Bridge of Life Education

FINN Compiler

Lecturer: Hua-Yang Weng

Date: 2022/4/27

[FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks]

[FPGA'17: FINN: A Framework for Fast, Scalable Binarized Neural Network Inference]
(https://arxiv.org/abs/1612.07119)

From AI to Gate Textbook

Bridge of Life Education

Signed in as:
翁華揚

Textbooks | Dashboard | Sign out

From AI to Gate

- Preface
- Getting Started
- Network Define
- Compiler
  - **Introduction**
  - Tidy-up and Preprocess
  - Streamlining
  - HLS Dataflow
  - Hardware Build
- Verification
- NN Hardware
- HLS
- Case Study
- Code Repository
- Translations

Compiler

In this chapter, we are going to explain how FINN maps the deep learning network from an exported ONNX graph into high-level-synthesized layers. From the Wikipedia definition, the name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g. assembly language, object code, or machine code) to create an executable program. However, what the FINN Compiler here actually transforms is **from high-level operation representations such as ONNX graph, into another operation representation that is compatible with some high-level-synthesis libraries**. The later will perform C/C++ function-calls to the HLS library and then be synthesized by another High-Level-Synthesis compiler such as Vitis-HLS (into hardware representations and then the bitstream file for FPGAs).

Goals

The figure below shows our objective in this chapter. As chapter one has shown, we are going to transform the onnx graph from the left to the right. We can see that all the operations are now mapped into hardware operations supported by certain HLS hardware unit.

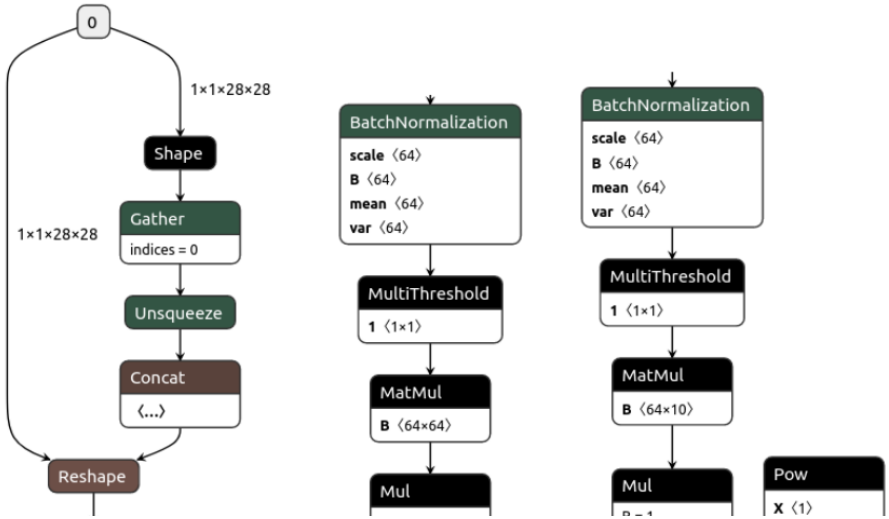


Table of Contents

- **Compiler**
- Goals
- Chapter structure

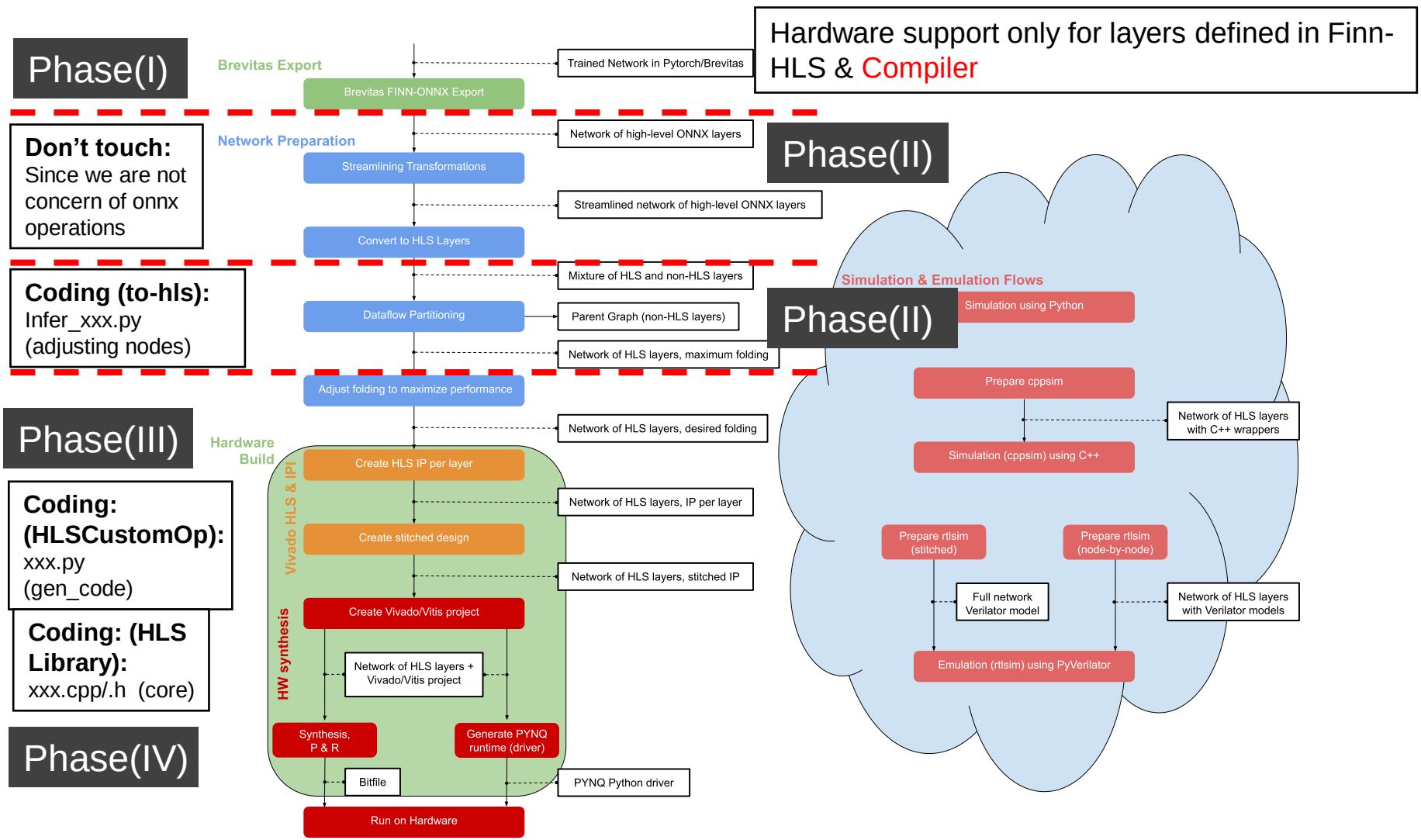
Overview

- End-to-End Compiling
- Phase (I): Brevitas export
- Phase (II):
  - Network preparation
  - Conversion to HLS layers
- Phase (III): Hardware Build
- Phase (IV): PYNQ deployment

Overview

- End-to-End Compiling
- Phase (I): Brevitas export
- Phase (II):
  - Network preparation
  - Conversion to HLS layers
- Phase (III): Hardware Build
- Phase (IV): PYNQ deployment

Recap: End2End Flow



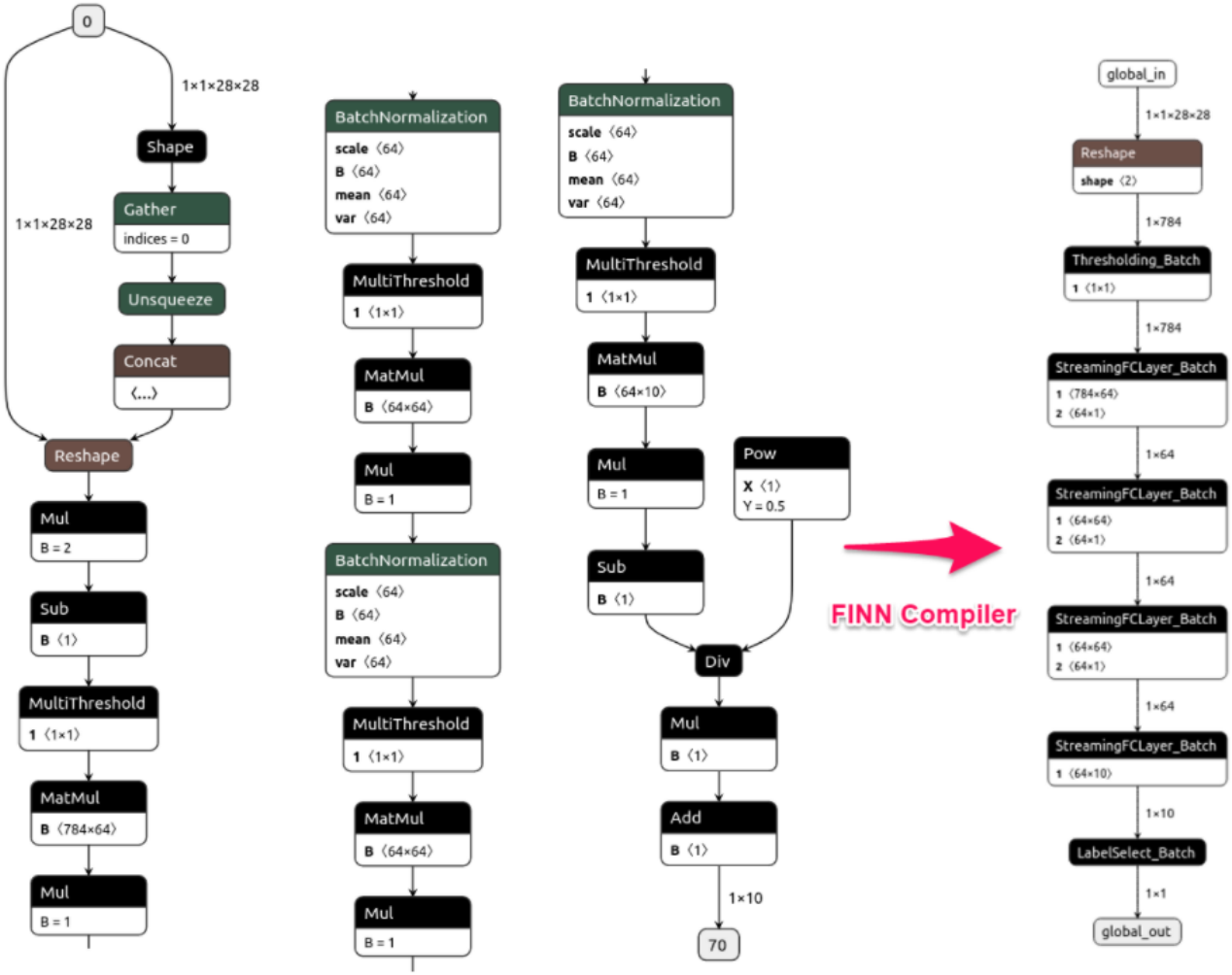
https://github.com/Xilinx/finn/blob/main/notebooks/end2end_example/bnn-pynq/tfc_end2end_example.ipynb

Introduction

- The FINN compiler comes with many *transformations* that modify the ONNX representation of the network according to certain patterns.

- Phase (I): Brevitas export
- Phase (II):
  - Network preparation
  - Conversion to HLS layers
- Phase (III): Hardware Build
- Phase (IV): PYNQ deployment

Goal


0
1×1×28×28
Shape
1×1×28×28
Gather
indices = 0
Unsqueeze
Concat
⟨...⟩
Reshape
Mul
B = 2
Sub
B ⟨1⟩
MultiThreshold
1 ⟨1×1⟩
MatMul
B ⟨784×64⟩
Mul
B = 1
BatchNormalization
scale ⟨64⟩
B ⟨64⟩
mean ⟨64⟩
var ⟨64⟩
MultiThreshold
1 ⟨1×1⟩
MatMul
B ⟨64×64⟩
Mul
B = 1
BatchNormalization
scale ⟨64⟩
B ⟨64⟩
mean ⟨64⟩
var ⟨64⟩
MultiThreshold
1 ⟨1×1⟩
MatMul
B ⟨64×64⟩
Mul
B = 1
BatchNormalization
scale ⟨64⟩
B ⟨64⟩
mean ⟨64⟩
var ⟨64⟩
MultiThreshold
1 ⟨1×1⟩
MatMul
B ⟨64×10⟩
Mul
B = 1
Sub
B ⟨1⟩
Pow
X ⟨1⟩
Y = 0.5
Div
Mul
B ⟨1⟩
Add
B ⟨1⟩
1×10
70
FINN Compiler
global_in
1×1×28×28
Reshape
shape ⟨2⟩
1×784
Thresholding_Batch
1 ⟨1×1⟩
1×784
StreamingFCLayer_Batch
1 ⟨784×64⟩
2 ⟨64×1⟩
1×64
StreamingFCLayer_Batch
1 ⟨64×64⟩
2 ⟨64×1⟩
1×64
StreamingFCLayer_Batch
1 ⟨64×64⟩
2 ⟨64×1⟩
1×64
StreamingFCLayer_Batch
1 ⟨64×10⟩
1×10
LabelSelect_Batch
1×1
global_out

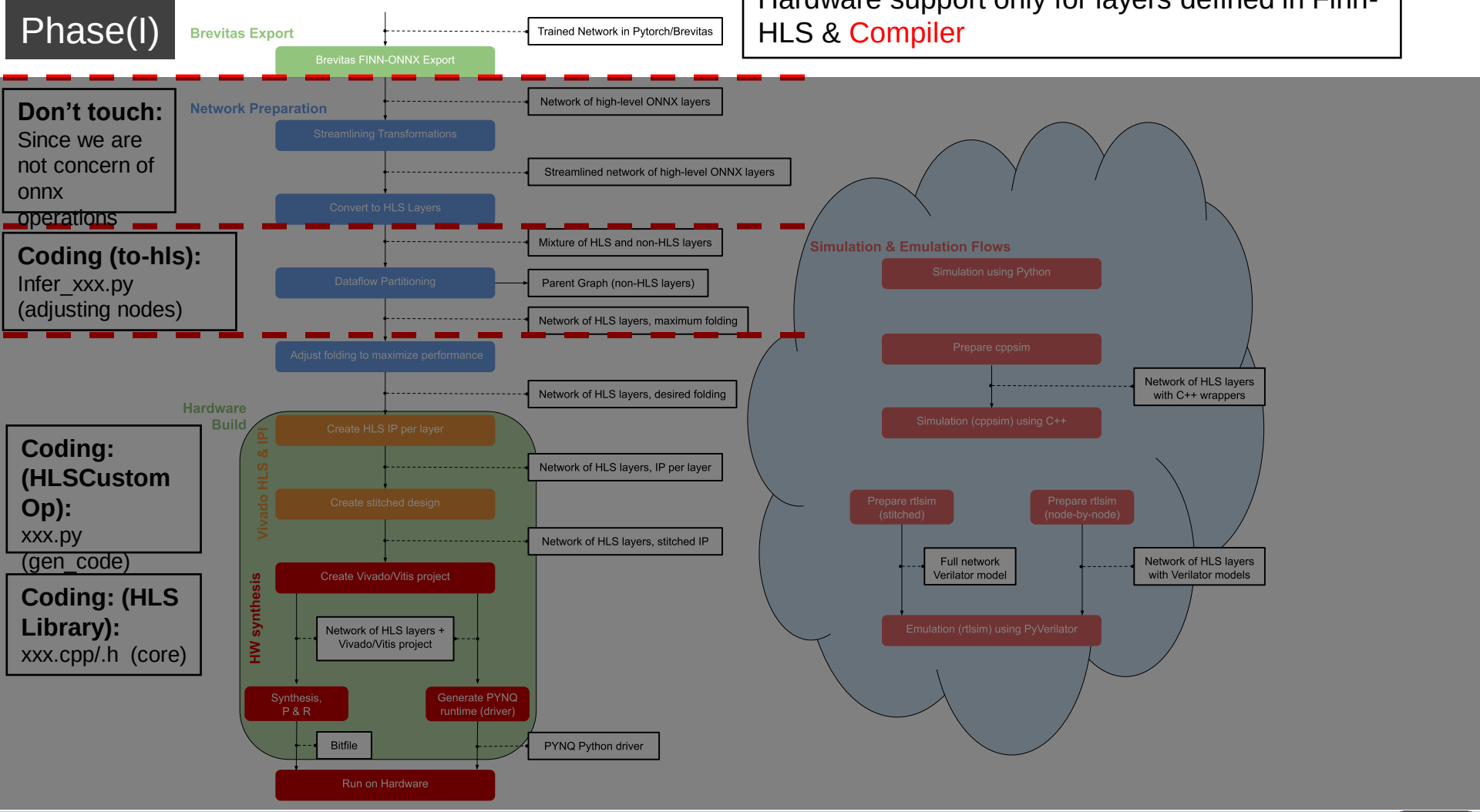
Overview

- End-to-End Compiling
- Phase (I): Brevitas export
- Phase (II):
  - Network preparation
  - Conversion to HLS layers
- Phase (III): Hardware Build
- Phase (IV): PYNQ deployment

Phase (I): Brevitas Export

Phase(I)

Hardware support only for layers defined in Finn-HLS & Compiler



Don't touch: Since we are not concern of onnx operations

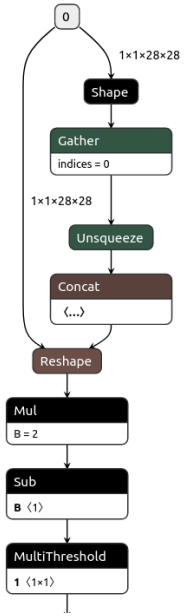
Coding (to-hls): Infer_xxx.py (adjusting nodes)

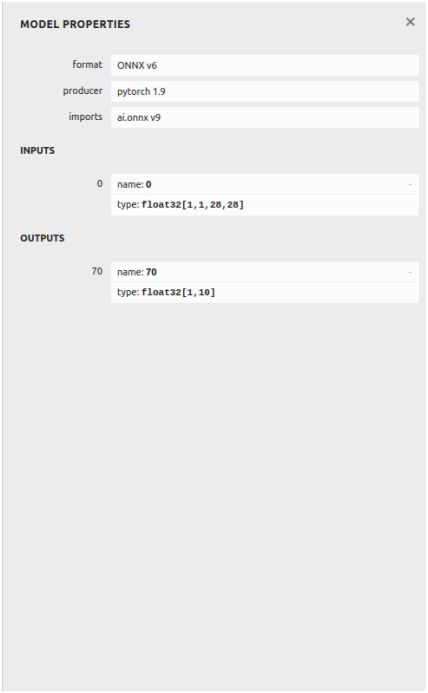
Coding: (HLSCustom Op): xxx.py (gen_code)

Coding: (HLS Library): xxx.cpp/.h (core)

Phase (I): Brevitas Export (1/2)

```
import onnx
from finn.util.test import get_test_model_trained
import brevitas.onnx as bo

tfc = get_test_model_trained("TFC", 1, 1)
bo.export_finn_onnx(tfc, (1, 1, 28, 28), build_dir+"/tfc_w1_a1.onnx")
```





Phase (I): Brevitas Export (2/2)

- **ModelWrapper**
  - Wrapper around the ONNX model which provides several helper functions to make it easier to work with the model.

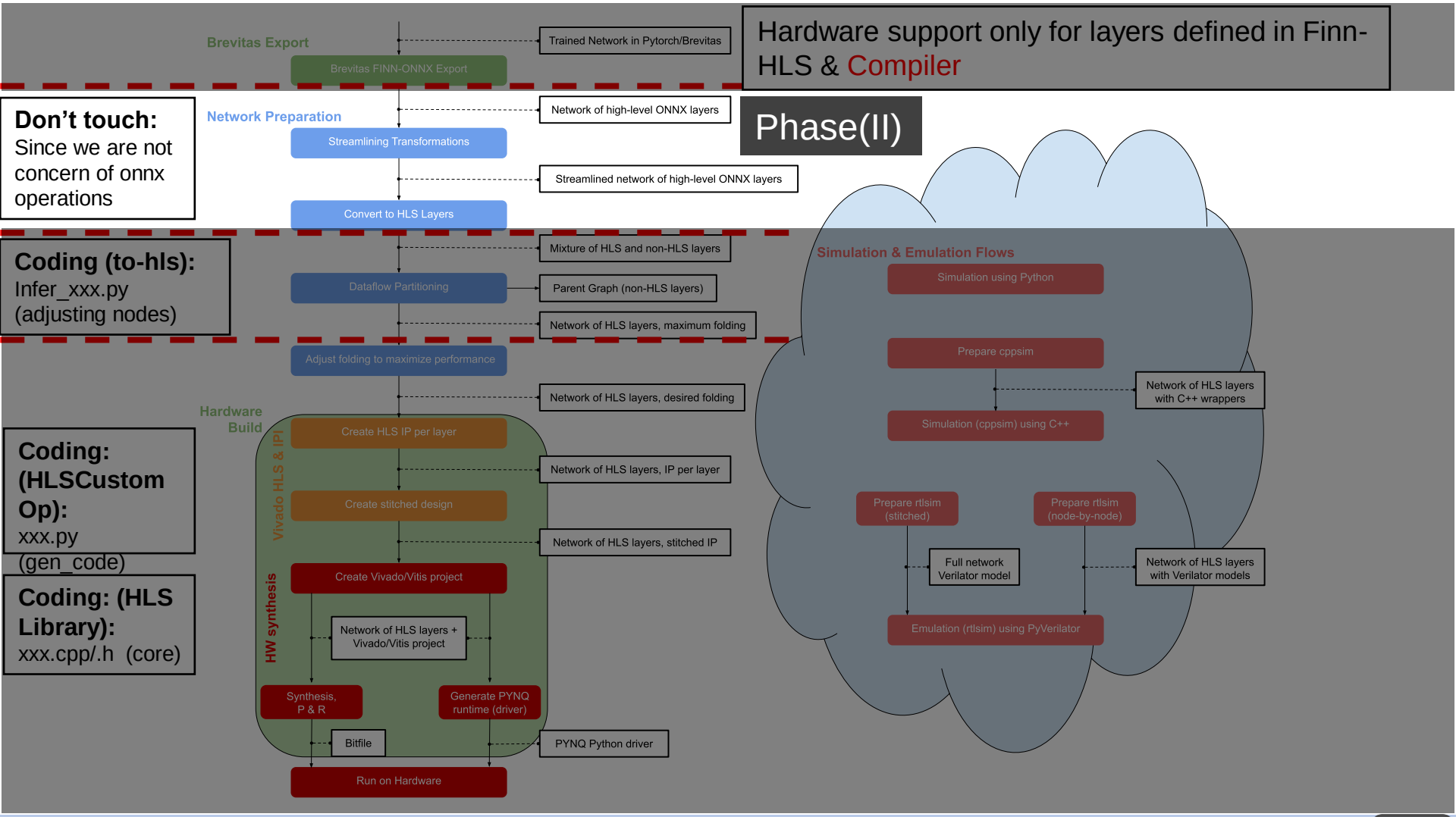
```
from finn.core.modelwrapper import ModelWrapper
model = ModelWrapper(build_dir+"/tfc_w1_a1.onnx")
```

- Principle of FINN: **Analysis** and **Transformation passes**
  - **Analysis pass:** extracts specific information about the model.
  - **Transformation pass:** changes the model and returns the changed model

Overview

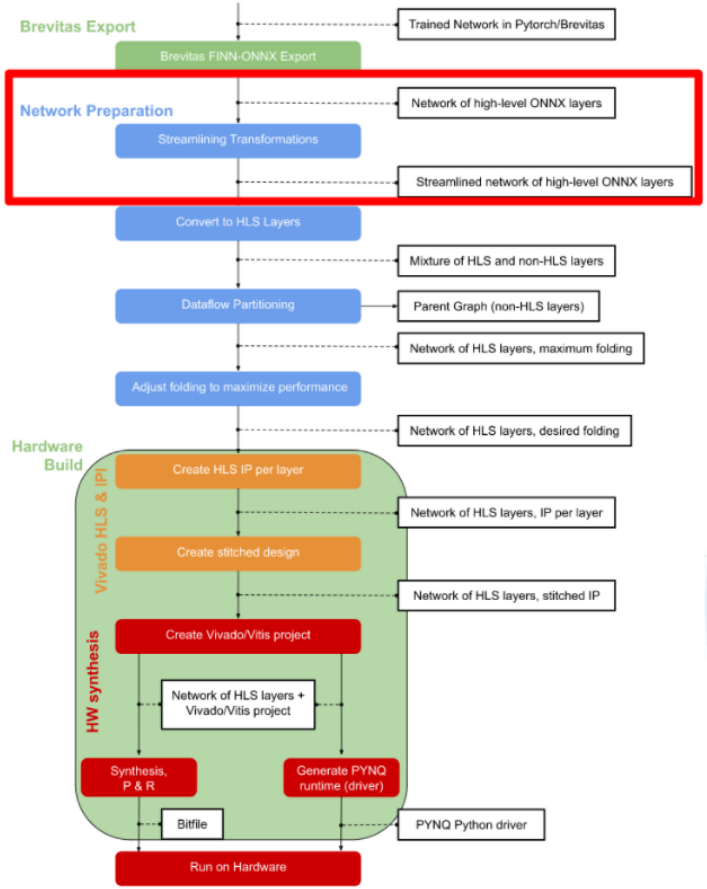
- End-to-End Compiling
- Phase (I): Brevitas export
- Phase (II):
  - Network preparation
  - Conversion to HLS layers
- Phase (III): Hardware Build
- Phase (IV): PYNQ deployment

Phase (II): Network preparation

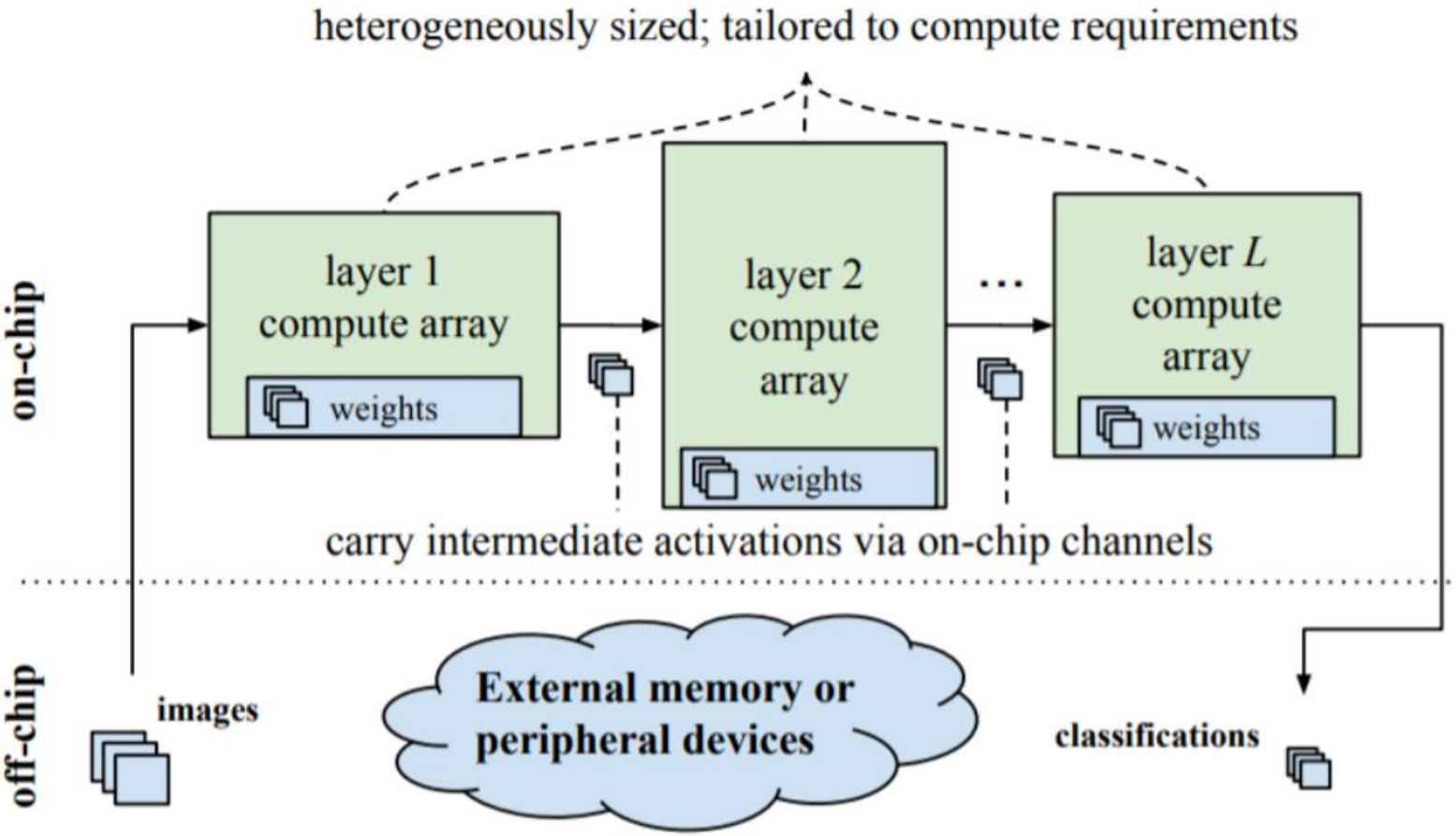


Phase (II): Network preparation

- (I) FINN-style Dataflow Architectures
- (II) Tidy-up transformations
- (III) Adding Pre- and Postprocessing
- (IV) Streamlining
- (V) Conversion to HLS layers
- (VI) Creating a Dataflow Partition
- (VII) Folding: Adjusting the Parallelism



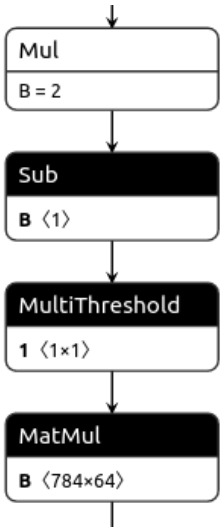
(I) FINN-style Dataflow Architectures

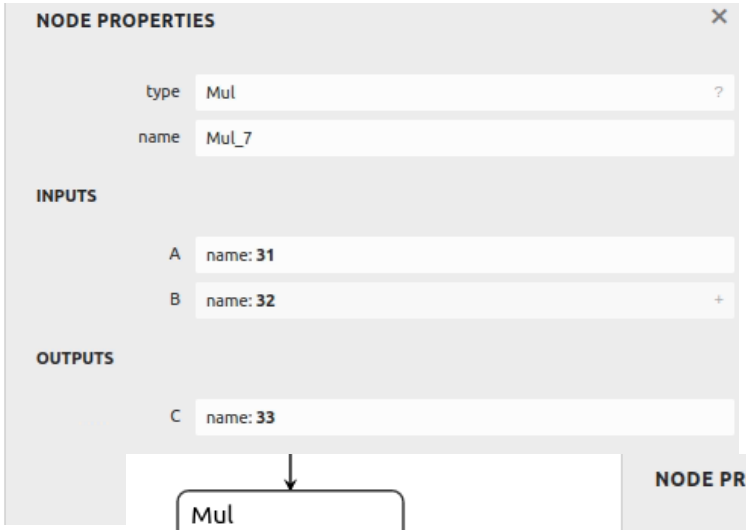


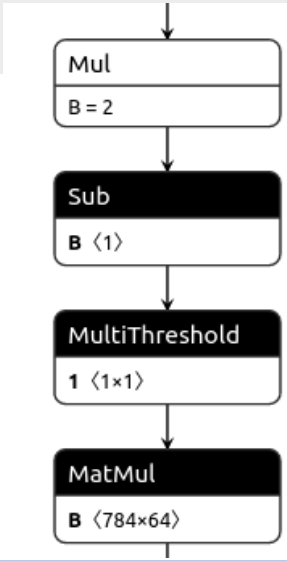
(II) Tidy-up transformations

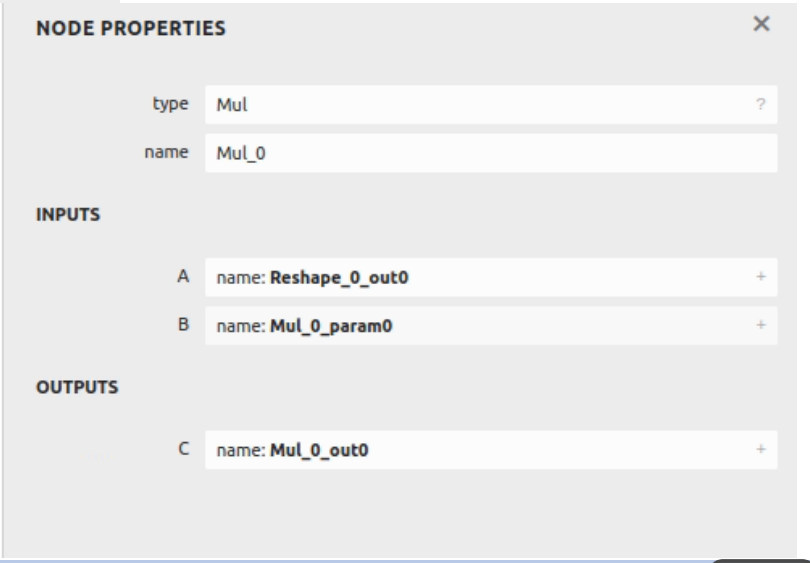
- GiveUniqueNodeNames
- GiveReadableTensorNames
- InferShapes
- InferDataTypes
- FoldConstants
- RemoveStaticGraphInputs

GiveUniqueNodeNames / GiveReadableTensorNames

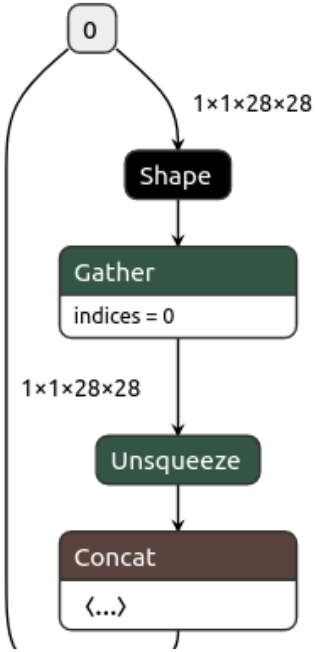








InferShapes / InferDataTypes



MODEL PROPERTIES

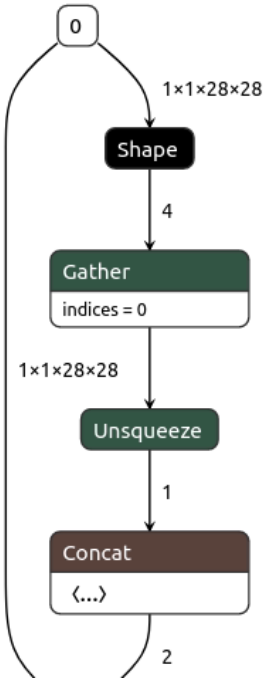
format	ONNX v6
producer	pytorch 1.9
imports	ai.onnx v9

INPUTS

0	name: **0**
	type: **float32[1,1,28,28]**

OUTPUTS

70	name: **70**
	type: **float32[1,10]**



MODEL PROPERTIES

format	ONNX v6
producer	pytorch 1.9
imports	ai.onnx v9

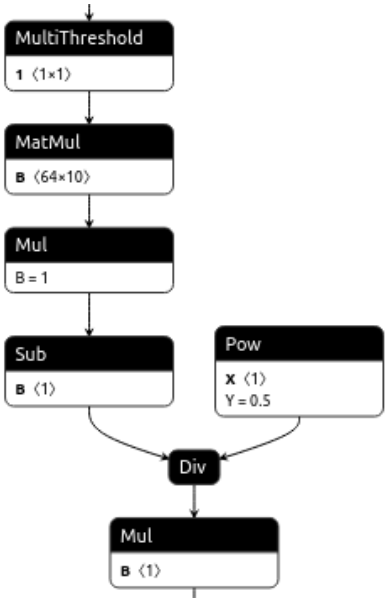
INPUTS

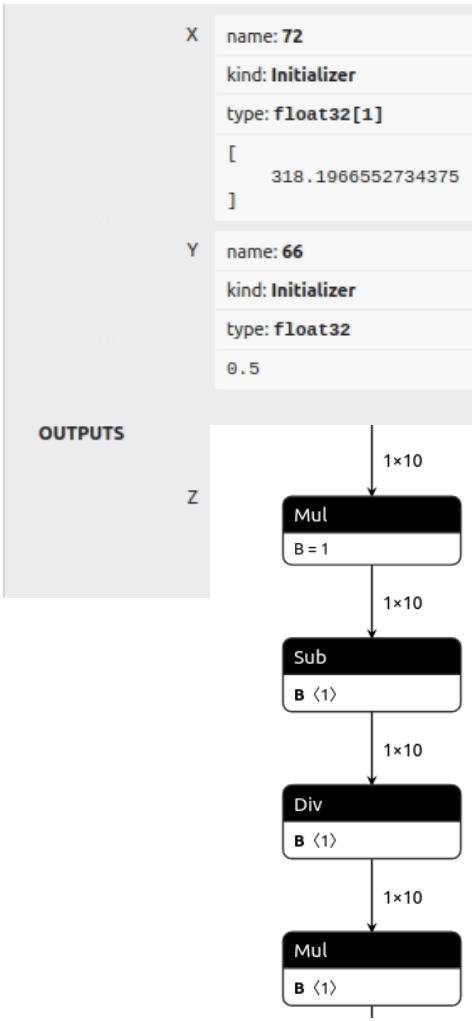
0	name: **0**
	type: **float32[1,1,28,28]**

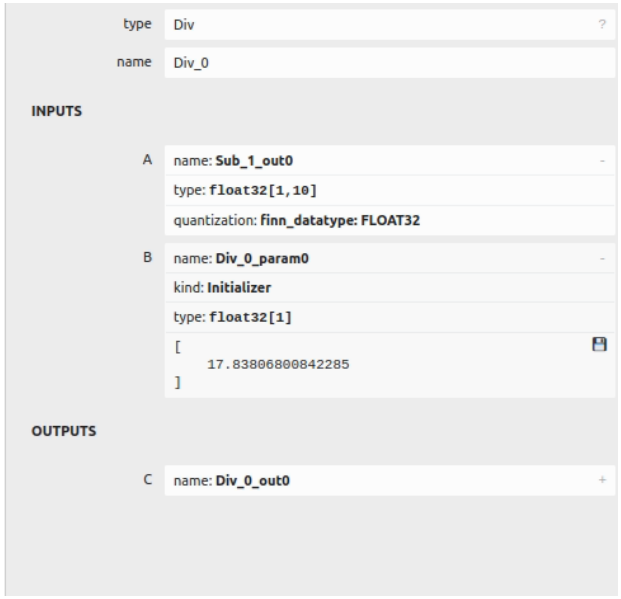
OUTPUTS

70	name: **70**
	type: **float32[1,10]**
	quantization: **finn_datatype: FLOAT32**

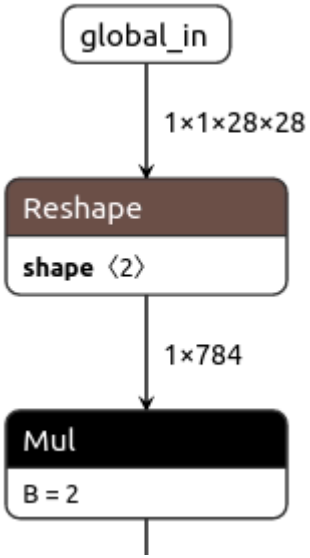
FoldConstants

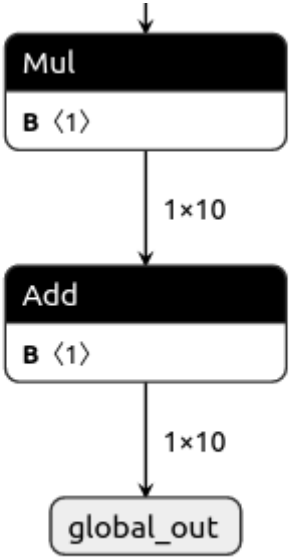


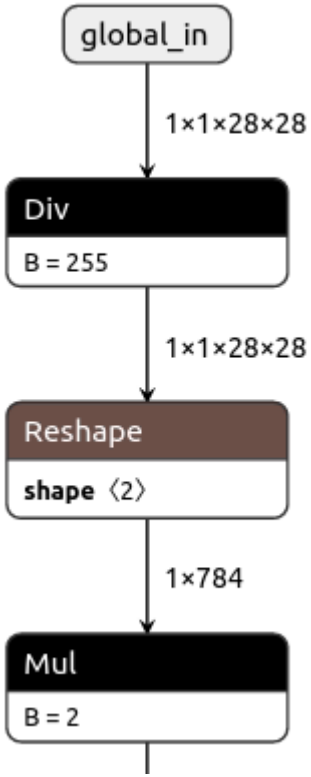


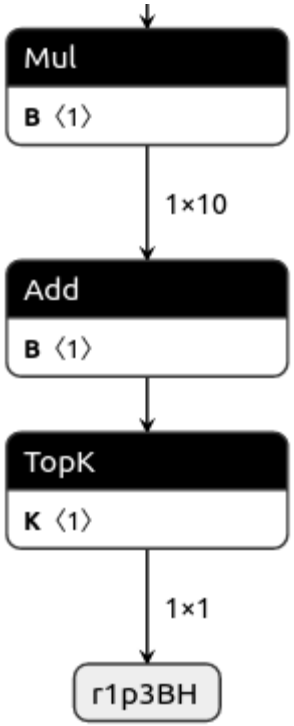


(III) Adding Pre- and Postprocessing









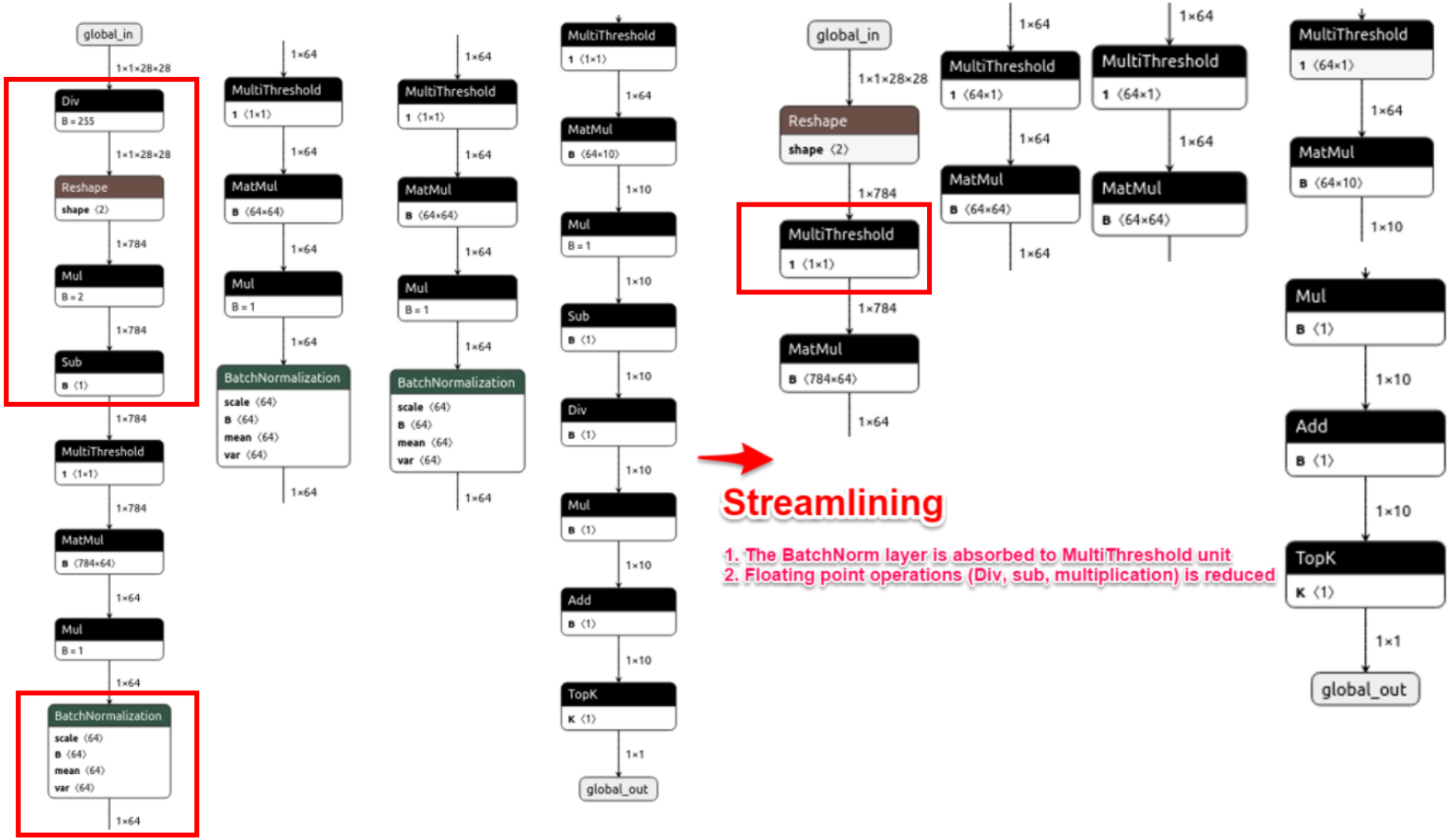
(IV) Streamlining(1/4)

- **Goal:** eliminate, collapsing floating point operations

```
from finn.transformation.streamline.reorder import MoveScalarLinearPastInvariants
import finn.transformation.streamline.absorb as absorb

model = ModelWrapper(build_dir+"/tfc_w1_a1_pre_post.onnx")
# move initial Mul (from preproc) past the Reshape
model = model.transform(MoveScalarLinearPastInvariants())
# streamline
model = model.transform(Streamline())
model.save(build_dir+"/tfc_w1_a1_streamlined.onnx")
showInNetron(build_dir+"/tfc_w1_a1_streamlined.onnx")
```

```
class Streamline(Transformation):
    """Apply the streamlining transform, see arXiv:1709.04060."""

    def apply(self, model):
        streamline_transformations = [
            ConvertSubToAdd(),
            ConvertDivToMul(),
            BatchNormToAffine(),
            ConvertSignToThres(),
            AbsorbSignBiasIntoMultiThreshold(),
            MoveAddPastMul(),
            MoveScalarAddPastMatMul(),
            MoveAddPastConv(),
            MoveScalarMulPastMatMul(),
            MoveScalarMulPastConv(),
            MoveAddPastMul(),
            CollapseRepeatedAdd(),
            CollapseRepeatedMul(),
            AbsorbAddIntoMultiThreshold(),
            FactorOutMulSignMagnitude(),
            AbsorbMulIntoMultiThreshold(),
            Absorb1BitMulIntoMatMul(),
            Absorb1BitMulIntoConv(),
            RoundAndClipThresholds(),
        ]
        for trn in streamline_transformations:
            model = model.transform(trn)
            model = model.transform(RemoveIdentityOps())
            model = model.transform(GiveUniqueNodeNames())
            model = model.transform(GiveReadableTensorNames())
            model = model.transform(InferDataTypes())
        return (model, False)
```

(IV) Streamlining(2/4)

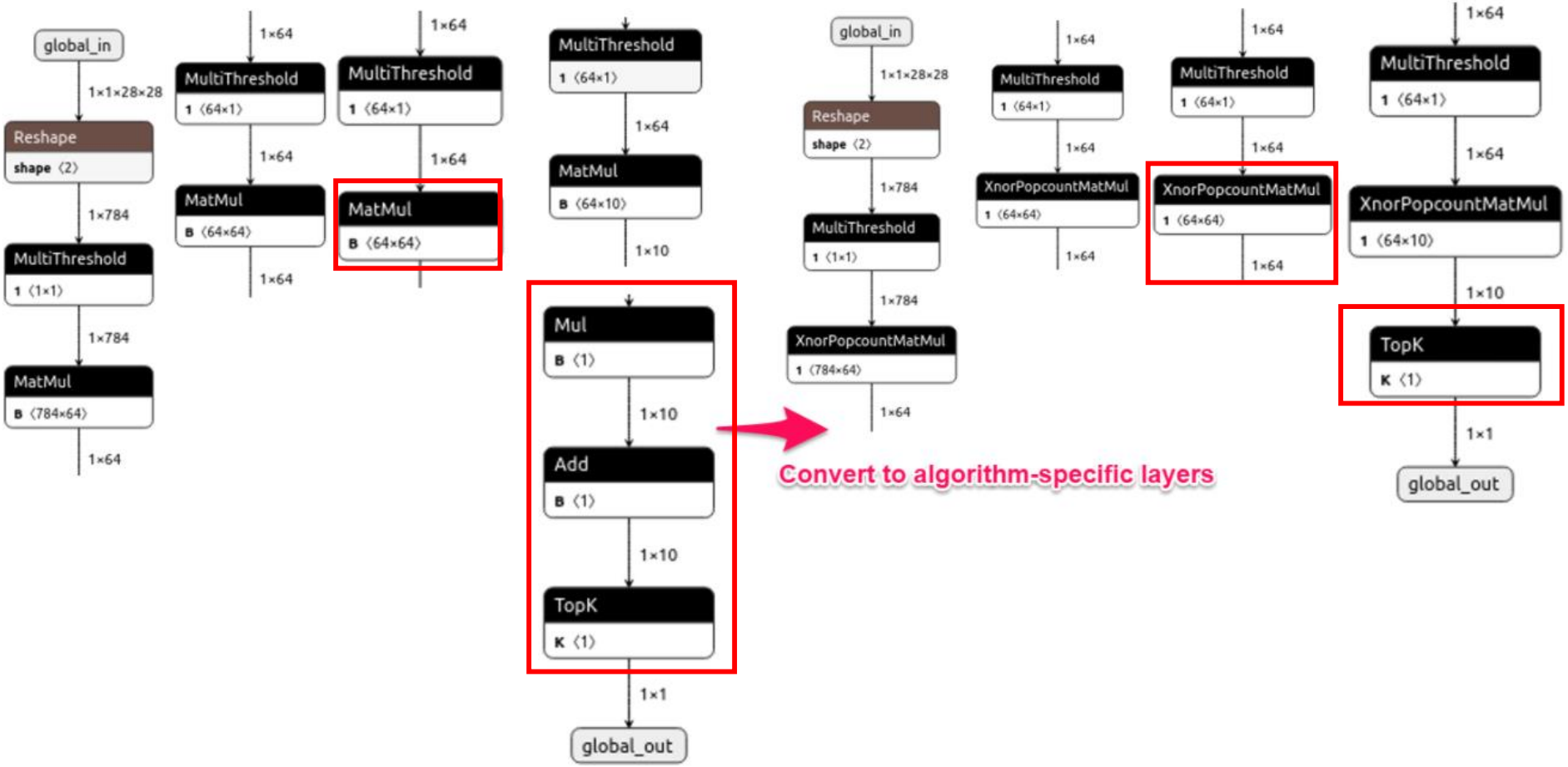


(IV) Streamlining(3/4)

- Current implementation of streamlining:
  - Highly network-specific (topology change)

```
from finn.transformation.bipolar_to_xnor import ConvertBipolarMatMulToXnorPopcount
from finn.transformation.streamline.round_thresholds import RoundAndClipThresholds
from finn.transformation.infer_data_layouts import InferDataLayouts
from finn.transformation.general import RemoveUnusedTensors

model = model.transform(ConvertBipolarMatMulToXnorPopcount())
model = model.transform(absorb.AbsorbAddIntoMultiThreshold())
model = model.transform(absorb.AbsorbMulIntoMultiThreshold())
# absorb final add-mul nodes into TopK
model = model.transform(absorb.AbsorbScalarMulAddIntoTopK())
model = model.transform(RoundAndClipThresholds())

# bit of tidy-up
model = model.transform(InferDataLayouts())
model = model.transform(RemoveUnusedTensors())

model.save(build_dir+"/tfc_w1a1_ready_for_hls_conversion.onnx")
showInNetron(build_dir+"/tfc_w1a1_ready_for_hls_conversion.onnx")
```

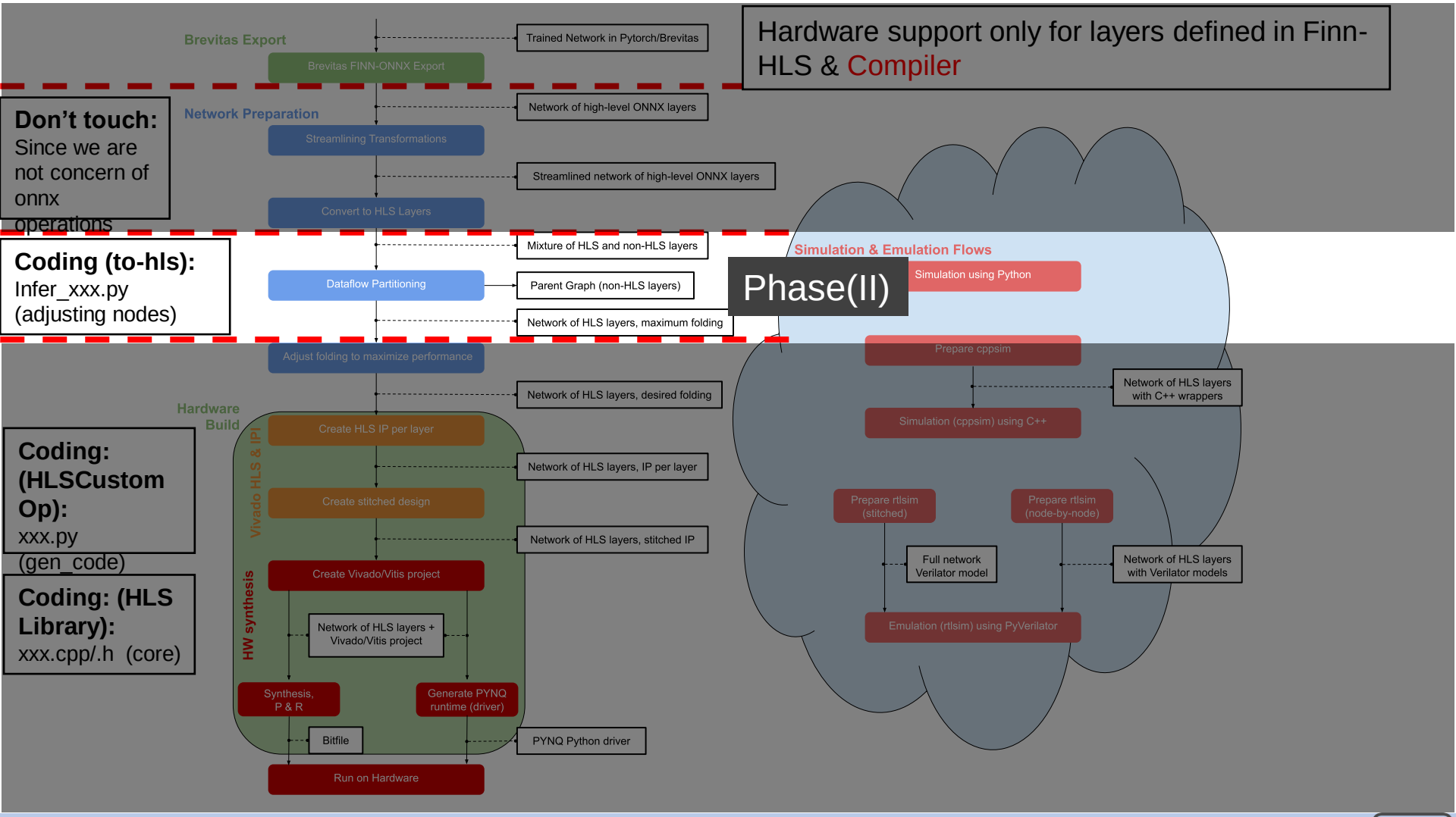
(IV) Streamlining(4/4)



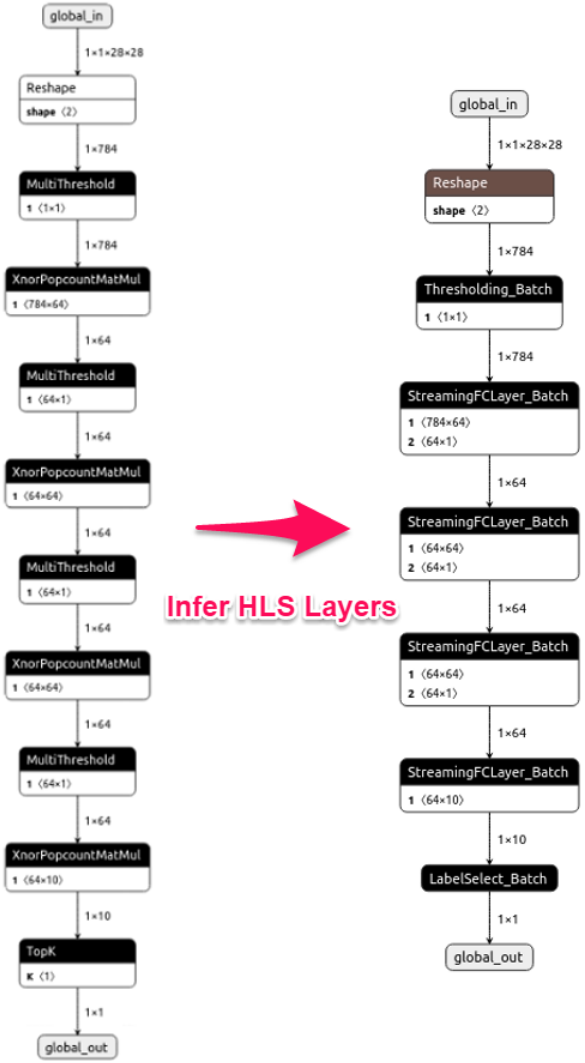
Overview

- End-to-End Compiling
- Phase (I): Brevitas export
- Phase (II):
  - Network preparation
  - Conversion to HLS layers
- Phase (III): Hardware Build
- Phase (IV): PYNQ deployment

Phase (II): (V) Conversion to HLS layers



(V) Conversion to HLS layers



NODE PROPERTIES

type	StreamingFCLayer_Batch
domain	finn.custom_op.fpgadataflow

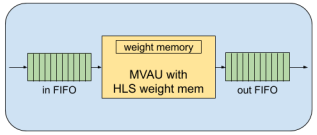
ATTRIBUTES

accDataType	UINT10
ActVal	0
backend	fpgadataflow
binaryXnorMode	1
inputDataType	BINARY
mem_mode	decoupled
MH	64
MW	784
noActivation	0
numInputVectors	1
outputDataType	BINARY
PE	1
SIMD	1
weightDataType	BINARY

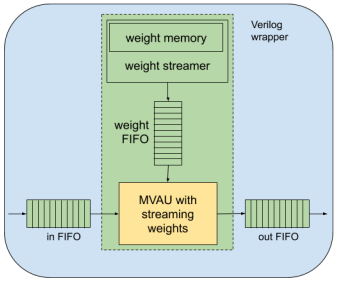
INPUTS

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

OUTPUTS



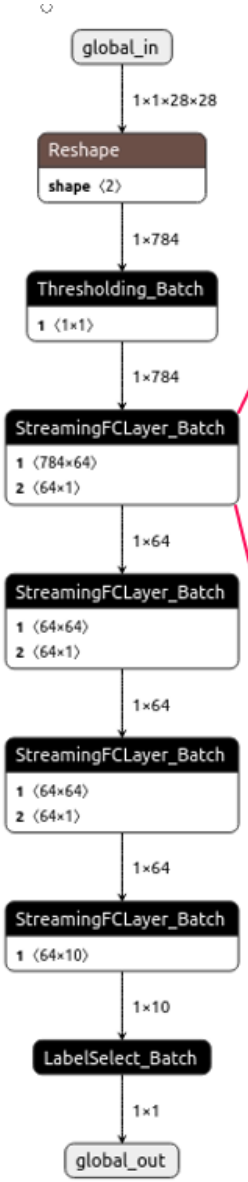
mem_mode = "const"



mem_mode = "decoupled"

HLS components

Verilog components



NODE PROPERTIES

type	StreamingFCLayer_Batch
domain	finn.custom_op.fpgadataflow

ATTRIBUTES

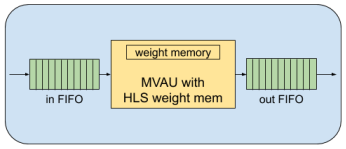
accDataType	UINT10
ActVal	0
backend	fpgadataflow
binaryXnorMode	1
inputDataType	BINARY
mem_mode	decoupled
MH	64
MW	784
noActivation	0
numInputVectors	1
outputDataType	BINARY
PE	1
SIMD	1
weightDataType	BINARY

INPUTS

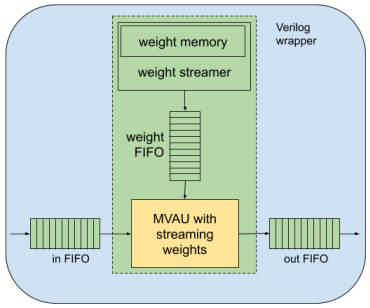
0	name: **MultiThreshold_0_out0**
1	name: **MatMul_0_param0**
2	name: **MultiThreshold_1_param0**

OUTPUTS

XILINX FINN



mem_mode = "const"



mem_mode = "decoupled"

mem_mode:

- const: weights are "baked in" into the HLS code
- decoupled: weights are streamed into the core using streamers and fifos

(See FINN doc for details: https://finn.readthedocs.io/en/latest/internals.html)

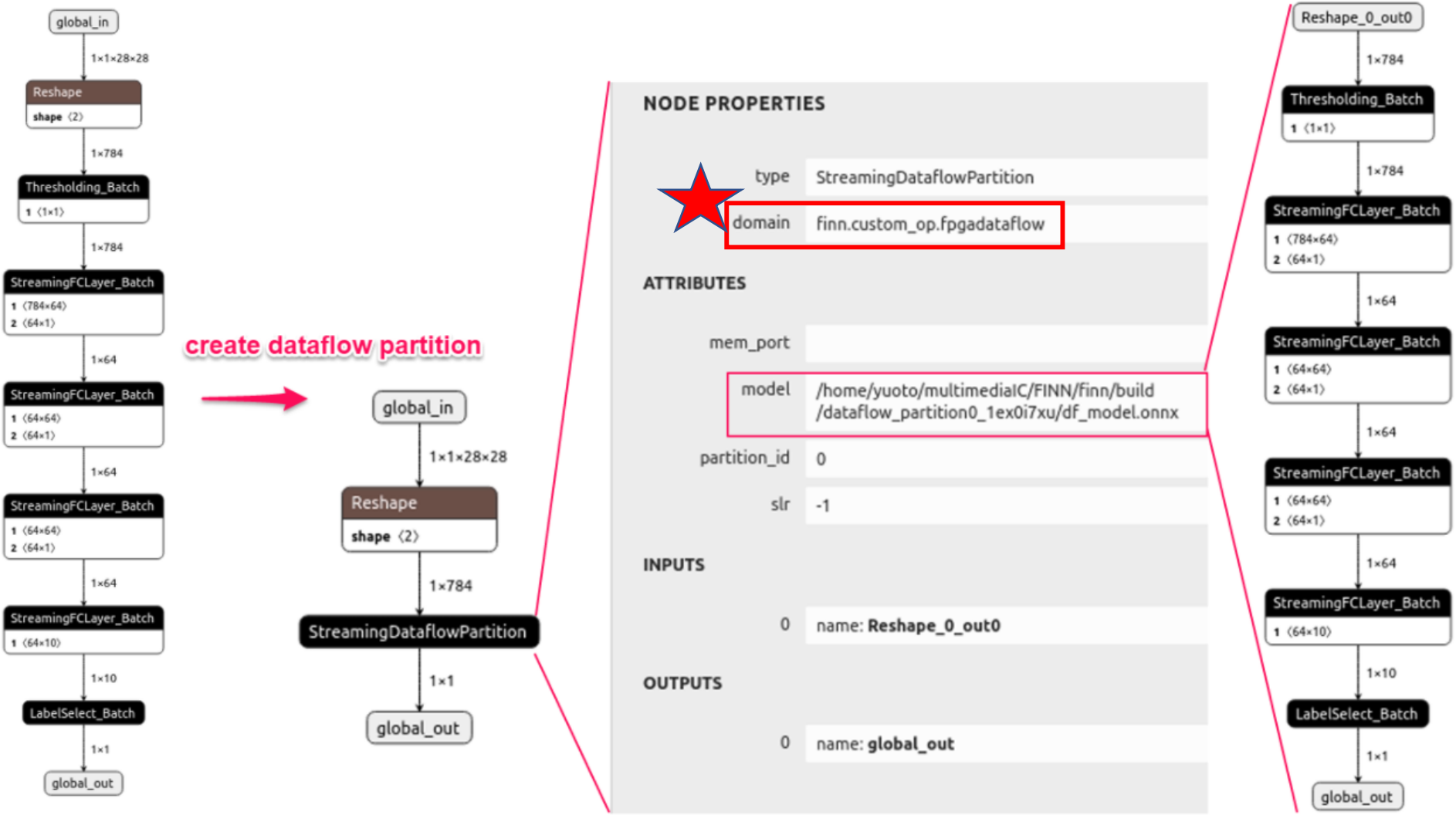
(V) Conversion to HLS layers: Infer_xxx.py

```
import finn.transformation.fpgadataflow.convert_to_hls_layers as to_hls
model = ModelWrapper(build_dir+"/tfc_w1a1_ready_for_hls_conversion.onnx")
model = model.transform(to_hls.InferBinaryStreamingFCLayer("decoupled"))
# TopK to LabelSelect
model = model.transform(to_hls.InferLabelSelectLayer())
# input quantization (if any) to standalone thresholding
model = model.transform(to_hls.InferThresholdingLayer())
model.save(build_dir+"/tfc_w1_a1_hls_layers.onnx")
showInNetron(build_dir+"/tfc_w1_a1_hls_layers.onnx")
```

```
class InferBinaryStreamingFCLayer(Transformation):
    """Convert XnorPopcountMatMul layers to
    StreamingFCLayer_Batch layers. Any immediately following MultiThreshold
    layers will also be absorbed into the MVTU."""

    def __init__(self, mem_mode="const"):
        super().__init__()
        self.mem_mode = mem_mode

    def apply(self, model):
        graph = model.graph
        node_ind = 0
        graph_modified = False
        for n in graph.node:
            node_ind += 1
            if n.op_type == "XnorPopcountMatMul":
                mm_input = n.input[0]
                mm_weight = n.input[1]
                mm_output = n.output[0]
                mm_in_shape = model.get_tensor_shape(mm_input)
                mm_out_shape = model.get_tensor_shape(mm_output)
                assert (
                    model.get_tensor_datatype(mm_input) == DataType.BINARY
                ), """First
```

```
# create and insert new StreamingFCLayer node
new_node = helper.make_node(
    "StreamingFCLayer_Batch",
    [mm_input, mm_weight, mt_thres],
    [mt_output],
    domain="finn.custom_op.fpgadataflow",
    backend="fpgadataflow",
    MW=mw,
    MH=mh,
    SIMD=simd,
    PE=pe,
    inputDataType=idt.name,
    weightDataType=wdt.name,
    outputDataType=odt.name,
    ActVal=actval,
    binaryXnorMode=1,
    noActivation=0,
    numInputVectors=list(mm_in_shape[:-1]),
    mem_mode=self.mem_mode,
)
graph.node.insert(node_ind, new_node)
# remove old nodes
```

(VI) Creating a Dataflow Partition



(VII) Folding: Adjusting the Parallelism (1/3)

- Manually set the folding factors and FIFO depths

- Automatically tune parameters:
  - Given an FPGA resource budget
  - Analytical model from the FINN-R paper.

```
CustomOp wrapper is of class Thresholding_Batch
{'PE': ('i', True, 0),
 'NumChannels': ('i', True, 0),
 'ram_style': ('s', False, 'distributed'),
 'inputDataType': ('s', True, ''),
 'outputDataType': ('s', True, ''),
 'inFIFODepth': ('i', False, 2),
 'outFIFODepth': ('i', False, 2),
 'numInputVectors': ('ints', False, [1]),
 'ActVal': ('i', False, 0),
 'backend': ('s', True, 'fpgadataflow'),
 'code_gen_dir_cppsim': ('s', False, ''),
 'code_gen_dir_ipgen': ('s', False, ''),
 'executable_path': ('s', False, ''),
 'ipgen_path': ('s', False, ''),
 'ip_path': ('s', False, ''),
 'ip_vlnv': ('s', False, ''),
 'exec_mode': ('s', False, ''),
 'cycles_rtlsim': ('i', False, 0),
 'cycles_estimate': ('i', False, 0),
 'rtlsim_trace': ('s', False, ''),
 'res_estimate': ('s', False, ''),
 'res_hls': ('s', False, ''),
 'res_synth': ('s', False, ''),
 'rtlsim_so': ('s', False, ''),
 'partition_id': ('i', False, 0)}
```

(VII) Folding: Adjusting the Parallelism (2/3)

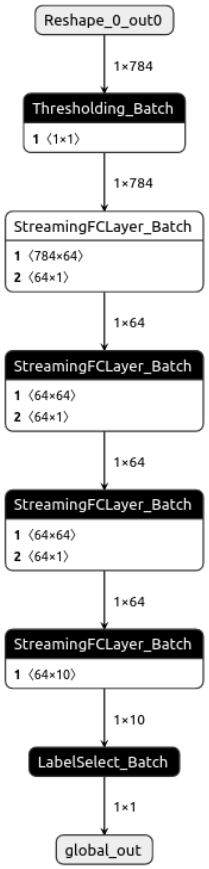
- Below:

Will be added later with 'InsertFIFO()' Transformation

Set the PE and SIMD s.t. II = 64 for each layer

```
fc_layers = model.get_nodes_by_op_type("StreamingFCLayer_Batch")
# (PE, SIMD, in_fifo_depth, out_fifo_depth, ramstyle) for each layer
config = [
    (16, 49, 16, 64, "block"),
    (8, 8, 64, 64, "auto"),
    (8, 8, 64, 64, "auto"),
    (10, 8, 64, 10, "distributed"),
]
for fcl, (pe, simd, ififo, ofifo, ramstyle) in zip(fc_layers, config):
    fcl_inst = getCustomOp(fcl)
    fcl_inst.set_nodeattr("PE", pe)
    fcl_inst.set_nodeattr("SIMD", simd)
    fcl_inst.set_nodeattr("inFIFODepth", ififo)
    fcl_inst.set_nodeattr("outFIFODepth", ofifo)
    fcl_inst.set_nodeattr("ram_style", ramstyle)

# set parallelism for input quantizer to be same as first layer's SIMD
inp_qnt_node = model.get_nodes_by_op_type("Thresholding_Batch")[0]
inp_qnt = getCustomOp(inp_qnt_node)
inp_qnt.set_nodeattr("PE", 49)
```

(VII) Folding: Adjusting the Parallelism (3/3)



NODE PROPERTIES

type	StreamingFCLayer_Batch
module	finn.custom_op.fpgadataflow

ATTRIBUTES

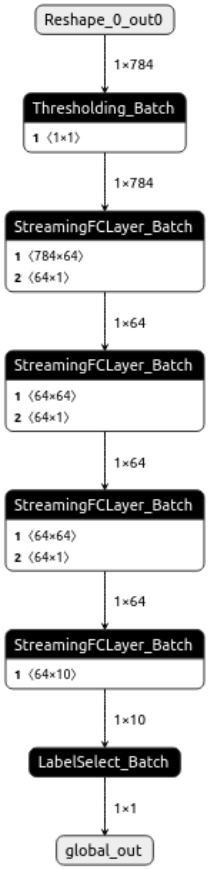
accDataType	UINT10
ActVal	0
backend	fpgadataflow
binaryXnorMode	1
inputDataType	BINARY
mem_mode	decoupled
MH	64
MW	784
noActivation	0
numInputVectors	1
outputDataType	BINARY
PE	1
SIMD	1
weightDataType	BINARY

INPUTS

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

OUTPUTS

0	name: **MultiThreshold_1_out0**



NODE PROPERTIES

type	StreamingFCLayer_Batch
module	finn.custom_op.fpgadataflow

ATTRIBUTES

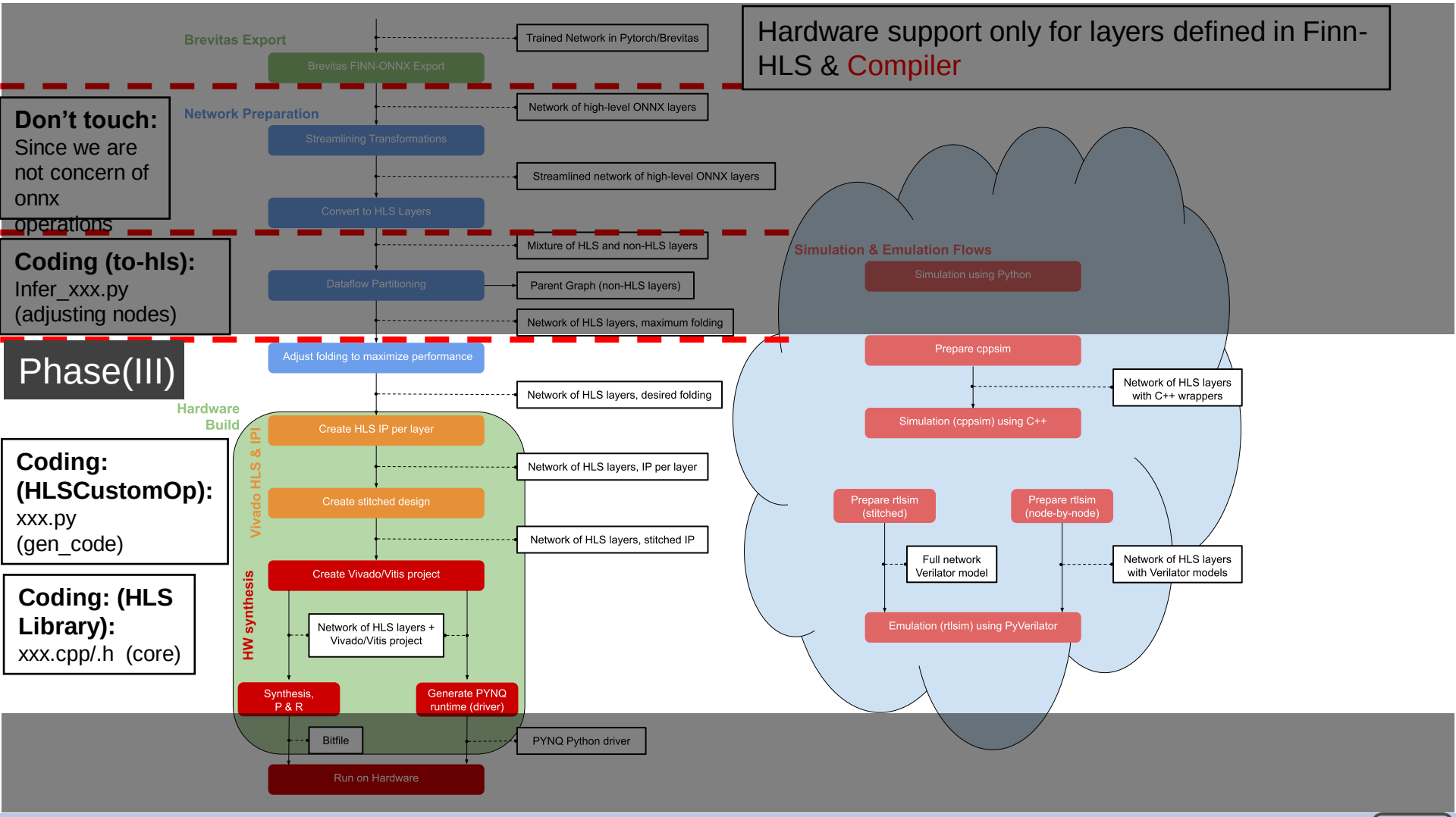
accDataType	UINT10
ActVal	0
backend	fpgadataflow
binaryXnorMode	1
inFIFODepth	16
inputDataType	BINARY
mem_mode	decoupled
MH	64
MW	784
noActivation	0
numInputVectors	1
outFIFODepth	64
outputDataType	BINARY
PE	16
ram_style	block
SIMD	49
weightDataType	BINARY

INPUTS

Overview

- End-to-End Compiling
- Phase (I): Brevitas export
- Phase (II):
  - Network preparation
  - Conversion to HLS layers
- Phase (III): Hardware Build
- Phase (IV): PYNQ deployment

Phase (III): Hardware Build



Phase (III): Hardware Build

- Automatically generate, link all the components and generate hardware bitstream
  - Zynq, Alveo -> ZynqBuild(), VitisBuild()

- ZynqBuild:
  - Preprocessing (Adding IODMA, DataWidthConverter, FIFO)
  1. PrepareIP
  2. HLSSynthIP
  3. CreateStitchedIP
  4. MakeZynqProject

Phase (III): Hardware Build

- Zynq, Alveo -> ZynqBuild(), VitisBuild()

```
# print the names of the supported PYNQ boards
from finn.util.basic import pynq_part_map
print(pynq_part_map.keys())
```

```
dict_keys(['Ultra96', 'Pynq-Z1', 'Pynq-Z2', 'ZCU102', 'ZCU104'])
```

```
# change this if you have a different PYNQ board, see list above
pynq_board = "Pynq-Z2"
fpga_part = pynq_part_map[pynq_board]
target_clk_ns = 10
```

```
from finn.transformation.fpgadataflow.make_zynq_proj import ZynqBuild
model = ModelWrapper(build_dir+"/tfc_w1_a1_set_folding_factors.onnx")
model = model.transform(ZynqBuild(platform = pynq_board, period_ns = target_clk_ns))
```

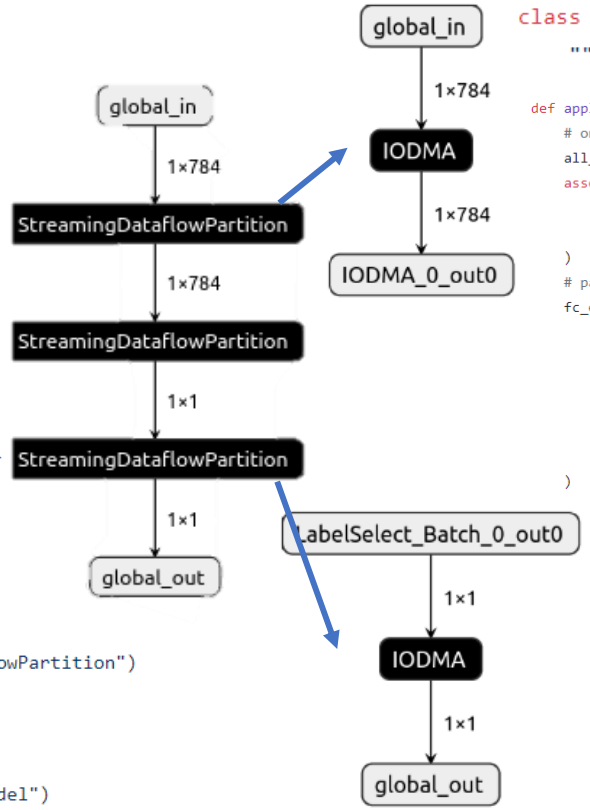
```
/workspace/finn/src/finn/transformation/fpgadataflow/floorplan.py:107: UserWarning: 10 nodes have no entry in the
provided floorplan, SLR was set to -1
  warnings.warn(
/workspace/finn/src/finn/transformation/fpgadataflow/insert_fifo.py:154: UserWarning: Overriding input FIFO depth
to 32
  warnings.warn("Overriding input FIFO depth to 32")
/workspace/finn/src/finn/transformation/fpgadataflow/insert_fifo.py:200: UserWarning: Overriding output FIFO depth
to 32
  warnings.warn("Overriding output FIFO depth to 32")
```

ZynqBuild Preprocessing: IODMA

XILINX FINN

```python
class ZynqBuild(Transformation):
    """Best-effort attempt at building the accelerator for Zynq.
    It assumes the model has only fpgadataflow nodes

    """

    def __init__(self, platform, period_ns, enable_debug=False):
        super().__init__()
        self.fpga_part = pynq_part_map[platform]
        self.period_ns = period_ns
        self.platform = platform
        self.enable_debug = enable_debug

    def apply(self, model):
        # first infer layouts
        model = model.transform(InferDataLayouts())
        # prepare at global level, then break up into kernels
        prep_transforms = [
            InsertIODMA(64),
            InsertDWC(),
            Floorplan(),
            CreateDataflowPartition(),
        ]
        for trn in prep_transforms:
            model = model.transform(trn)
            model = model.transform(GiveUniqueNodeNames())
            model = model.transform(GiveReadableTensorNames())
        # Build each kernel individually
        sdp_nodes = model.get_nodes_by_op_type("StreamingDataflowPartition")
        for sdp_node in sdp_nodes:
            prefix = sdp_node.name + "_"
            sdp_node = getCustomOp(sdp_node)
            dataflow_model_filename = sdp_node.get_nodeattr("model")
            kernel_model = ModelWrapper(dataflow_model_filename)
            kernel_model = kernel_model.transform(InsertFIFO())
            kernel_model = kernel_model.transform(GiveUniqueNodeNames(prefix))
            kernel_model.save(dataflow_model_filename)
```

Here, the first and last partitions contain only an IODMA node, which was inserted automatically to move data between DRAM and the accelerator.



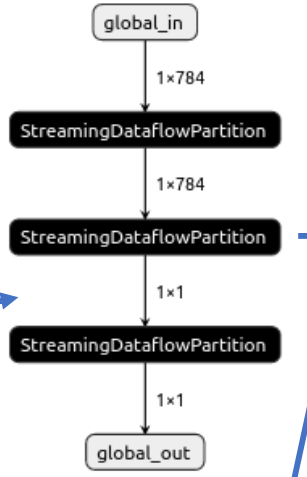

```python
class InsertIODMA(Transformation):
    """Insert DMA nodes on all inputs and outputs."""

    def apply(self, model):
        # only makes sense for a pure fpgadataflow graph -- so we check!
        all_nodes = list(model.graph.node)
        assert all(
            get_by_name(x.attribute, "backend").s.decode("UTF-8") == "fpgadataflow"
            for x in all_nodes
        )
        # parse streamingfclayers looking for external weights with no attached IODMA
        fc_extw_nodes = list(
            filter(
                lambda x: x.op_type == "StreamingFCLayer_Batch"
                and getCustomOp(x).get_nodeattr("mem_mode") == "external"
                and model.find_producer(x.input[1]) is None,
                all_nodes,
            )
        )
```

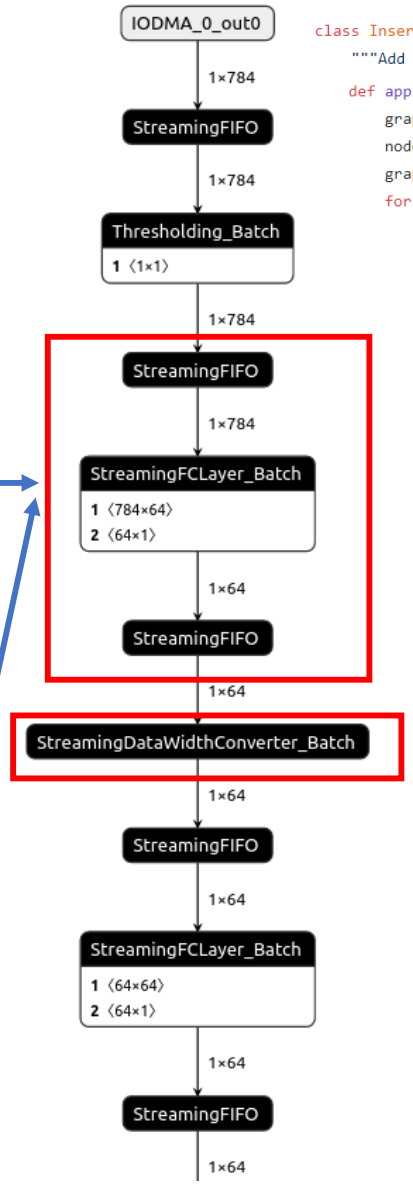
Transform Class is used to modify nodes, set weights

ZynqBuild Preprocessing: DWC

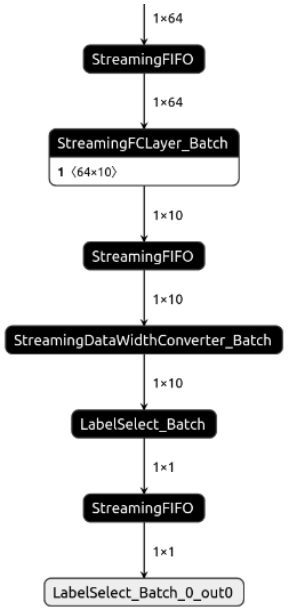
XILINX FINN

```python
class ZynqBuild(Transformation):
    """Best-effort attempt at building the accelerator for Zynq.
    It assumes the model has only fpgadataflow nodes

    """

    def __init__(self, platform, period_ns, enable_debug=F
        super().__init__()
        self.fpga_part = pynq_part_map[platform]
        self.period_ns = period_ns
        self.platform = platform
        self.enable_debug = enable_debug

    def apply(self, model):
        # first infer layouts
        model = model.transform(InferDataLayouts())
        # prepare at global level, then break up into kern
        prep_transforms = [
            InsertIODMA(64),
            InsertDWC(),
            Floorplan(),
            CreateDataflowPartition(),
        ]
        for trn in prep_transforms:
            model = model.transform(trn)
            model = model.transform(GiveUniqueNodeNames())
            model = model.transform(GiveReadableTensorNames())
        # Build each kernel individually
        sdp_nodes = model.get_nodes_by_op_type("StreamingDataflowPartition")
        for sdp_node in sdp_nodes:
            prefix = sdp_node.name + "_"
            sdp_node = getCustomOp(sdp_node)
            dataflow_model_filename = sdp_node.get_nodeattr("model")
            kernel_model = ModelWrapper(dataflow_model_filename)
            kernel_model = kernel_model.transform(InsertFIFO())
            kernel_model = kernel_model.transform(GiveUniqueNodeNames(prefix))
            kernel_model.save(dataflow_model_filename)
```






```python
class InsertDWC(Transformation):
    """Add data width converters between layers where necessary."""
    def apply(self, model):
        graph = model.graph
        node_ind = -1
        graph_modified = False
        for n in graph.node:
            node_ind += 1
            if _suitable_node(n):
                for output_name in n.output:

            dwc_node = oh.make_node(
                "StreamingDataWidthConverter_Batch",
                [output_name],
                [dwc_output_tensor.name],
                domain="finn.custom_op.fpgadataflow",
                backend="fpgadataflow",
                shape=dwc_shape,
                inWidth=dwc_in_width,
```



Continue ZynqBuild( )

```
sdp_nodes = model.get_nodes_by_op_type("StreamingDataflowPartition")
for sdp_node in sdp_nodes:
    prefix = sdp_node.name + "_"
    sdp_node = getCustomOp(sdp_node)
    dataflow_model_filename = sdp_node.get_nodeattr("model")
    kernel_model = ModelWrapper(dataflow_model_filename)
    kernel_model = kernel_model.transform(InsertFIFO())
    kernel_model = kernel_model.transform(GiveUniqueNodeNames(prefix))
    kernel_model.save(dataflow_model_filename)
    kernel_model = kernel_model.transform(
      PrepareIP(self.fpga_part, self.period_ns)
    )
    kernel_model = kernel_model.transform(HLSSynthIP())
    kernel_model = kernel_model.transform(
       CreateStitchedIP(
            self.fpga_part, self.period_ns, sdp_node.onnx_node.name, True
        )
    )
    kernel_model.set_metadata_prop("platform", "zynq-iodma")
    kernel_model.save(dataflow_model_filename)
# Assemble design from IPs
model = model.transform(
   MakeZYNQProject(self.platform, enable_debug=self.enable_debug)
)
```

- Flow
  - Preprocessing (Adding IODMA, DataWidthConverter, FIFO)

1. PrepareIP
2. HLSSynthIP
3. CreateStitchedIP
4. MakeZynqProject

ZynqBuild: 1.PrepareIP

```
class PrepareIP(Transformation):
    """Call custom implementation to generate code for single custom node
    and create folder that contains all the generated files.
    All nodes in the graph must have the fpgadataflow backend attribute and
    transformation gets additional arguments:

    * fpgapart (string)

    * clk in ns (int)

    Any nodes that already have a code_gen_dir_ipgen attribute pointing to a
    will be skipped.

    Outcome if succesful: Node attribute "code_gen_dir_ipgen" contains path t
    that contains generated C++ code that can be used to generate a Vivado IP
    The subsequent transformation is HLSSynthIP"""

    def __init__(self, fpgapart, clk):
        super().__init__()
        self.fpgapart = fpgapart
        self.clk = clk

    def apply(self, model):
        for node in model.graph.node:
            if is_fpgadataflow_node(node) is True:
                _codegen_single_node(node, model, self.fpgapart, self.clk)
        return (model, False)
```

```
def _codegen_single_node(node, model, fpgapart, clk):
    """Calls C++ code generation for one node. Resulting code can be used
    to generate a Vivado IP block for the node."""

    op_type = node.op_type
    try:
        # lookup op_type in registry of CustomOps
        inst = registry.getCustomOp(node)
        # get the path of the code generation directory
        code_gen_dir = inst.get_nodeattr("code_gen_dir_ipgen")
        # ensure that there is a directory
        if code_gen_dir == "" or not os.path.isdir(code_gen_dir):
            code_gen_dir = make_build_dir(
                prefix="code_gen_ipgen_" + str(node.name) + "_"
            )
            inst.set_nodeattr("code_gen_dir_ipgen", code_gen_dir)
            # ensure that there is generated code inside the dir
            inst.code_generation_ipgen(model, fpgapart, clk)
        else:
            warnings.warn("Using pre-existing code for %s" % node.name)
    except KeyError:
        # exception if op_type is not supported
        raise Exception("Custom op_type %s is currently not supported." % op_type)
```

This computation node is of class **HLSCustomOp**

How is the HLS code generated?

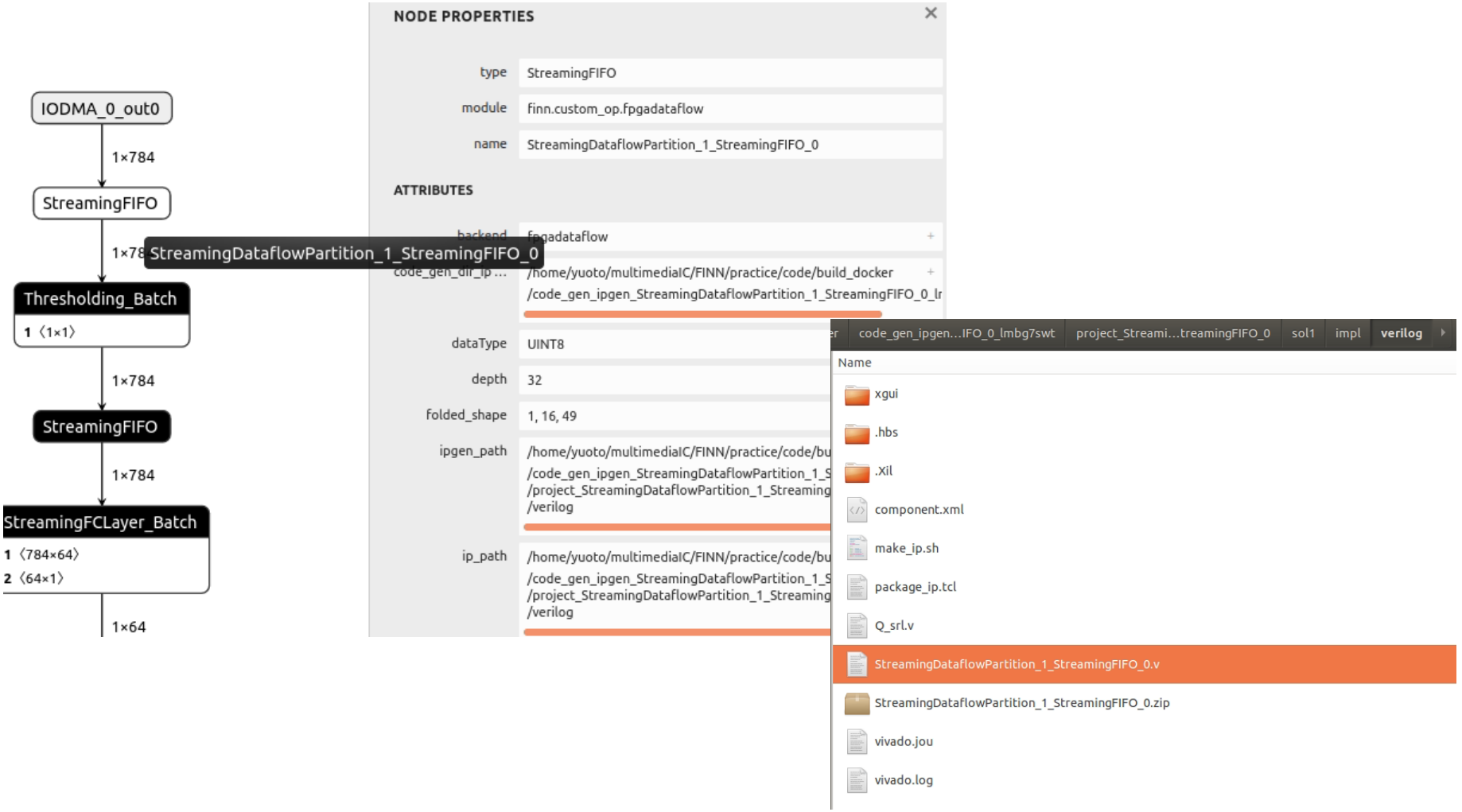
-> We will explain later

2. HLSsynthIP

```
class HLSSynthIP(NodeLocalTransformation):
    """For each node: generate IP block from code in fold
    that is referenced in node attribute "code_gen_dir_ip
    and save path of generated project in node attribute
    All nodes in the graph must have the fpgadataflow bac
    Any nodes that already have a ipgen_path attribute pc
    will be skipped.

    This transformation calls Vivado HLS for synthesis, s
    some time (minutes to hours depending on configuratic
```

- Flow
  - Preprocessing (Adding IODMA, DataWidthConverter, FIFO)
  1. PrepareIP
  2. HLSSynthIP
  3. CreateStitchedIP
  4. MakeZynqProject

```
def applyNodeLocal(self, node):
    op_type = node.op_type
    if is_fpgadataflow_node(node) is True:
        try:
            # lookup op_type in registry of CustomOps
            inst = registry.getCustomOp(node)
            # ensure that code is generated
            assert (
                inst.get_nodeattr("code_gen_dir_ipgen") != ""
            ), """Node
            attribute "code_gen_dir_ipgen" is empty. Please run
            transformation PrepareIP first."""
            if not os.path.isdir(inst.get_nodeattr("ipgen_path")):
                # call the compilation function for this node
                inst.ipgen_singlenode_code()
            else:
                warnings.warn("Using pre-existing IP for %s" % node.name)
            # ensure that executable path is now set
            assert (
                inst.get_nodeattr("ipgen_path") != ""
            ), """Transformation
            HLSSynthIP was not successful. Node attribute "ipgen_path"
            is empty."""
        except KeyError:
            # exception if op_type is not supported
            raise Exception(
                "Custom op_type %s is currently not supported." % op_type
            )
    return (node, False)
```


IODMA_0_out0
1×784
StreamingFIFO
1×78
StreamingDataflowPartition_1_StreamingFIFO_0
Thresholding_Batch
1 ⟨1×1⟩
1×784
StreamingFIFO
1×784
StreamingFCLayer_Batch
1 ⟨784×64⟩
2 ⟨64×1⟩
1×64
NODE PROPERTIES
type
StreamingFIFO
module
finn.custom_op.fpgadataflow
name
StreamingDataflowPartition_1_StreamingFIFO_0
ATTRIBUTES
backend
fpgadataflow
code_gen_dir_ip ...
/home/yuoto/multimediaIC/FINN/practice/code/build_docker
/code_gen_ipgen_StreamingDataflowPartition_1_StreamingFIFO_0_lr
dataType
UINT8
depth
32
folded_shape
1, 16, 49
ipgen_path
/home/yuoto/multimediaIC/FINN/practice/code/bu
/code_gen_ipgen_StreamingDataflowPartition_1_S
/project_StreamingDataflowPartition_1_Streaming
/verilog
ip_path
/home/yuoto/multimediaIC/FINN/practice/code/bu
/code_gen_ipgen_StreamingDataflowPartition_1_S
/project_StreamingDataflowPartition_1_Streaming
/verilog
code_gen_ipgen...IFO_0_lmbg7swt
project_Streami...treamingFIFO_0
sol1
impl
verilog
Name
xgui
.hbs
.Xil
component.xml
make_ip.sh
package_ip.tcl
Q_srl.v
StreamingDataflowPartition_1_StreamingFIFO_0.v
StreamingDataflowPartition_1_StreamingFIFO_0.zip
vivado.jou
vivado.log

3. CreateStitchedIP

```
class CreateStitchedIP(Transformation):
    """Create a Vivado IP Block Design project from all the generated IPs of a
    graph. All nodes in the graph must have the fpgadataflow backend attribute,
def apply(self, model):
    # ensure non-relative readmemh .dat files
    model = model.transform(ReplaceVerilogRelPaths())
    ip_dirs = ["list"]
    # add RTL streamer IP
    ip_dirs.append("/workspace/finn/finn-rtllib/memstream")
    # ensure that all nodes are fpgadataflow, and that IPs are generated
  # create a temporary folder for the project
  prjname = "finn_vivado_stitch_proj"
  vivado_stitch_proj_dir = make_build_dir(prefix="vivado_stitch_proj_")
  model.set_metadata_prop("vivado_stitch_proj", vivado_stitch_proj_dir)
  # start building the tcl script
  tcl = []
  # create vivado project
  tcl.append(
      "create_project %s %s -part %s"
      % (prjname, vivado_stitch_proj_dir, self.fpgapart)
  )
  # add all the generated IP dirs to ip_repo_paths
  ip_dirs_str = " ".join(ip_dirs)
  tcl.append("set_property ip_repo_paths [%s] [current_project]" % ip_dirs_str)
  tcl.append("update_ip_catalog")
  # create block design and instantiate all layers
  block_name = self.ip_name
  tcl.append('create_bd_design "%s"' % block_name)
  tcl.extend(self.create_cmds)
  tcl.extend(self.connect_cmds)
  fclk_mhz = 1 / (self.clk_ns * 0.001)
  fclk_hz = fclk_mhz * 1000000
```

- Flow
  - Preprocessing (Adding IODMA, DataWidthConverter, FIFO)
  1. PrepareIP
  2. HLSSynthIP
  3. CreateStitchedIP
  4. MakeZynqProject

```
  # create a shell script and call Vivado
  make_project_sh = vivado_stitch_proj_dir + "/make_project.sh"
  working_dir = os.environ["PWD"]
  with open(make_project_sh, "w") as f:
      f.write("#!/bin/bash \n")
      f.write("cd {}\n".format(vivado_stitch_proj_dir))
      f.write("vivado -mode batch -source make_project.tcl\n")
      f.write("cd {}\n".format(working_dir))
  bash_command = ["bash", make_project_sh]
  process_compile = subprocess.Popen(bash_command, stdout=subprocess.PIPE)
  process_compile.communicate()
  return (model, False)
```

4. MakeZYNQProject(1/2)

```
class MakeZYNQProject(Transformation):
    """Create a Vivado overlay project (including the shell infrastructure)
    from the already-stitched IP block for this graph.

    deploy_hwh_name = vivado_pynq_proj_dir + "/resizer.hwh"
    copy(hwh_name, deploy_hwh_name)
    model.set_metadata_prop("hw_handoff", deploy_hwh_name)
    # filename for the synth utilization report
    synth_report_filename = vivado_pynq_proj_dir + "/synth_report.xml"
    model.set_metadata_prop("vivado_synth_rpt", synth_report_filename)
    return (model, False)
```

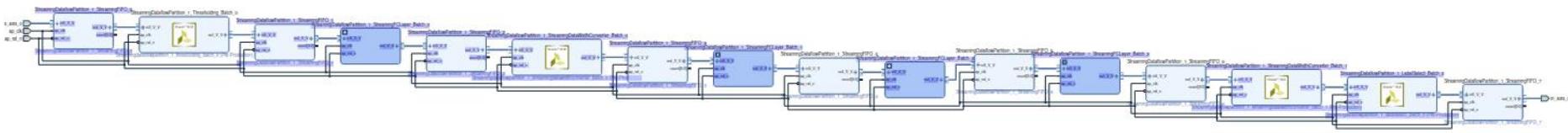
- Flow
  - Preprocessing (Adding IODMA, DataWidthConverter, FIFO)
  1. PrepareIP
  2. HLSSynthIP
  3. CreateStitchedIP
  4. MakeZynqProject

4. MakeZYNQProject(2/2)

- Vivado project was automatically built and stitched

```
model = ModelWrapper(postsynth_layers)
model.model.metadata_props
```

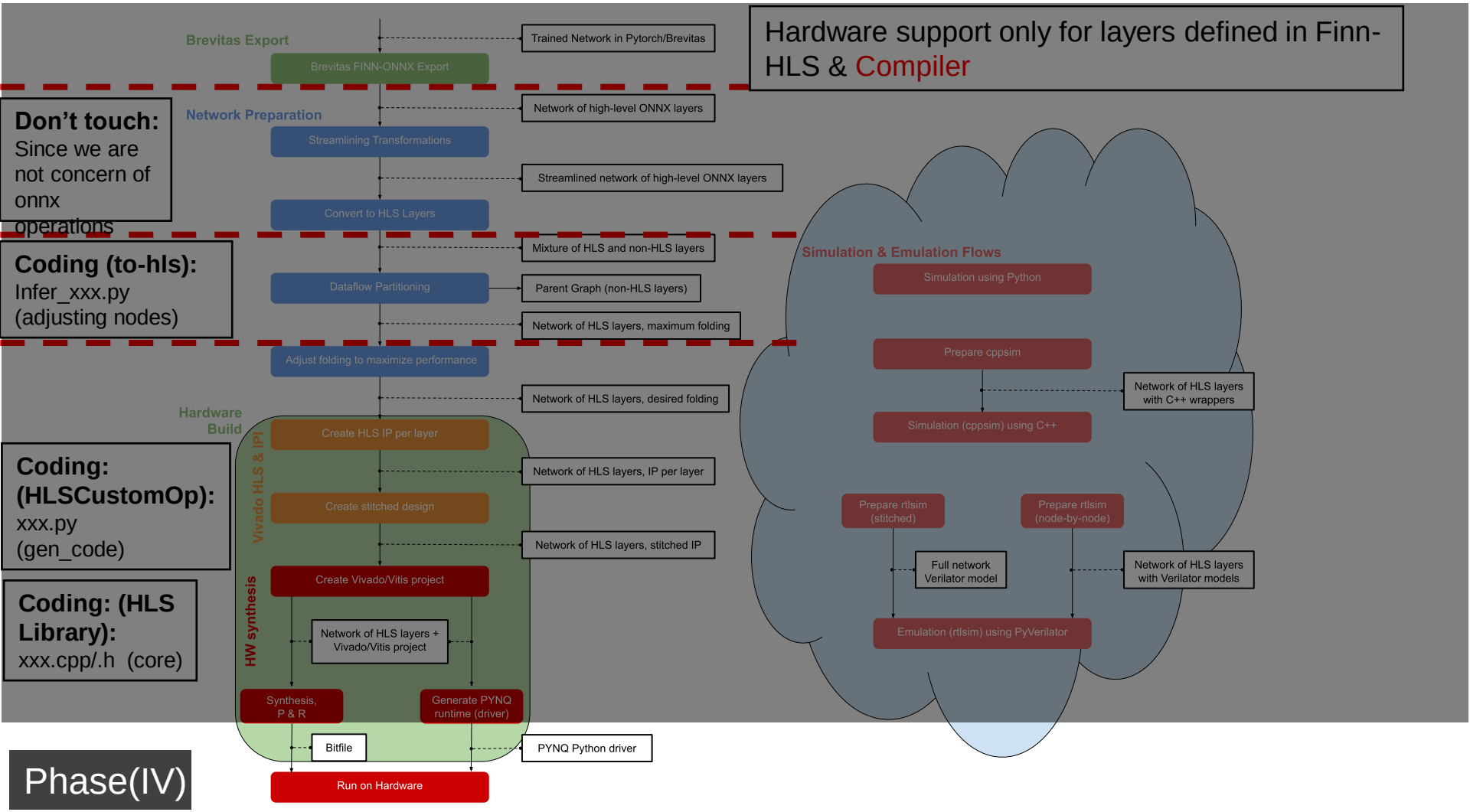
```
[key: "floorplan_json"
value: "/home/yuoto/multimediaIC/FINN/practice/code/build_docker/vitis_floorplan_zxz9em07/floorplan.json"
, key: "vivado_stitch_proj"
value: "/home/yuoto/multimediaIC/FINN/practice/code/build_docker/vivado_stitch_proj_ol2h0n49"
, key: "clk_ns"
value: "10"
, key: "wrapper_filename"
value: "/home/yuoto/multimediaIC/FINN/practice/code/build_docker/vivado_stitch_proj_ol2h0n49/finn_vivado_stitch_pr
oj.srcs/sources_1/bd/StreamingDataflowPartition_1/hdl/StreamingDataflowPartition_1_wrapper.v"
, key: "vivado_stitch_vlnv"
value: "xilinx_finn:finn:StreamingDataflowPartition_1:1.0"
, key: "vivado_stitch_ifnames"
value: "{\"clk\": [\"ap_clk\"], \"rst\": [\"ap_rst_n\"], \"s_axis\": [[\"s_axis_0\", 392]], \"m_axis\": [[\"m_axis
_0\", 8]], \"aximm\": [], \"axilite\": []}"
, key: "platform"
value: "zynq-iodma"
]
```



Overview

- End-to-End Compiling
- Phase (I): Brevitas export
- Phase (II):
  - Network preparation
  - Conversion to HLS layers
- Phase (III): Hardware Build
- Phase (IV): PYNQ deployment

Phase (IV): PYNQ deployment



Phase(IV)

Phase (IV): PYNQ deployment

- Deployment and Remote Execution
- Validating the Accuracy on a PYNQ Board
- Throughput Test on PYNQ Board

Appendix:

How is HLS code generated?
How to (possibly) add custom operation?

```
#define AP_INT_MAX_W 784

#include "bnn-library.h"

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

// defines for network parameters
#define MW1 784
 #define MH1 64

          #define SIMD1 49
 #define PE1 16
 #define WMEM1 64

          #define TMEM1 4
 #define numReps 1
#define WP1 1
```

```
set config_proj_name project_StreamingDataflowPartition_1_StreamingFCLayer_Batch_0
puts "HLS project: $config_proj_name"
set config_hwsrcdir "/home/yuoto/multimediaIC/FINN/practice/code/build_docker/code_gen_ipgen_StreamingDa
puts "HW source dir: $config_hwsrcdir"
set config_proj_part "xc7z020clg400-1"

set config_bnnlibdir "/workspace/finn-hlslib"

set config_toplevelfxn "StreamingDataflowPartition_1_StreamingFCLayer_Batch_0"
set config_clkperiod 10

open_project $config_proj_name
add_files $config_hwsrcdir/top_StreamingDataflowPartition_1_StreamingFCLayer_Batch_0.cpp -cflags "-std=

set_top $config_toplevelfxn
open_solution sol1
set_part $config_proj_part

config_compile -ignore_long_run_time -disable_unroll_code_size_check
config_interface -m_axi_addr64
config_rtl -auto_prefix


create_clock -period $config_clkperiod -name default
csynth_design
```

```
void StreamingDataflowPartition_1_StreamingFCLayer_Batch_0(
                    hls::stream<ap_uint<49>> &in0,
                    hls::stream<ap_uint<784>> &weights,
                    hls::stream<ap_uint<16>> &out
                    )
{
#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
Matrix_Vector_Activate_Stream_Batch<MW1, MH1, SIMD1, PE1, Recast<XnorMul>, Slice<ap_uint<1>>, Identity,
                (in0, out, weights, threshs, numReps, ap_resource_lut());
```

This hls top function code & its tcl file is generated automatically by previous process

HLSCustomOp Class

```
class HLSCustomOp(CustomOp):
    """HLSCustomOp class all custom ops that correspond to a finn-hlslib
    function are based on. Contains different functions every fpgadataflow
    custom node should have. Some as abstract methods, these have to be filled
    when writing a new fpgadataflow custom op node."""

    def __init__(self, onnx_node):
        super().__init__(onnx_node)

        self.code_gen_dict = {}

        # getting templates from templates.py

        # template for single node execution
        self.docompute_template = templates.docompute_template

        # templates for single node ip generation
        # cpp file
        self.ipgen_template = templates.ipgen_template
        # tcl script
        self.ipgentcl_template = templates.ipgentcl_template
```

Abstract methods

```
@abstractmethod
def get_number_output_values(self):
    """Function to get the number of expected output values,
    is member function of HLSCustomOp class but has to be filled
    by every node."""
    pass

@abstractmethod
def global_includes(self):
    """Function to set the global includes for c++ code that has to be generated
    for cppsim or rtlsim, is member function of HLSCustomOp class but has to
    be filled by every node."""
    pass

@abstractmethod
def defines(self, var):
    """Function to set the define commands for c++ code that has to be generated
    for cppsim or rtlsim, is member function of HLSCustomOp class but has to
    be filled by every node.

    var: makes it possible to reuse the function for different c++ code generation.
    I.e. if set to "ipgen" in StreamingFCLayer_Batch additional PRAGMA defines are
    added."""
    pass
```

Abstract methods

- get_number_output_values( ) -> PE=10, numInputVectors=1 ->10*1=10
- global_includes( ) -> self.code_gen_dict[“$GLOBALS$”].append()
- defines( ) -> self.code_gen_dict[“$DEFINES$”].append()
- blackboxfunction( ) -> self.code_gen_dict[“$BLACKBOXFUNCTION$”]
- pragmas( ) -> self.code_gen_dict[“$PRAGMAS$”]
- docompute( ) -> self.code_gen_dict[“$DOCPUMPUTE$”]
- read_npy_data
- strm_decl
- dataoutstrm
- save_as_npy

This part is for the cpp simulation using g++

StreamingFCLayer_Batch.py (~1500 lines)

```python
class StreamingFCLayer_Batch(HLSCustomOp):
    """Class that corresponds to finn-hls StreamingFCLayer_Batch function."""

    def __init__(self, onnx_node):
        super().__init__(onnx_node)
        self.decoupled_wrapper = templates.decoupled_wrapper

    def get_nodeattr_types(self):
        my_attrs = {
            "PE": ("i", True, 0),
            "SIMD": ("i", True, 0),
            "MW": ("i", True, 0),
            "MH": ("i", True, 0),
            "resType": ("s", False, "lut", {"auto", "lut", "dsp"}),
            "ActVal": ("i", False, 0),
            # FINN DataTypes for inputs, weights, outputs
            "inputDataType": ("s", True, ""),
            "weightDataType": ("s", True, ""),
            "outputDataType": ("s", True, ""),
            # FINN DataType for accumulator -- auto-computed and updated
            "accDataType": ("s", False, "INT32"),
            # use xnor-popcount for binary weights/inputs, thus treating them
            # as bipolar
            "binaryXnorMode": ("i", False, 0, {0, 1}),
```

```
def docompute(self):
    mem_mode = self.get_nodeattr("mem_mode")
    map_to_hls_mult_style = {
        "auto": "ap_resource_dflt()",
        "lut": "ap_resource_lut()",
        "dsp": "ap_resource_dsp()",
    }
    tmpl_args = self.get_template_param_values()
    if self.calc_tmem() == 0:
        odtype_hls_str = self.get_output_datatype().get_hls_datatype_str()
        threshs = "PassThroughActivation<%s>()" % odtype_hls_str
    else:
        threshs = "threshs"
    if mem_mode == "const":
        node = self.onnx_node
        self.code_gen_dict["$DOCOMPUTE$"] = [
            """{}<MW1, MH1, SIMD1, PE1, {}, {}, {}>
            (in0, out, weights, {}, numReps, {});""".format(
                node.op_type,
                tmpl_args["TSrcI"],
                tmpl_args["TDstI"],
                tmpl_args["TWeightI"],
                threshs,
                map_to_hls_mult_style[self.get_nodeattr("resType")],
```

```
#define AP_INT_MAX_W 784

#include "bnn-library.h"

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

// defines for network parameters
#define MW1 784
 #define MH1 64

          #define SIMD1 49
 #define PE1 16
 #define WMEM1 64

          #define TMEM1 4
 #define numReps 1
#define WP1 1
void StreamingDataflowPartition_1_StreamingFCLayer_Batch_0(
                    hls::stream<ap_uint<49>> &in0,
                    hls::stream<ap_uint<784>> &weights,
                    hls::stream<ap_uint<16>> &out
                    )
{
#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
Matrix_Vector_Activate_Stream_Batch<MW1, MH1, SIMD1, PE1, Recast<XnorMul>, Slice<ap_uint<1>>, Identity,
                (in0, out, weights, threshs, numReps, ap_resource_lut());
```

```
    def global_includes(self):
        self.code_gen_dict["$GLOBALS$"] = ['#include "weights.hpp"']
        self.code_gen_dict["$GLOBALS$"] += ['#include "activations.hpp"']

        mem_mode = self.get_nodeattr("mem_mode")
        if mem_mode == "const":
            # self.code_gen_dict["$GLOBALS$"] += ['#include "params.h"']
            pass
        elif mem_mode == "decoupled" or mem_mode == "external":
            self.code_gen_dict["$GLOBALS$"] += ['#include "mvau.hpp"']
        else:
            raise Exception(
                """Please set mem_mode to "const", "decoupled", or "external",
                currently no other parameter value is supported!"""
            )
        if self.calc_tmem() != 0:
            # TODO find a better way of checking for no pregenerated thresholds
            self.code_gen_dict["$GLOBALS$"] += ['#include "thresh.h"']
```

```
#define AP_INT_MAX_W 784

#include "bnn-library.h"

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

// defines for network parameters
#define MW1 784
 #define MH1 64

          #define SIMD1 49
 #define PE1 16
 #define WMEM1 64

          #define TMEM1 4
 #define numReps 1
#define WP1 1
void StreamingDataflowPartition_1_StreamingFCLayer_Batch_0(
                    hls::stream<ap_uint<49>> &in0,
                    hls::stream<ap_uint<784>> &weights,
                    hls::stream<ap_uint<16>> &out
                    )
{
#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
Matrix_Vector_Activate_Stream_Batch<MW1, MH1, SIMD1, PE1, Recast<XnorMul>, Slice<ap_uint<1>>, Identity,
          (in0, out, weights, threshs, numReps, ap_resource_lut());
```

```
def defines(self, var):
    mem_mode = self.get_nodeattr("mem_mode")
    numInputVectors = list(self.get_nodeattr("numInputVectors"))
    numReps = np.prod(numInputVectors)
    self.code_gen_dict["$DEFINES$"] = [
        """#define MW1 {}\n #define MH1 {}\n
        #define SIMD1 {}\n #define PE1 {}\n #define WMEM1 {}\n
        #define TMEM1 {}\n #define numReps {}""".format(
            self.get_nodeattr("MW"),
            self.get_nodeattr("MH"),
            self.get_nodeattr("SIMD"),
            self.get_nodeattr("PE"),
            self.calc_wmem(),
            self.calc_tmem(),
            numReps,
        )
    ]
    if mem_mode == "decoupled" or mem_mode == "external":
        wdt = self.get_weight_datatype()
        self.code_gen_dict["$DEFINES$"].append(
            "#define WP1 {}\n".format(wdt.bitwidth())
        )
```

```
#define AP_INT_MAX_W 784

#include "bnn-library.h"

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

// defines for network parameters
#define MW1 784
 #define MH1 64

          #define SIMD1 49
 #define PE1 16
 #define WMEM1 64

          #define TMEM1 4
 #define numReps 1
#define WP1 1
void StreamingDataflowPartition_1_StreamingFCLayer_Batch_
                    hls::stream<ap_uint<49>> &in0,
                    hls::stream<ap_uint<784>> &weights,
                    hls::stream<ap_uint<16>> &out
                    )
{
#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 (
#pragma HLS ARRAY_PARTITION variable=threshs.m_thresholds complete dim=3
Matrix_Vector_Activate_Stream_Batch<MW1, MH1, SIMD1, PE1, Recast<XnorMul>, Slice<ap_uint<1>>, Identity,
            (in0, out, weights, threshs, numReps, ap_resource_lut());
```

```
def docompute(self):
    mem_mode = self.get_nodeattr("mem_mode")
    map_to_hls_mult_style = {
        "auto": "ap_resource_dflt()",
        "lut": "ap_resource_lut()",
        "dsp": "ap_resource_dsp()",
    }
    tmpl_args = self.get_template_param_values()
    if self.calc_tmem() == 0:
        odtype_hls_str = self.get_output_datatype().get_hls_datatype_str()
        threshs = "PassThroughActivation<%s>()" % odtype_hls_str
    else:
        threshs = "threshs"
    if mem_mode == "const":
        node = self.onnx_node
        self.code_gen_dict["$DOCOMPUTE$"] = [
            """{}<MW1, MH1, SIMD1, PE1, {}, {}, {}>
            (in0, out, weights, {}, numReps, {});""".format(
                node.op_type,
                tmpl_args["TSrcI"],
                tmpl_args["TDstI"],
                tmpl_args["TWeightI"],
```

```
elif mem_mode == "decoupled" or mem_mode == "external":
    wdt = self.get_weight_datatype()
    if wdt == DataType.BIPOLAR:
        export_wdt = DataType.BINARY
    else:
        export_wdt = wdt
    wdtype_hls_str = export_wdt.get_hls_datatype_str()
    self.code_gen_dict["$DOCOMPUTE$"] = [
        """Matrix_Vector_Activate_Stream_Batch<MW1, MH1, SIMD1, PE1, {}, {}, {}, {} >
        (in0, out, weights, {}, numReps, {});""".format(
            tmpl_args["TSrcI"],
            tmpl_args["TDstI"],
            tmpl_args["TWeightI"],
            wdtype_hls_str,
            threshs,
            map_to_hls_mult_style[self.get_nodeattr("resType")],
        )
    ]
```

```
#define AP_INT_MAX_W 784

#include "bnn-library.h"

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

// defines for network parameters
#define MW1 784
 #define MH1 64

          #define SIMD1 49
 #define PE1 16
 #define WMEM1 64

          #define TMEM1 4
 #define numReps 1
#define WP1 1
void StreamingDataflowPartition_1_StreamingFCLayer_Batch
                    hls::stream<ap_uint<49>> &in0,
                    hls::stream<ap_uint<784>> &weights,
                    hls::stream<ap_uint<16>> &out
                    )
{
#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
Matrix_Vector_Activate_Stream_Batch<MW1, MH1, SIMD1, PE1, Recast<XnorMul>, Slice<ap_uint<1>>, Identity,
          (in0, out, weights, threshs, numReps, ap_resource_lut());
```

```
def pragmas(self):
    mem_mode = self.get_nodeattr("mem_mode")
    self.code_gen_dict["$PRAGMAS$"] = ["#pragma HLS INTERFACE axis port=in0"]
    self.code_gen_dict["$PRAGMAS$"].append("#pragma HLS INTERFACE axis port=out")
    in_fifo_depth = self.get_nodeattr("inFIFODepth")
    out_fifo_depth = self.get_nodeattr("outFIFODepth")
    # insert depth pragmas only if specified
    if in_fifo_depth != 0:
        self.code_gen_dict["$PRAGMAS$"].append(
            "#pragma HLS stream depth=%d variable=in0" % in_fifo_depth
        )
    if out_fifo_depth != 0:
        self.code_gen_dict["$PRAGMAS$"].append(
            "#pragma HLS stream depth=%d variable=out" % out_fifo_depth
        )
    self.code_gen_dict["$PRAGMAS$"].append(
        "#pragma HLS INTERFACE ap_ctrl_none port=return"
    )

    if mem_mode == "const":
        self.code_gen_dict["$PRAGMAS$"].append('#include "params.h"')
        # the weight tensor is ap_uint<simd*prec> [PE][WMEM]
        # partition for parallel access along the PE dimension (dim 1)
        self.code_gen_dict["$PRAGMAS$"].append(
            (
```

```
#define AP_INT_MAX_W 784

#include "bnn-library.h"

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

// defines for network parameters
#define MW1 784
 #define MH1 64

          #define SIMD1 49
 #define PE1 16
 #define WMEM1 64

          #define TMEM1 4
 #define numReps 1
#define WP1 1
void StreamingDataflowPartition_1_StreamingFCLayer_Batch_0(
                    hls::stream<ap_uint<49>> &in0,
                    hls::stream<ap_uint<784>> &weights,
                    hls::stream<ap_uint<16>> &out
                    )
{
#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
Matrix_Vector_Activate_Stream_Batch<MW1, MH1, SIMD1, PE1, Recast<XnorMul>, Slice<ap_uint<1>>, Identity,
                (in0, out, weights, threshs, numReps, ap_resource_lut());
```

```
def blackboxfunction(self):
    mem_mode = self.get_nodeattr("mem_mode")
    if mem_mode == "const":
        self.code_gen_dict["$BLACKBOXFUNCTION$"] = [
            """void {}(hls::stream<ap_uint<{}>> &in0,
                hls::stream<ap_uint<{}>> &out
                )""".format(
                self.onnx_node.name,
                self.get_instream_width(),
                self.get_outstream_width(),
            )
        ]
    elif mem_mode == "decoupled" or mem_mode == "external":
        self.code_gen_dict["$BLACKBOXFUNCTION$"] = [
            """void {}(
                hls::stream<ap_uint<{}>> &in0,
                hls::stream<ap_uint<{}>> &weights,
                hls::stream<ap_uint<{}>> &out
                )""".format(
                self.onnx_node.name,
                self.get_instream_width(),
                self.get_weightstream_width(),
                self.get_outstream_width(),
            )
```

```python
def get_input_datatype(self):
    """Returns FINN DataType of input."""
    ret = DataType[self.get_nodeattr("inputDataType")]
    return ret

def get_output_datatype(self):
    """Returns FINN DataType of output."""
    ret = DataType[self.get_nodeattr("outputDataType")]
    return ret

def get_instream_width(self):
    """Returns input stream width."""
    ibits = self.get_input_datatype().bitwidth()
    pe = self.get_nodeattr("PE")
    in_width = pe * ibits
    return in_width
```

Other helper functions are also need to be written

```python
class DataType(Enum):
    """Enum class that contains FINN data types to set the quantization annotation.
    ONNX does not support data types smaller than 8-bit integers, whereas in FINN we are
    interested in smaller integers down to ternary and bipolar.

    Assignment of DataTypes to indices based on following ordering:

    * unsigned to signed

    * fewer to more bits

    Currently supported DataTypes:"""

    # important: the get_smallest_possible() member function is dependent on ordering.
    BINARY = auto()
    UINT2 = auto()
    UINT3 = auto()
    UINT4 = auto()
    UINT5 = auto()
    UINT6 = auto()
```

Later, the HLSCustomOp Class member will…

Generate HLS top function code & .tcl file

```
def code_generation_ipgen(self, model, fpgapart, clk):
    """Generates c++ code and tcl script for ip generation."""
    node = self.onnx_node

    # generate top cpp file for ip generation
    path = self.get_nodeattr("code_gen_dir_ipgen")
    self.code_gen_dict["$AP_INT_MAX_W$"] = [str(self.get_ap_int_max_w())]
    self.generate_params(model, path)
    self.global_includes()
    self.defines("ipgen")
    self.blackboxfunction()
    self.pragmas()
    self.docompute()

    template = self.ipgen_template

    for key in self.code_gen_dict:
        # transform list into long string separated by '\n'
        code_gen_line = "\n".join(self.code_gen_dict[key])
        template = template.replace(key, code_gen_line)
    code_gen_dir = self.get_nodeattr("code_gen_dir_ipgen")
    f = open(os.path.join(code_gen_dir, "top_{}.cpp".format(node.name)), "w")
    f.write(template)
    f.close()
```

- Synthesis the hardware by calling vivado_hls

```
def ipgen_singlenode_code(self):
    """Builds the bash script for ip generation using the CallHLS from
    finn.util.hls."""
    node = self.onnx_node
    code_gen_dir = self.get_nodeattr("code_gen_dir_ipgen")
    builder = CallHLS()
    builder.append_tcl(code_gen_dir + "/hls_syn_{}.tcl".format(node.name))
    builder.set_ipgen_path(code_gen_dir + "/project_{}".format(node.name))
    builder.build(code_gen_dir)
    ipgen_path = builder.ipgen_path
    assert os.path.isdir(ipgen_path), "IPGen failed: %s not found" % (ipgen_path)
    self.set_nodeattr("ipgen_path", ipgen_path)
    ip_path = ipgen_path + "/sol1/impl/ip"
    assert os.path.isdir(
        ip_path
    ), "IPGen failed: %s not found. Check log under %s" % (ip_path, code_gen_dir)
    self.set_nodeattr("ip_path", ip_path)
    vlnv = "xilinx.com:hls:%s:1.0" % node.name
    self.set_nodeattr("ip_vlnv", vlnv)
```