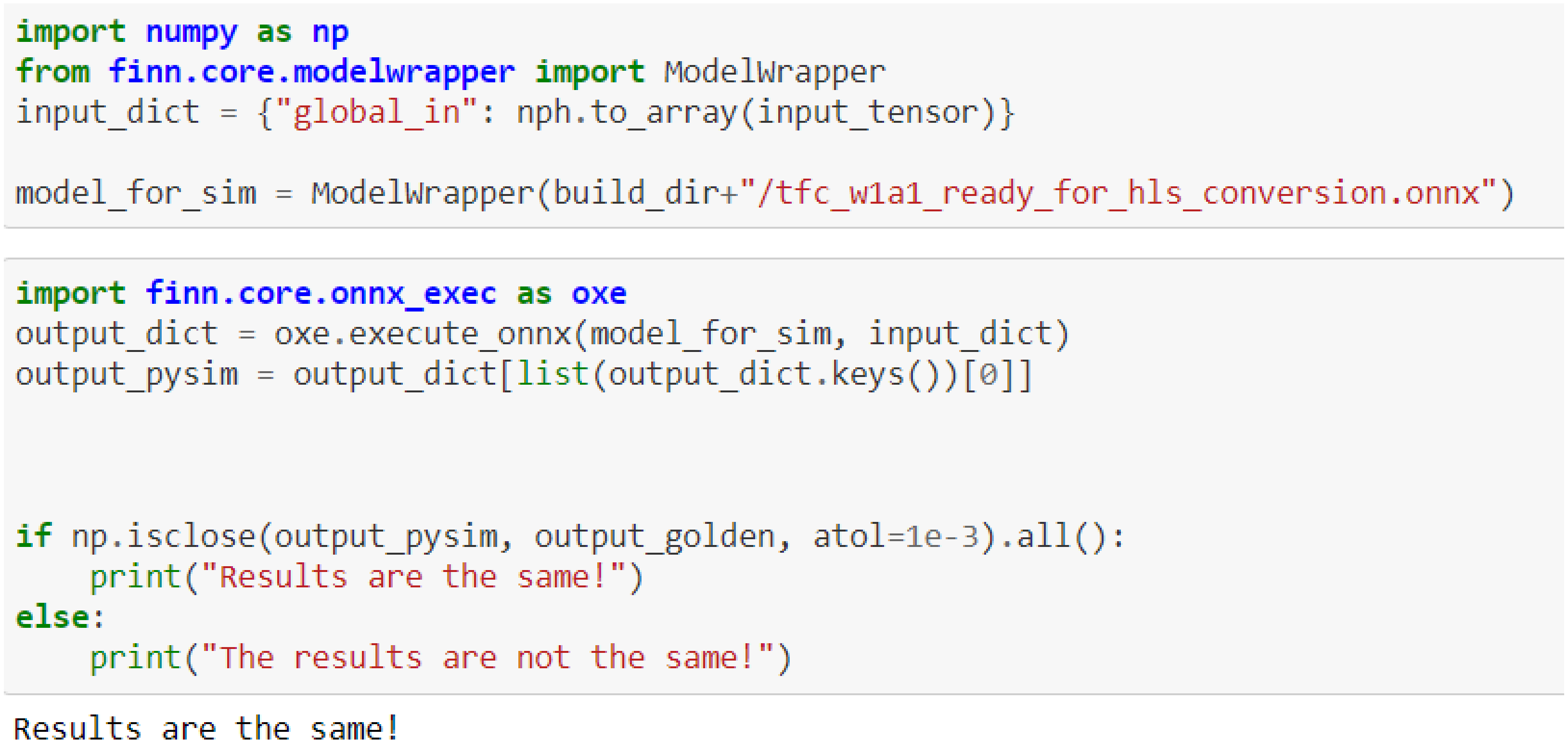
Python Simulation (2/3)

- Standard ONNX node simulation
  - **onnxruntime** is used. (open source tool)
  - Runs at host side

- **execute_onnx( )**
  - FINN API to simulate the model node-by-node using onnxruntime
  - The result is stored in a context dictionary

©BOLEDU

Python Simulation (3/3)



```
import numpy as np
from finn.core.modelwrapper import ModelWrapper
input_dict = {"global_in": nph.to_array(input_tensor)}

model_for_sim = ModelWrapper(build_dir+"/tfc_w1a1_ready_for_hls_conversion.onnx")
```

```
import finn.core.onnx_exec as oxe
output_dict = oxe.execute_onnx(model_for_sim, input_dict)
output_pysim = output_dict[list(output_dict.keys())[0]]



if np.isclose(output_pysim, output_golden, atol=1e-3).all():
    print("Results are the same!")
else:
    print("The results are not the same!")
```

```
Results are the same!
```

The result is compared with the theoretical "golden" value for verification.

Overview

- Introduction
- Python Simulation
- Cpp Simulation
- RTL Simulation
- HLS Testbench

©BOLEDU

Cpp Simulation (1/6)

- Simulate nodes with backend attribute of

  “HLS custom op”

- c++ executable tests the c++ HLS functions
  1. Input data from Brevitas is stored in an .npy file
  2. Reads the input
  3. Streams to HLS function (finn-hlslib)
  4. Writes the result to a new .npy file.
- The resulting .npy file can be read into FINN

©BOLEDU

Cpp Simulation (2/6)

- Flow:
  1. PrepareCppSim( )
  2. **CompileCppSim( )**
  3. SetExecMode( )
  4. execute_onnx( )

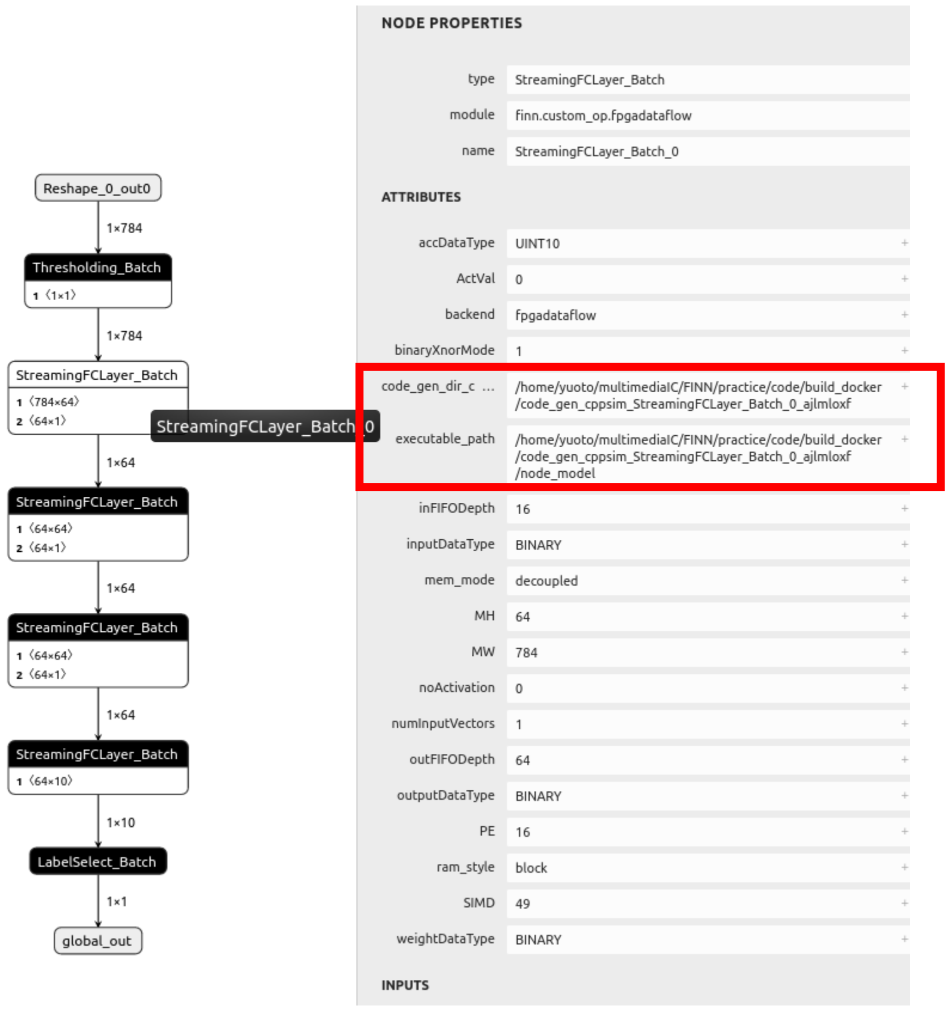
1. PrepareCppSim: generates the C++ code for the corresponding hls layer

2. CompileCppSim: Compiles the C++ code and stores the path to the executable

```
from finn.transformation.fpgadataflow.prepare_cppsim import PrepareCppSim
from finn.transformation.fpgadataflow.compile_cppsim import CompileCppSim
from finn.transformation.general import GiveUniqueNodeNames

model_for_cppsim = model_for_cppsim.transform(GiveUniqueNodeNames())
model_for_cppsim = model_for_cppsim.transform(PrepareCppSim())
model_for_cppsim = model_for_cppsim.transform(CompileCppSim())
```

©BOLEDU

Cpp Simulation (3/6)



NODE PROPERTIES

type	StreamingFCLayer_Batch
module	finn.custom_op.fpgadataflow
name	StreamingFCLayer_Batch_0

ATTRIBUTES

accDataType	UINT10
ActVal	0
backend	fpgadataflow
binaryXnorMode	1
code_gen_dir_c …	/home/yuoto/multimedialC/FINN/practice/code/build_docker /code_gen_cppsim_StreamingFCLayer_Batch_0_ajlmloxf
executable_path	/home/yuoto/multimedialC/FINN/practice/code/build_docker /code_gen_cppsim_StreamingFCLayer_Batch_0_ajlmloxf /node_model
inFIFODepth	16
inputDataType	BINARY
mem_mode	decoupled
MH	64
MW	784
noActivation	0
numInputVectors	1
outFIFODepth	64
outputDataType	BINARY
PE	16
ram_style	block
SIMD	49
weightDataType	BINARY

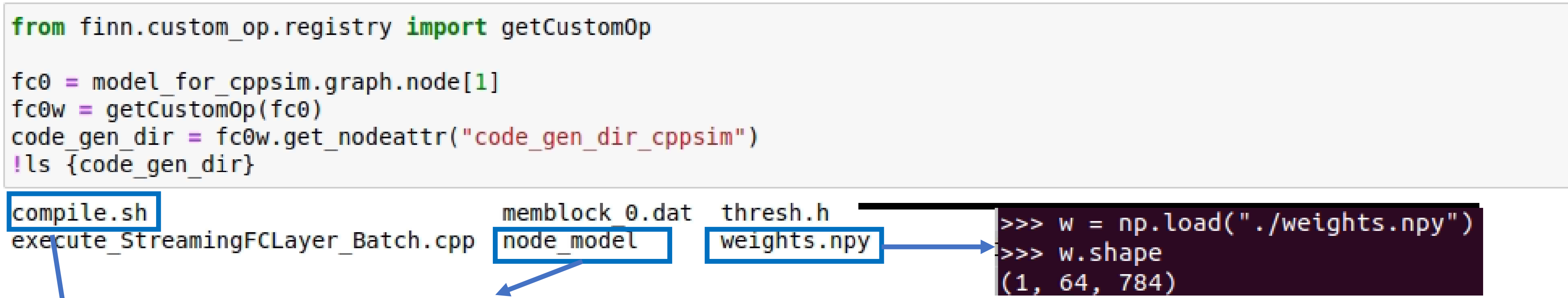
INPUTS

The following node attributes have been added:

- **code_gen_dir_cppsim:** Directory where the files for the simulation using C++ are stored
- **executable_path:** specifies the path to the executable

Cpp Simulation (4/6)

- Directory “*StreamingFCLayer_Batch_0_ajlmloxf*”



```
from finn.custom_op.registry import getCustomOp

fc0 = model_for_cppsim.graph.node[1]
fc0w = getCustomOp(fc0)
code_gen_dir = fc0w.get_nodeattr("code_gen_dir_cppsim")
!ls {code_gen_dir}
```

```
compile.sh                           memblock_0.dat   thresh.h
execute_StreamingFCLayer_Batch.cpp   node_model       weights.npy
```

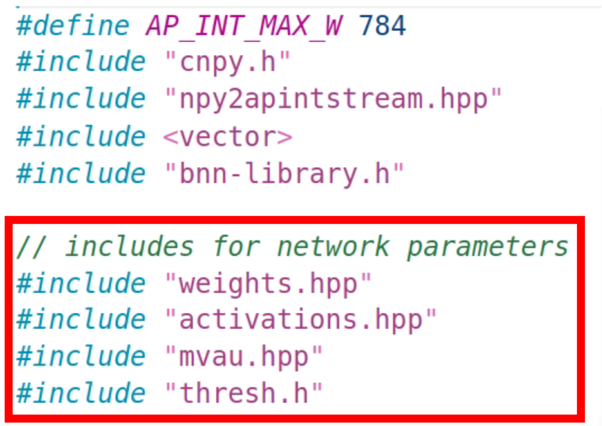
```
>>> w = np.load("./weights.npy")
>>> w.shape
(1, 64, 784)
```

Executable file

Inside **compile.sh**
g++ -o
/home/yuoto/multimediaIC/FINN/practice/code/build_docker/code_gen_cppsim_StreamingFCLayer_Batch_0_ajlmloxf/node_model
/home/yuoto/multimediaIC/FINN/practice/code/build_docker/code_gen_cppsim_StreamingFCLayer_Batch_0_ajlmloxf/*.cpp /workspace/cnpy/cnpy.cpp
-I/workspace/finn/src/finn/qnn-data/cpp
-I/workspace/cnpy/
-I/workspace/finn-hlslib
-I/home/yuoto/YuotoSSD/Xilinx/Vivado/2020.1/include --std=c++11 -O3 -lz

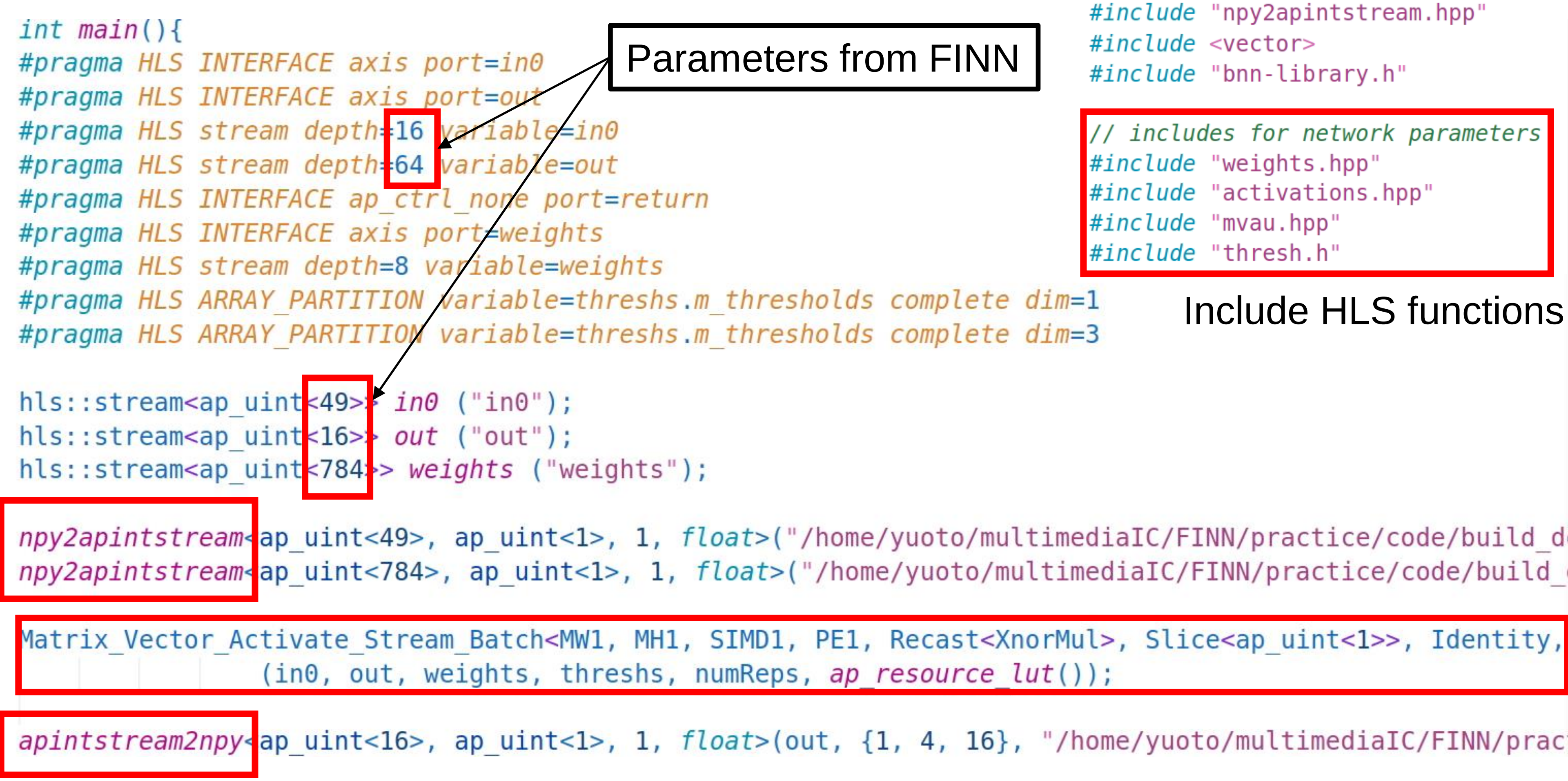
Cpp Simulation (5/6)

Inside “execute_StreamingFCLayer_Batch.cpp”



```
#define AP_INT_MAX_W 784
#include "cnpy.h"
#include "npy2apintstream.hpp"
#include <vector>
#include "bnn-library.h"

// includes for network parameters
#include "weights.hpp"
#include "activations.hpp"
#include "mvau.hpp"
#include "thresh.h"
```

Include HLS functions



Parameters from FINN

```
int main(){
#pragma HLS INTERFACE axis port=in0
#pragma HLS INTERFACE axis port=out
#pragma HLS stream depth=16 variable=in0
#pragma HLS stream depth=64 variable=out
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=weights
#pragma HLS stream depth=8 variable=weights
#pragma HLS ARRAY_PARTITION variable=threshs.m_thresholds complete dim=1
#pragma HLS ARRAY_PARTITION variable=threshs.m_thresholds complete dim=3

hls::stream<ap_uint<49>> in0 ("in0");
hls::stream<ap_uint<16>> out ("out");
hls::stream<ap_uint<784>> weights ("weights");

npy2apintstream<ap_uint<49>, ap_uint<1>, 1, float>("/home/yuoto/multimediaIC/FINN/practice/code/build_d
npy2apintstream<ap_uint<784>, ap_uint<1>, 1, float>("/home/yuoto/multimediaIC/FINN/practice/code/build_

Matrix_Vector_Activate_Stream_Batch<MW1, MH1, SIMD1, PE1, Recast<XnorMul>, Slice<ap_uint<1>>, Identity,
            (in0, out, weights, threshs, numReps, ap_resource_lut());

apintstream2npy<ap_uint<16>, ap_uint<1>, 1, float>(out, {1, 4, 16}, "/home/yuoto/multimediaIC/FINN/prac
```

Cpp Simulation (6/6)

3. SetExecMode( )

- Sets the execution mode
- E.g. cpp simulation -> "cppsim".

```
from finn.transformation.fpgadataflow.set_exec_mode import SetExecMode

model_for_cppsim = model_for_cppsim.transform(SetExecMode("cppsim"))
model_for_cppsim.save(build_dir+"/tfc_w1_a1_for_cppsim.onnx")
```

4. execute_onnx( )

- Need to integrate the child model in the parent model first.

```
parent_model = ModelWrapper(build_dir+"/tfc_w1_a1_dataflow_parent.onnx")
sdp_node = parent_model.graph.node[2]
child_model = build_dir + "/tfc_w1_a1_for_cppsim.onnx"
getCustomOp(sdp_node).set_nodeattr("model", child_model)
output_dict = oxe.execute_onnx(parent_model, input_dict)
output_cppsim = output_dict[list(output_dict.keys())[0]]

if np.isclose(output_cppsim, output_golden, atol=1e-3).all():
    print("Results are the same!")
else:
    print("The results are not the same!")
```

©BOLEDU

Overview

- Introduction
- Python Simulation
- Cpp Simulation
- RTL Simulation
- HLS Testbench

©BOLEDU

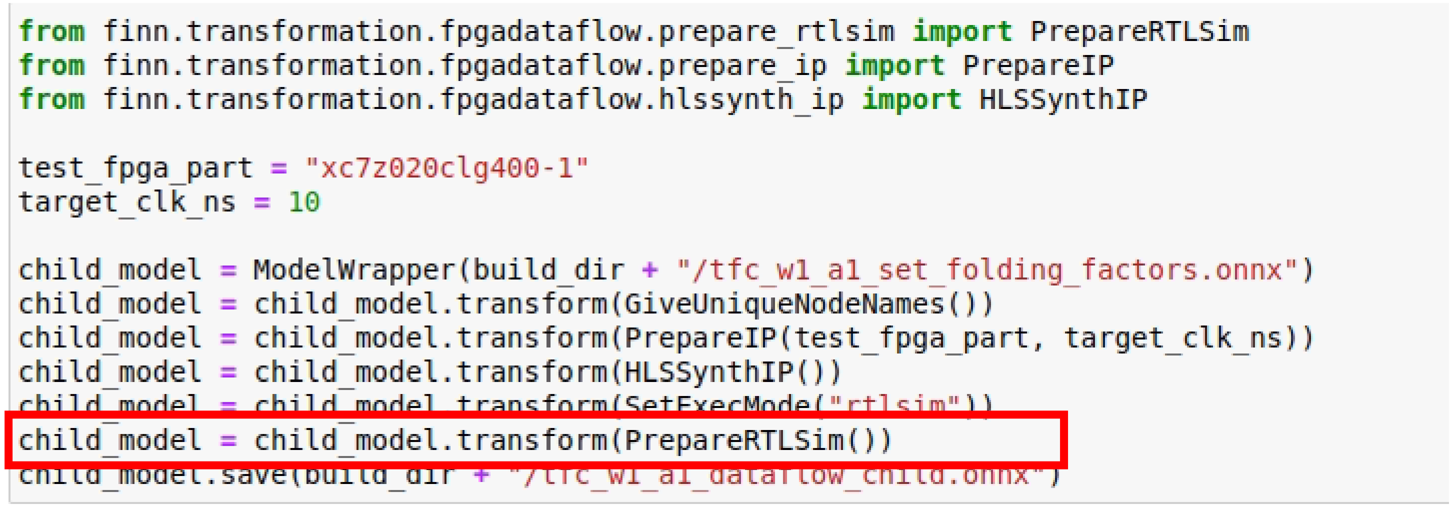
RTL Simulation (1/6)

- After IP blocks are generated from the corresponding HLS layers.
- RTL cosimulation using PyVerilator
- Two ways for rtlsim:
  1. node-by-node
  2. Executed as whole
     - Required all nodes to be HLS nodes

©BOLEDU

RTL Simulation (2/6): 1. node-by-node

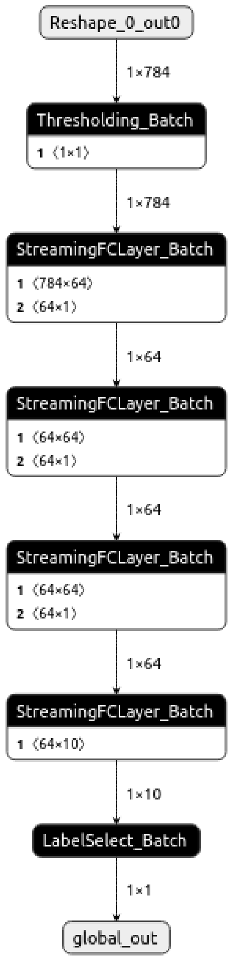
- PrepareRTLSim( )
  - Apply to the child model.
  - Sets the execution mode to “rtlsim".
  - New node attribute "rtlsim_so" are created

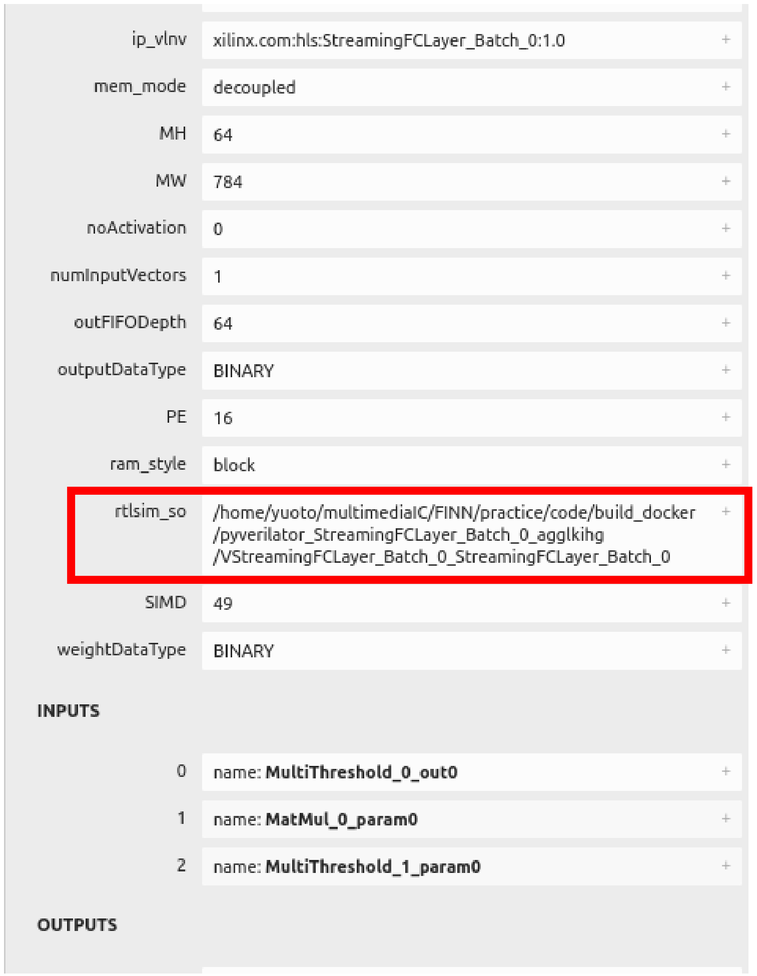


```
from finn.transformation.fpgadataflow.prepare_rtlsim import PrepareRTLSim
from finn.transformation.fpgadataflow.prepare_ip import PrepareIP
from finn.transformation.fpgadataflow.hlssynth_ip import HLSSynthIP

test_fpga_part = "xc7z020clg400-1"
target_clk_ns = 10

child_model = ModelWrapper(build_dir + "/tfc_w1_a1_set_folding_factors.onnx")
child_model = child_model.transform(GiveUniqueNodeNames())
child_model = child_model.transform(PrepareIP(test_fpga_part, target_clk_ns))
child_model = child_model.transform(HLSSynthIP())
child_model = child_model.transform(SetExecMode("rtlsim"))
child_model = child_model.transform(PrepareRTLSim())
child_model.save(build_dir + "/tfc_w1_a1_dataflow_child.onnx")
```

©BOLEDU

RTL Simulation (3/6): 1. node-by-node





ip_vlnv	xilinx.com:hls:StreamingFCLayer_Batch_0:1.0
mem_mode	decoupled
MH	64
MW	784
noActivation	0
numInputVectors	1
outFIFODepth	64
outputDataType	BINARY
PE	16
ram_style	block
rtlsim_so	/home/yuoto/multimedialC/FINN/practice/code/build_docker /pyverilator_StreamingFCLayer_Batch_0_agglkihg /VStreamingFCLayer_Batch_0_StreamingFCLayer_Batch_0
SIMD	49
weightDataType	BINARY

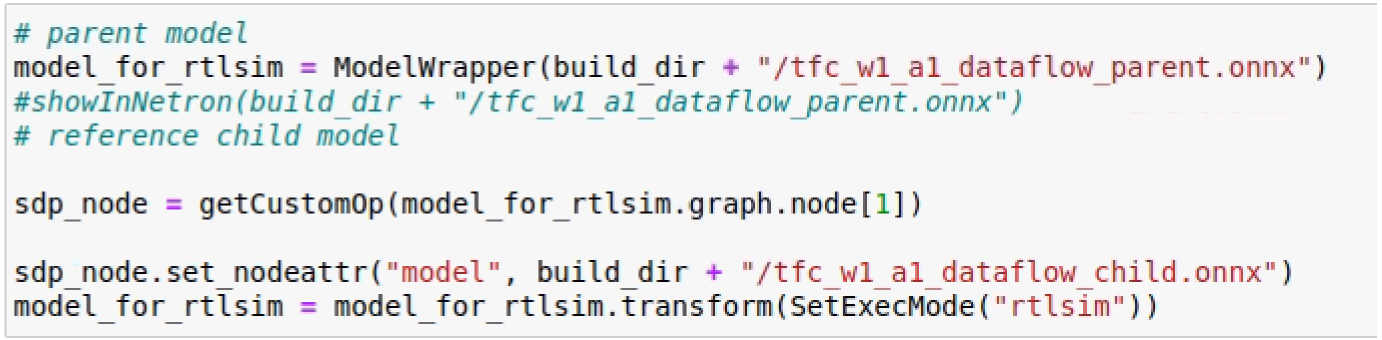
INPUTS

0	name: **MultiThreshold_0_out0**
1	name: **MatMul_0_param0**
2	name: **MultiThreshold_1_param0**

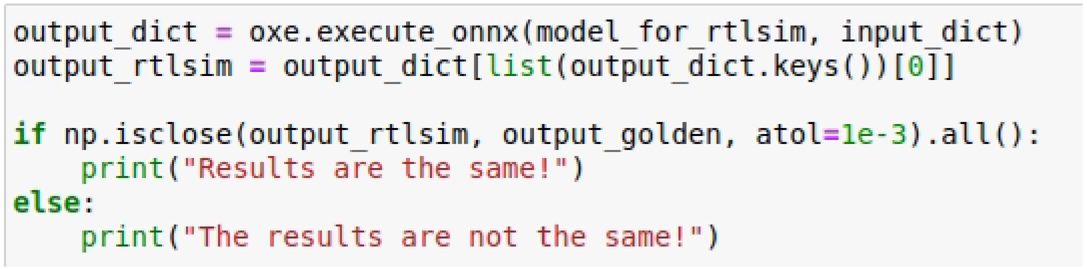
OUTPUTS

RTL Simulation (4/6): 1. node-by-node

- Merge the child node to the parent node
- Then, execute with execute_onnx( )



```
# parent model
model_for_rtlsim = ModelWrapper(build_dir + "/tfc_w1_a1_dataflow_parent.onnx")
#showInNetron(build_dir + "/tfc_w1_a1_dataflow_parent.onnx")
# reference child model

sdp_node = getCustomOp(model_for_rtlsim.graph.node[1])

sdp_node.set_nodeattr("model", build_dir + "/tfc_w1_a1_dataflow_child.onnx")
model_for_rtlsim = model_for_rtlsim.transform(SetExecMode("rtlsim"))
```

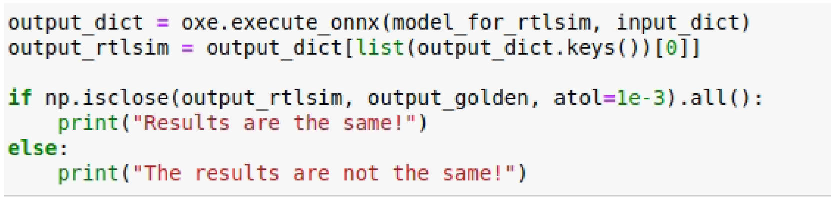


```
output_dict = oxe.execute_onnx(model_for_rtlsim, input_dict)
output_rtlsim = output_dict[list(output_dict.keys())[0]]

if np.isclose(output_rtlsim, output_golden, atol=1e-3).all():
    print("Results are the same!")
else:
    print("The results are not the same!")
```

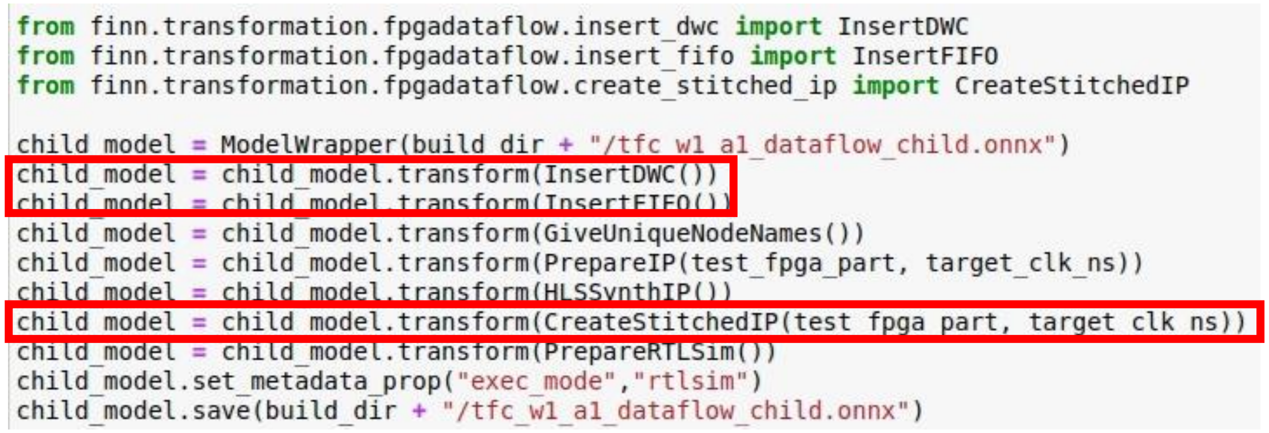
©BOLEDU

RTL Simulation (6/6): 2. Executed as whole

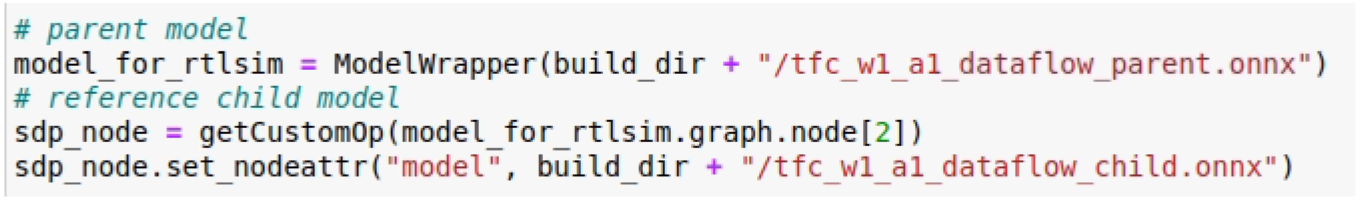
- Simulate the whole (stitched) child model at once.
  - Merged to parent model
  - Execute at parent model



```
output_dict = oxe.execute_onnx(model_for_rtlsim, input_dict)
output_rtlsim = output_dict[list(output_dict.keys())[0]]

if np.isclose(output_rtlsim, output_golden, atol=1e-3).all():
    print("Results are the same!")
else:
    print("The results are not the same!")
```



```
from finn.transformation.fpgadataflow.insert_dwc import InsertDWC
from finn.transformation.fpgadataflow.insert_fifo import InsertFIFO
from finn.transformation.fpgadataflow.create_stitched_ip import CreateStitchedIP

child_model = ModelWrapper(build_dir + "/tfc_w1_a1_dataflow_child.onnx")
child_model = child_model.transform(InsertDWC())
child_model = child_model.transform(InsertFIFO())
child_model = child_model.transform(GiveUniqueNodeNames())
child_model = child_model.transform(PrepareIP(test_fpga_part, target_clk_ns))
child_model = child_model.transform(HLSSynthIP())
child_model = child_model.transform(CreateStitchedIP(test_fpga_part, target_clk_ns))
child_model = child_model.transform(PrepareRTLSim())
child_model.set_metadata_prop("exec_mode","rtlsim")
child_model.save(build_dir + "/tfc_w1_a1_dataflow_child.onnx")
```



```
# parent model
model_for_rtlsim = ModelWrapper(build_dir + "/tfc_w1_a1_dataflow_parent.onnx")
# reference child model
sdp_node = getCustomOp(model_for_rtlsim.graph.node[2])
sdp_node.set_nodeattr("model", build_dir + "/tfc_w1_a1_dataflow_child.onnx")
```

Overview

- Introduction
- Python Simulation
- Cpp Simulation
- RTL Simulation
- HLS Testbench

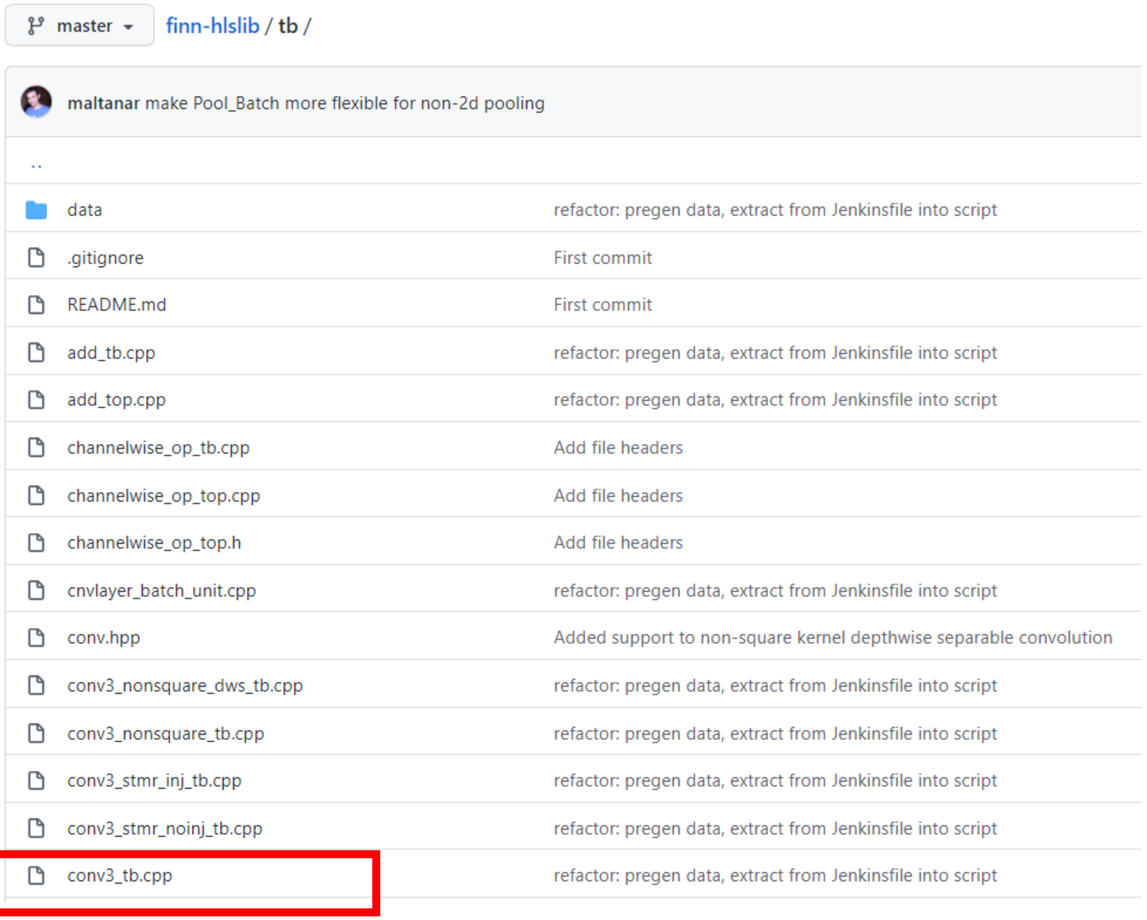
©BOLEDU

HLS Testbench

- For HLS library itself or other HLS custom function
- Can use the Vivado_hls or Vitis_hls tools just as regular HLS development flow
- Testbench for FINN-hlslib @
  - https://github.com/Xilinx/finn-hlslib/tree/master/tb

©BOLEDU

HLS Testbench



master finn-hlslib / tb /

maltanar make Pool_Batch more flexible for non-2d pooling

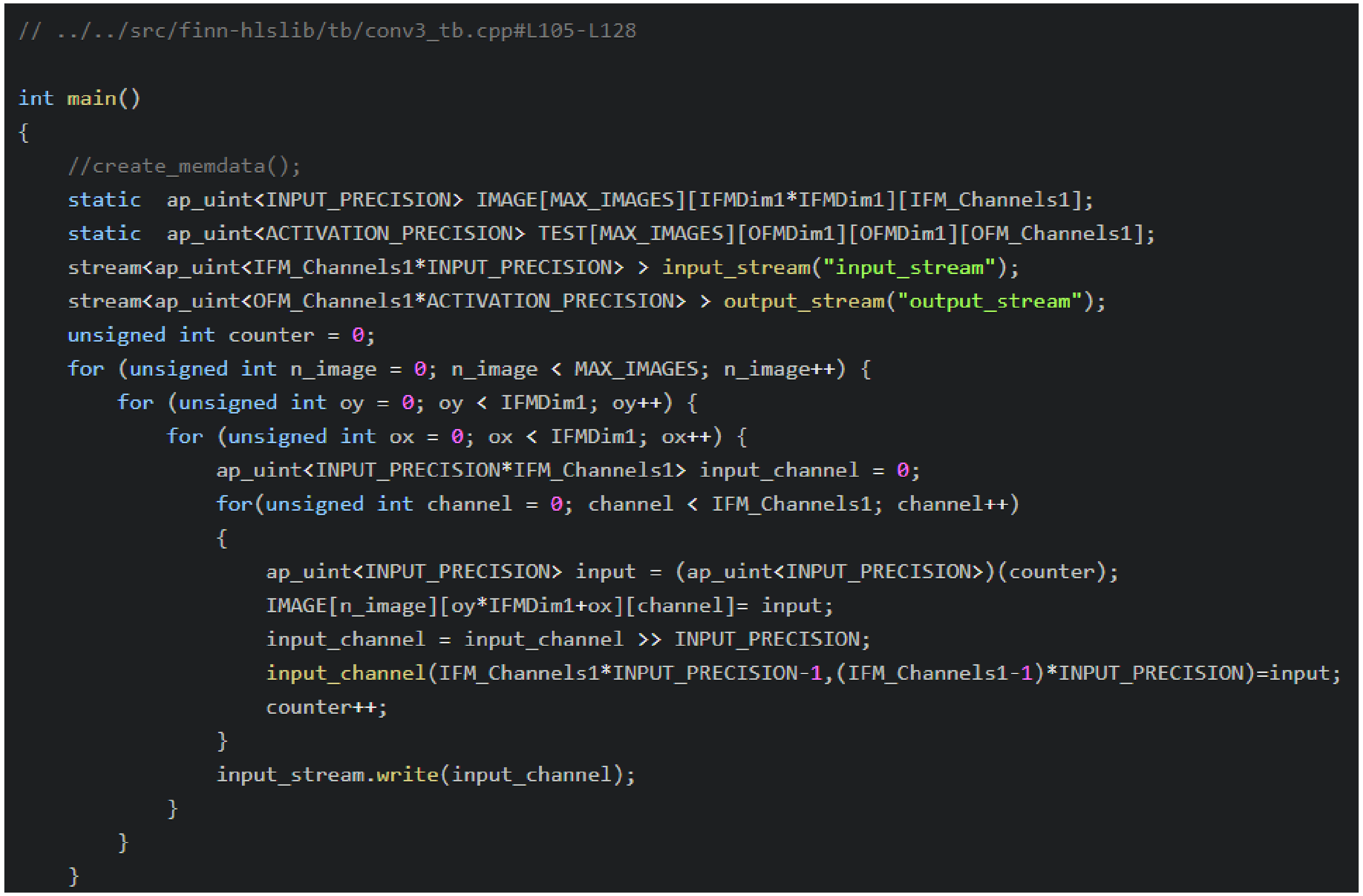
..

data	refactor: pregen data, extract from Jenkinsfile into script
.gitignore	First commit
README.md	First commit
add_tb.cpp	refactor: pregen data, extract from Jenkinsfile into script
add_top.cpp	refactor: pregen data, extract from Jenkinsfile into script
channelwise_op_tb.cpp	Add file headers
channelwise_op_top.cpp	Add file headers
channelwise_op_top.h	Add file headers
cnvlayer_batch_unit.cpp	refactor: pregen data, extract from Jenkinsfile into script
conv.hpp	Added support to non-square kernel depthwise separable convolution
conv3_nonsquare_dws_tb.cpp	refactor: pregen data, extract from Jenkinsfile into script
conv3_nonsquare_tb.cpp	refactor: pregen data, extract from Jenkinsfile into script
conv3_stmr_inj_tb.cpp	refactor: pregen data, extract from Jenkinsfile into script
conv3_stmr_noinj_tb.cpp	refactor: pregen data, extract from Jenkinsfile into script
conv3_tb.cpp	refactor: pregen data, extract from Jenkinsfile into script

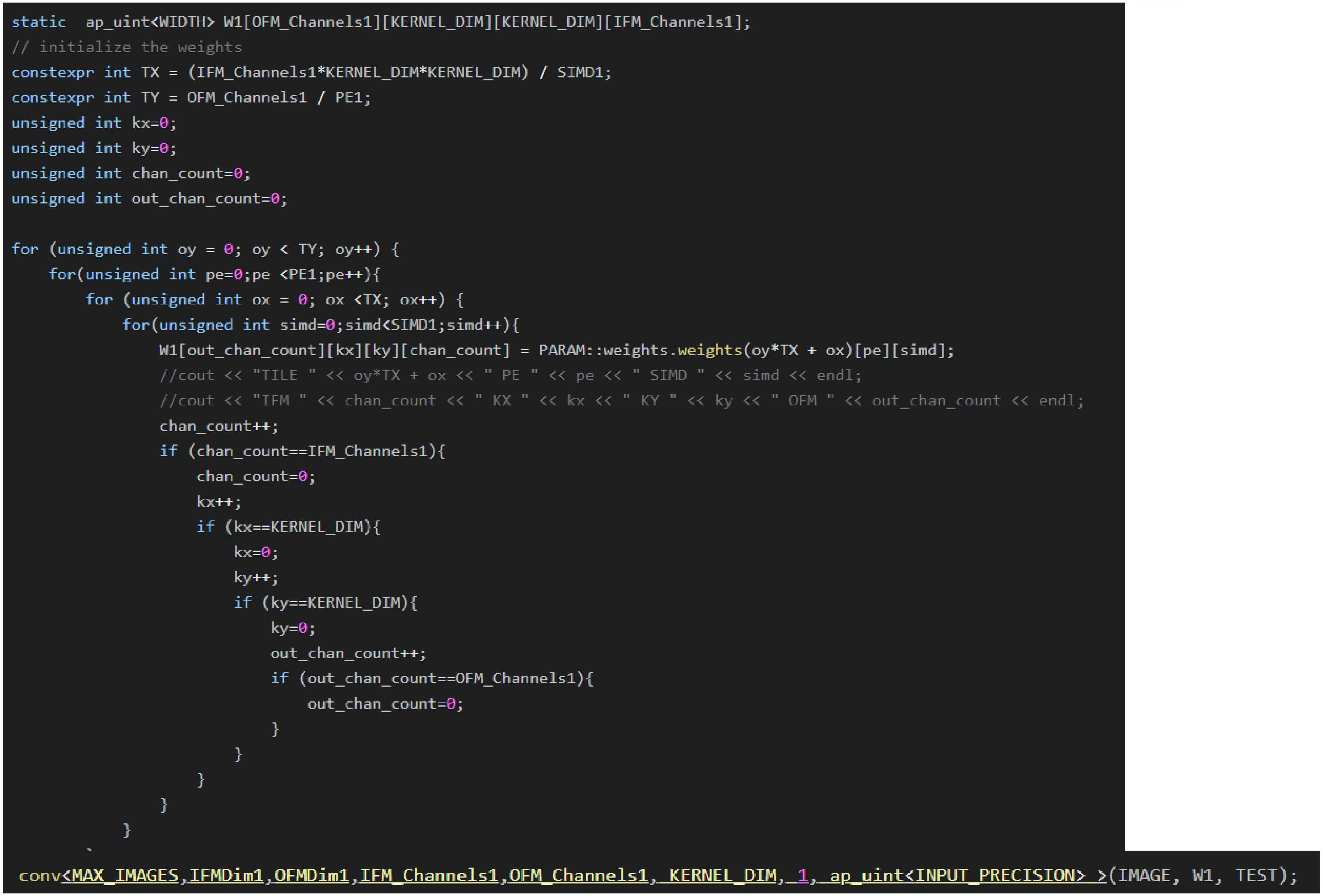
https://github.com/Xilinx/finn-hlslib/tree/master/tb

©BOLEDU

HLS Testbench



```
// ../../src/finn-hlslib/tb/conv3_tb.cpp#L105-L128

int main()
{
    //create_memdata();
    static  ap_uint<INPUT_PRECISION> IMAGE[MAX_IMAGES][IFMDim1*IFMDim1][IFM_Channels1];
    static  ap_uint<ACTIVATION_PRECISION> TEST[MAX_IMAGES][OFMDim1][OFMDim1][OFM_Channels1];
    stream<ap_uint<IFM_Channels1*INPUT_PRECISION> > input_stream("input_stream");
    stream<ap_uint<OFM_Channels1*ACTIVATION_PRECISION> > output_stream("output_stream");
    unsigned int counter = 0;
    for (unsigned int n_image = 0; n_image < MAX_IMAGES; n_image++) {
        for (unsigned int oy = 0; oy < IFMDim1; oy++) {
            for (unsigned int ox = 0; ox < IFMDim1; ox++) {
                ap_uint<INPUT_PRECISION*IFM_Channels1> input_channel = 0;
                for(unsigned int channel = 0; channel < IFM_Channels1; channel++)
                {
                    ap_uint<INPUT_PRECISION> input = (ap_uint<INPUT_PRECISION>)(counter);
                    IMAGE[n_image][oy*IFMDim1+ox][channel]= input;
                    input_channel = input_channel >> INPUT_PRECISION;
                    input_channel(IFM_Channels1*INPUT_PRECISION-1,(IFM_Channels1-1)*INPUT_PRECISION)=input;
                    counter++;
                }
                input_stream.write(input_channel);
            }
        }
    }
```



```
static  ap_uint<WIDTH> W1[OFM_Channels1][KERNEL_DIM][KERNEL_DIM][IFM_Channels1];
// initialize the weights
constexpr int TX = (IFM_Channels1*KERNEL_DIM*KERNEL_DIM) / SIMD1;
constexpr int TY = OFM_Channels1 / PE1;
unsigned int kx=0;
unsigned int ky=0;
unsigned int chan_count=0;
unsigned int out_chan_count=0;

for (unsigned int oy = 0; oy < TY; oy++) {
    for(unsigned int pe=0;pe <PE1;pe++){
        for (unsigned int ox = 0; ox <TX; ox++) {
            for(unsigned int simd=0;simd<SIMD1;simd++){
                W1[out_chan_count][kx][ky][chan_count] = PARAM::weights.weights(oy*TX + ox)[pe][simd];
                //cout << "TILE " << oy*TX + ox << " PE " << pe << " SIMD " << simd << endl;
                //cout << "IFM " << chan_count << " KX " << kx << " KY " << ky << " OFM " << out_chan_count << endl;
                chan_count++;
                if (chan_count==IFM_Channels1){
                    chan_count=0;
                    kx++;
                    if (kx==KERNEL_DIM){
                        kx=0;
                        ky++;
                        if (ky==KERNEL_DIM){
                            ky=0;
                            out_chan_count++;
                            if (out_chan_count==OFM_Channels1){
                                out_chan_count=0;
                            }
                        }
                    }
                }
            }

 conv<MAX_IMAGES,IFMDim1,OFMDim1,IFM_Channels1,OFM_Channels1, KERNEL_DIM, 1, ap_uint<INPUT_PRECISION> >(IMAGE, W1, TEST);
```

Debuging with Vitis_hls

- After C synthesis (FINN hardware build flow)
  - There is a vitis_hls project file in the directory
  - Directly open it and add testbench from github
- See textbooks for detail