Bridge of Life Education

FINN Compiler

Lecturer: Hua-Yang Weng

Date: 2022/08/18

[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

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.

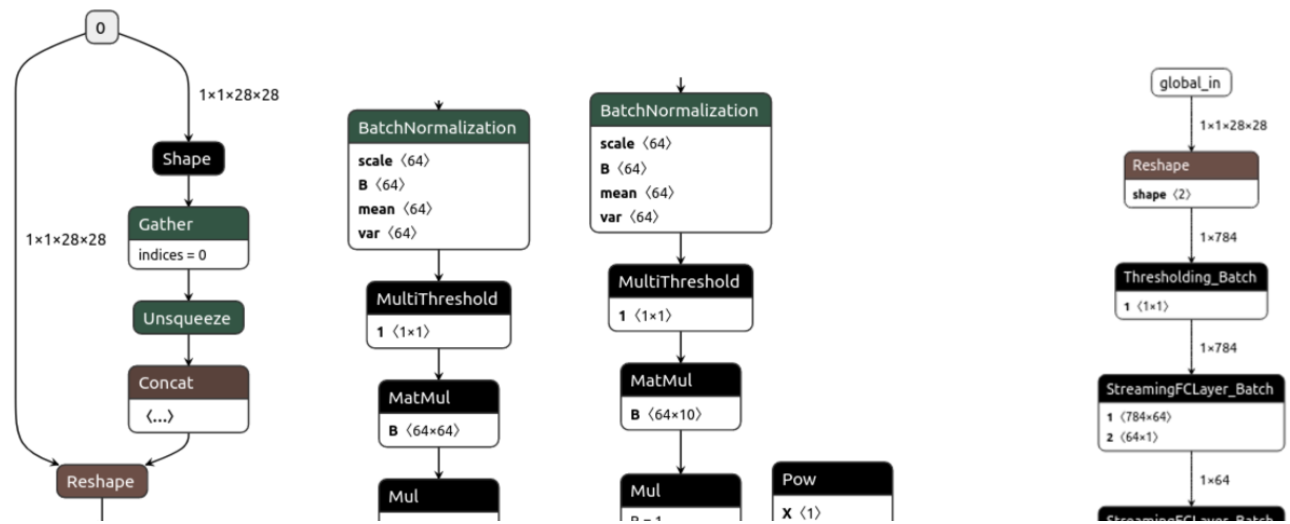


Table of Contents

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

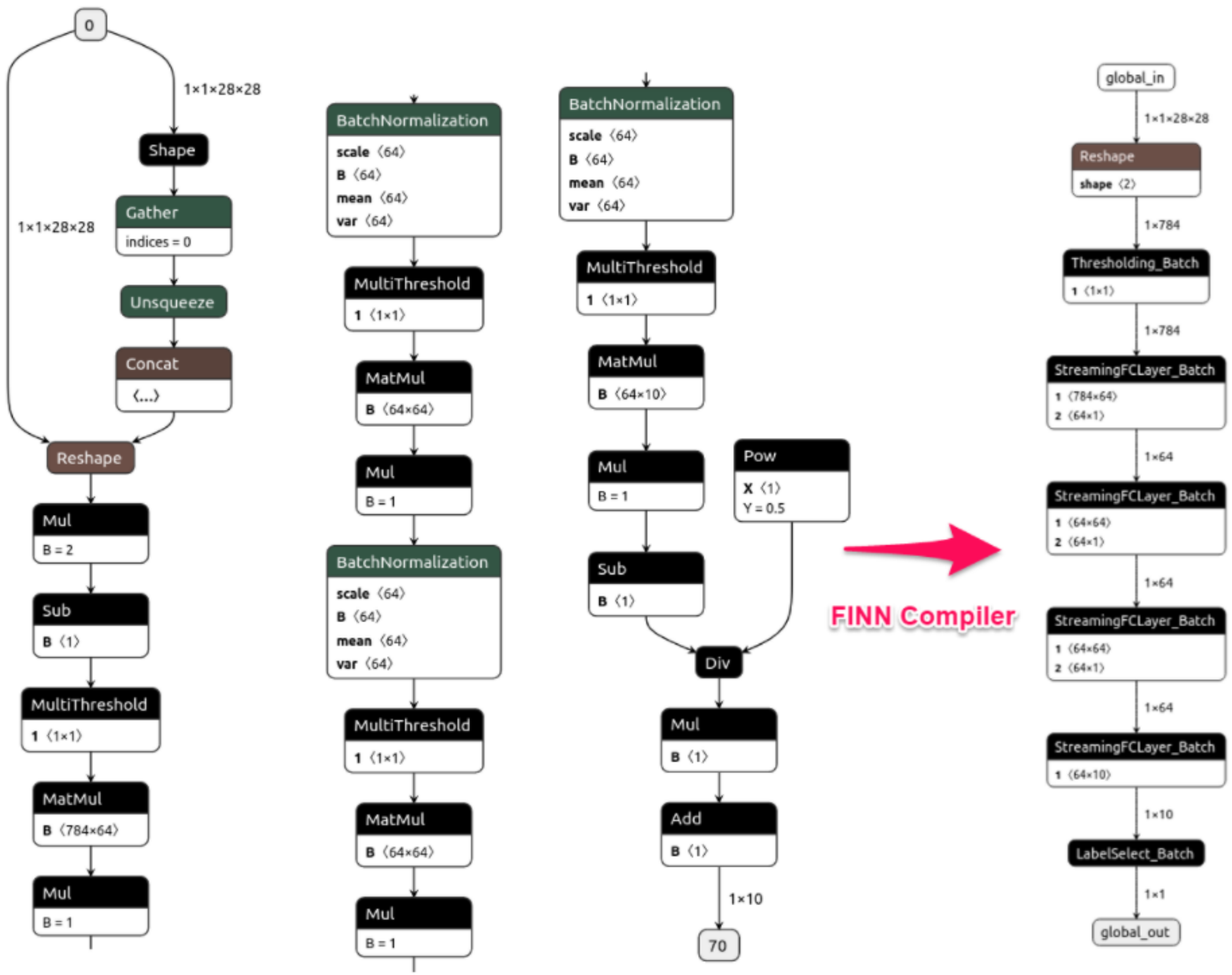
- End-to-End Compiling
- Phase (I): Brevitas export
- Phase (II):
  - Network preparation
  - Conversion to HLS layers
- Phase (III): Hardware Build
- Phase (IV): PYNQ deployment

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
Gather
indices = 0
1×1×28×28
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

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

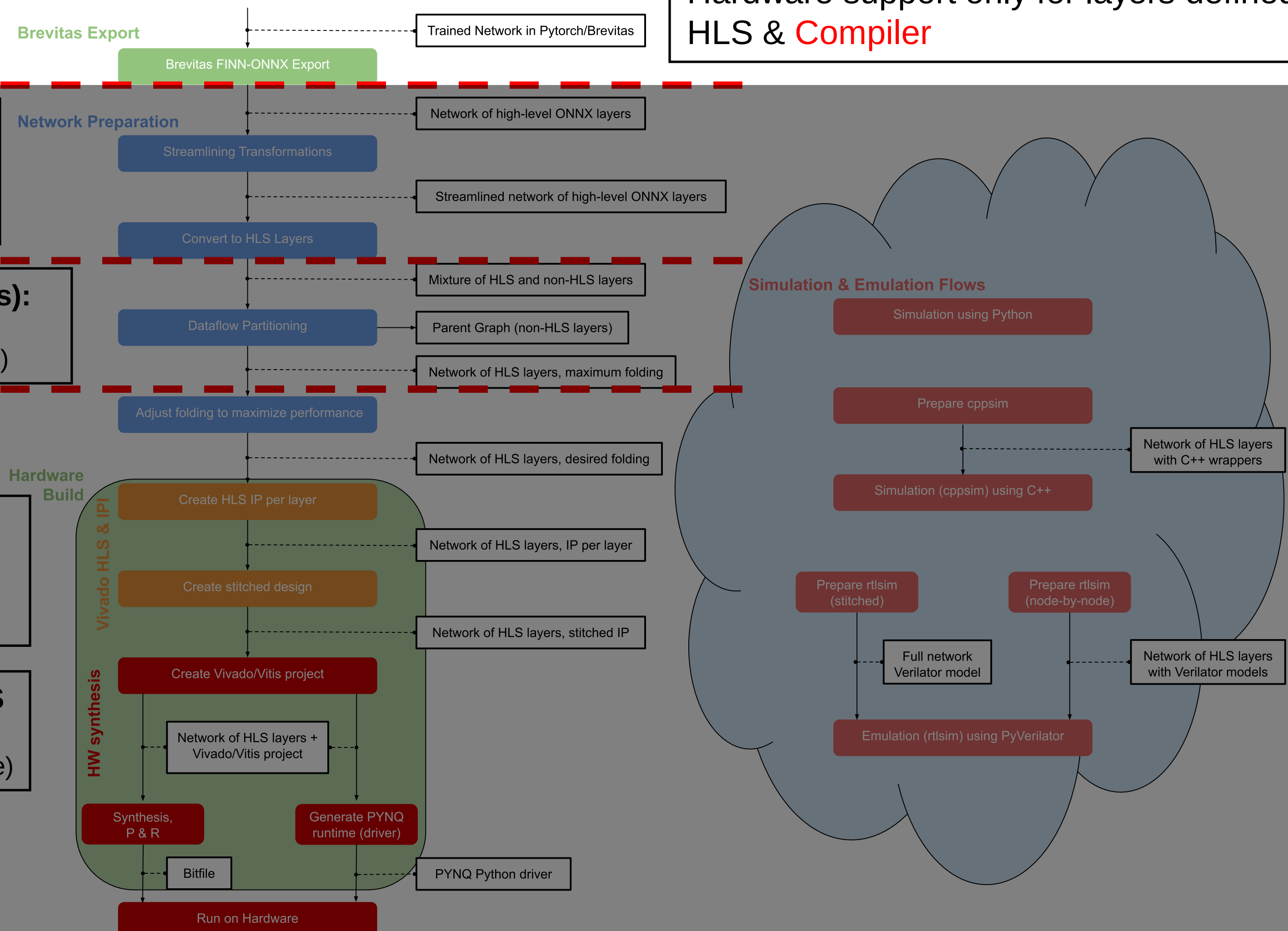
Phase(I)

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

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

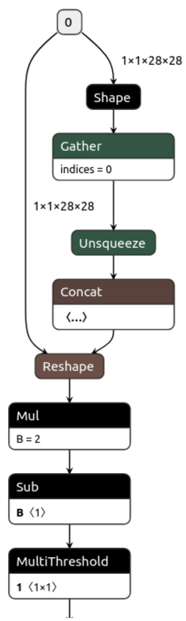
Coding: (HLSCustom Op):
xxx.py
(gen_code)

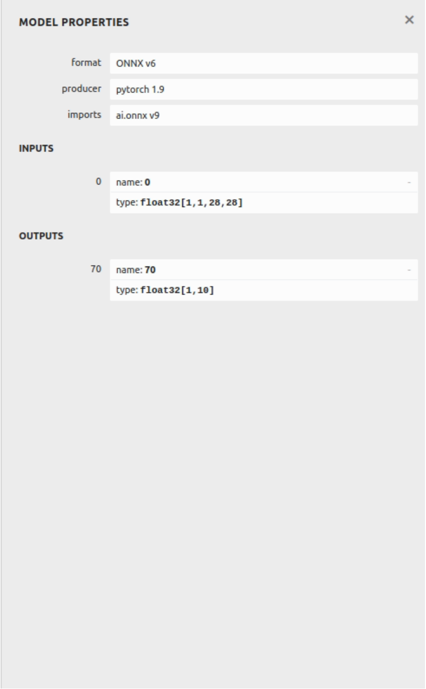
Coding: (HLS Library):
xxx.cpp/.h (core)



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

```python
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

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

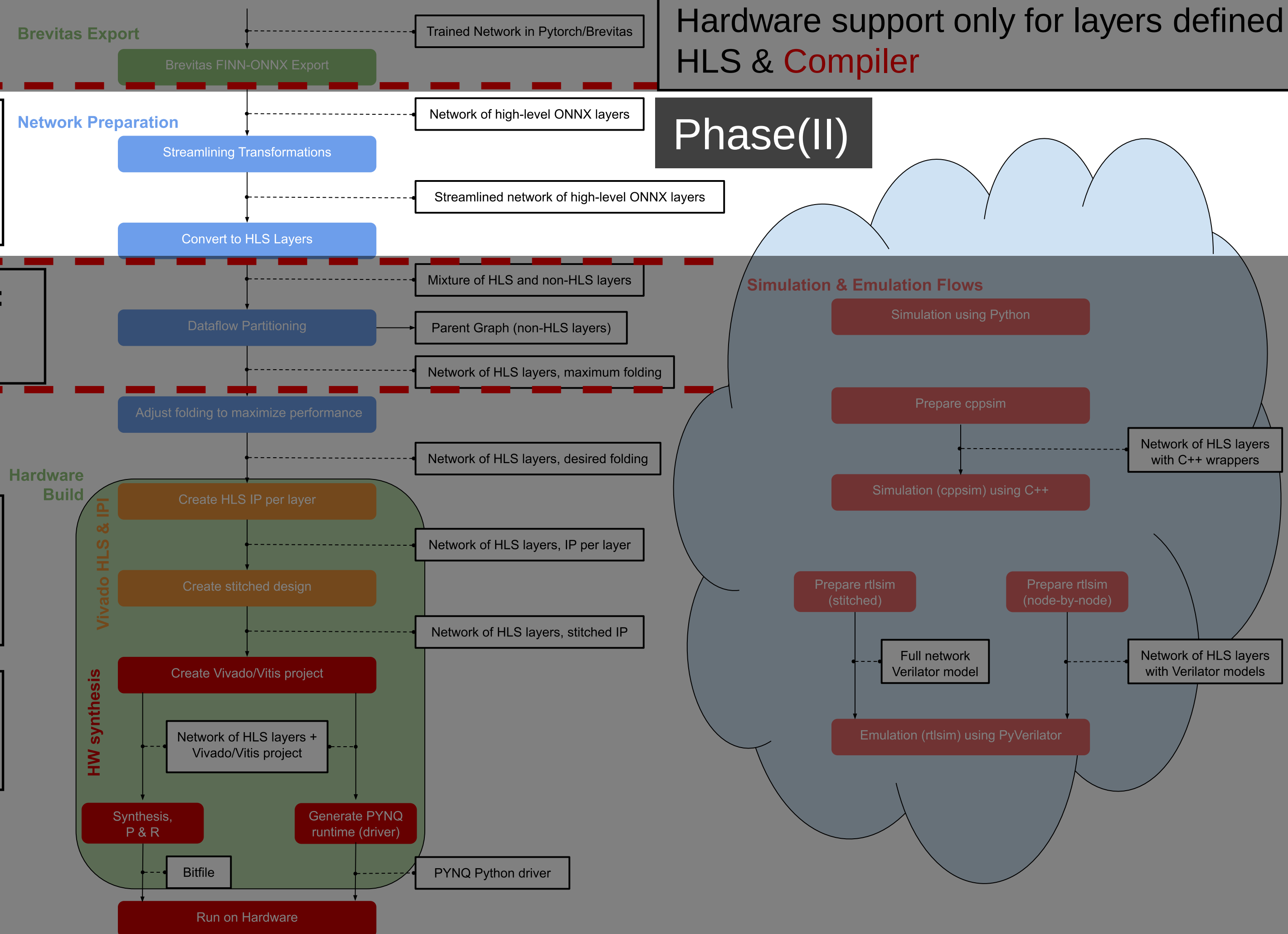
Phase(II)

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

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

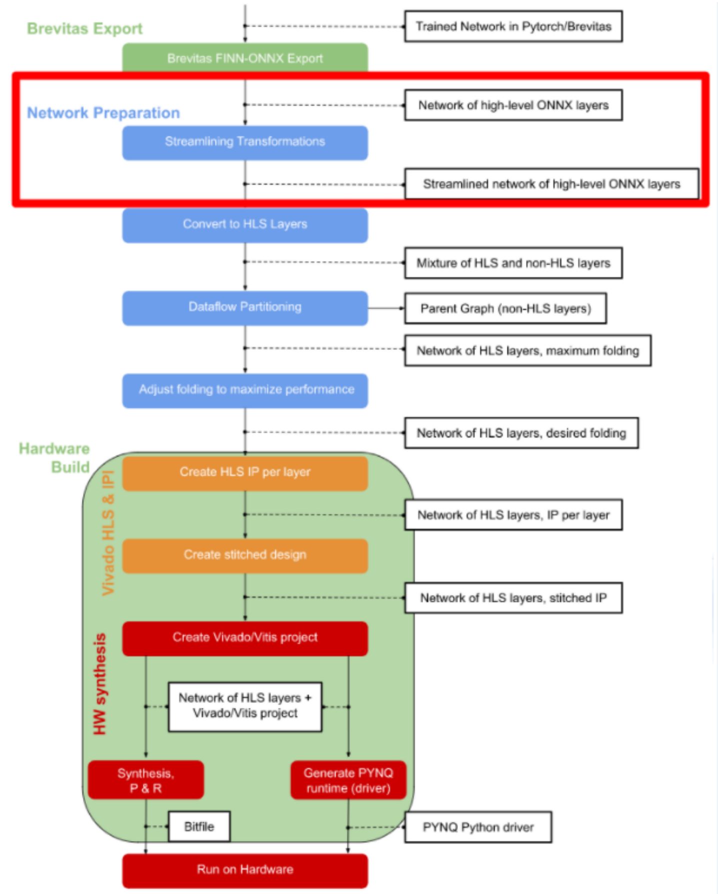
Coding: (HLSCustom Op): xxx.py (gen_code)

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



Phase (II): Network preparation

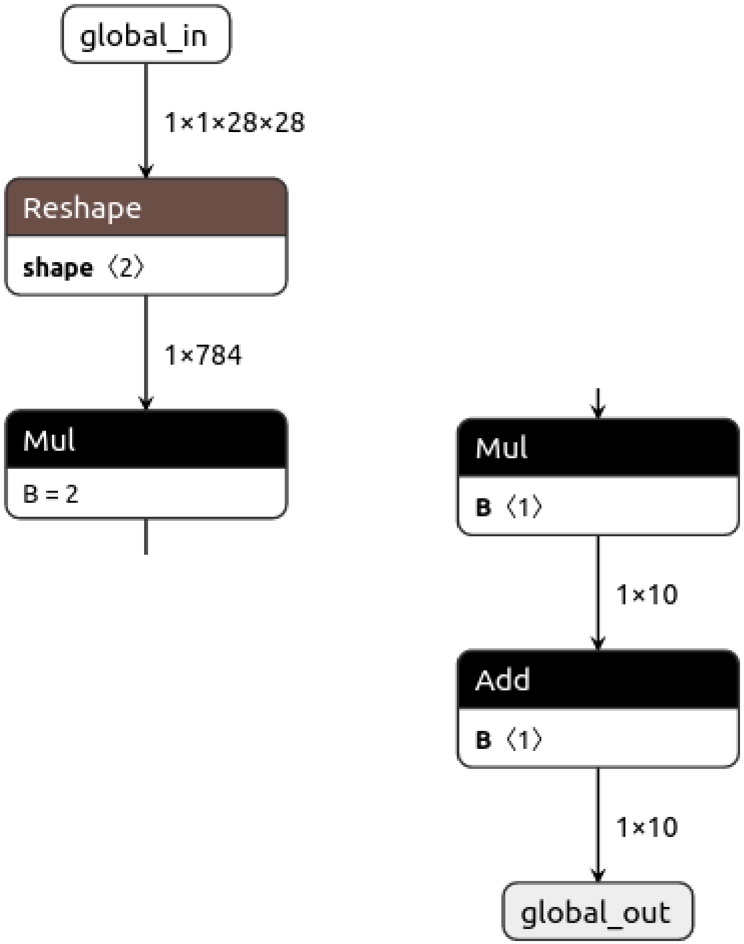
- (I) FINN-style Dataflow Architectures
- (II) Tidy-up transformations
- (III) Adding Pre- and Postprocessing
- (IV) Streamlining
- (V) Conversion to HLS layers
- (VI) Creating a Dataflow Partition
- (VII) Folding: Adjusting the Parallelism

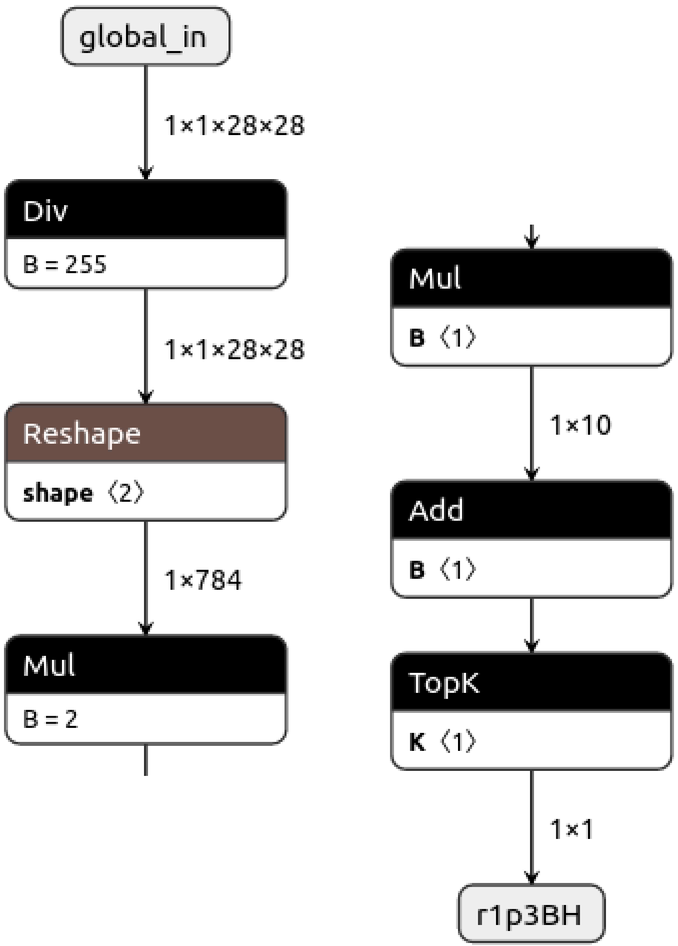


(II) Tidy-up transformations

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

(III) Adding Pre- and Postprocessing





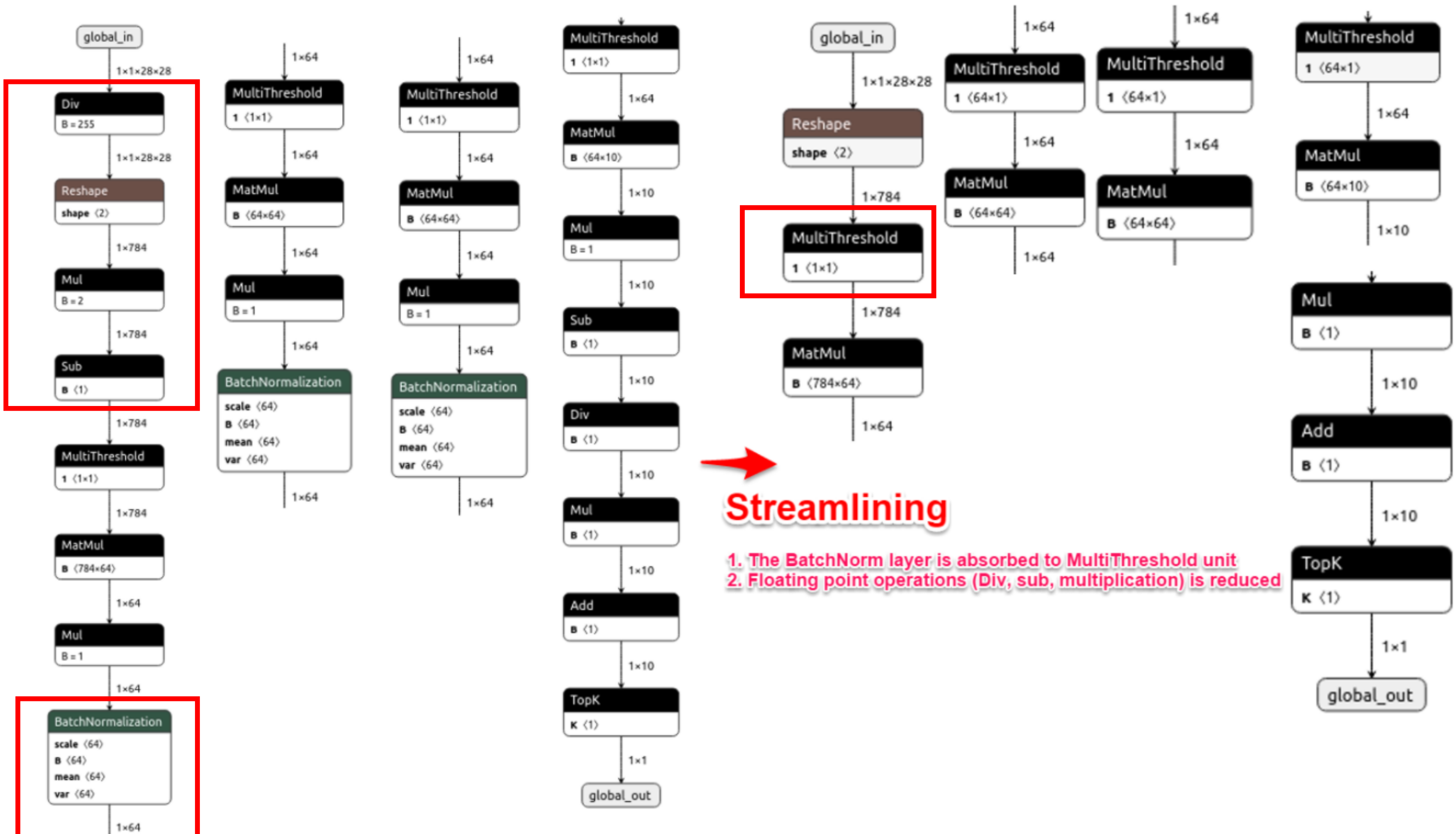
(IV) Streamlining(1/4)

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

```python
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)
```

```python
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")
```

(IV) Streamlining(2/4)

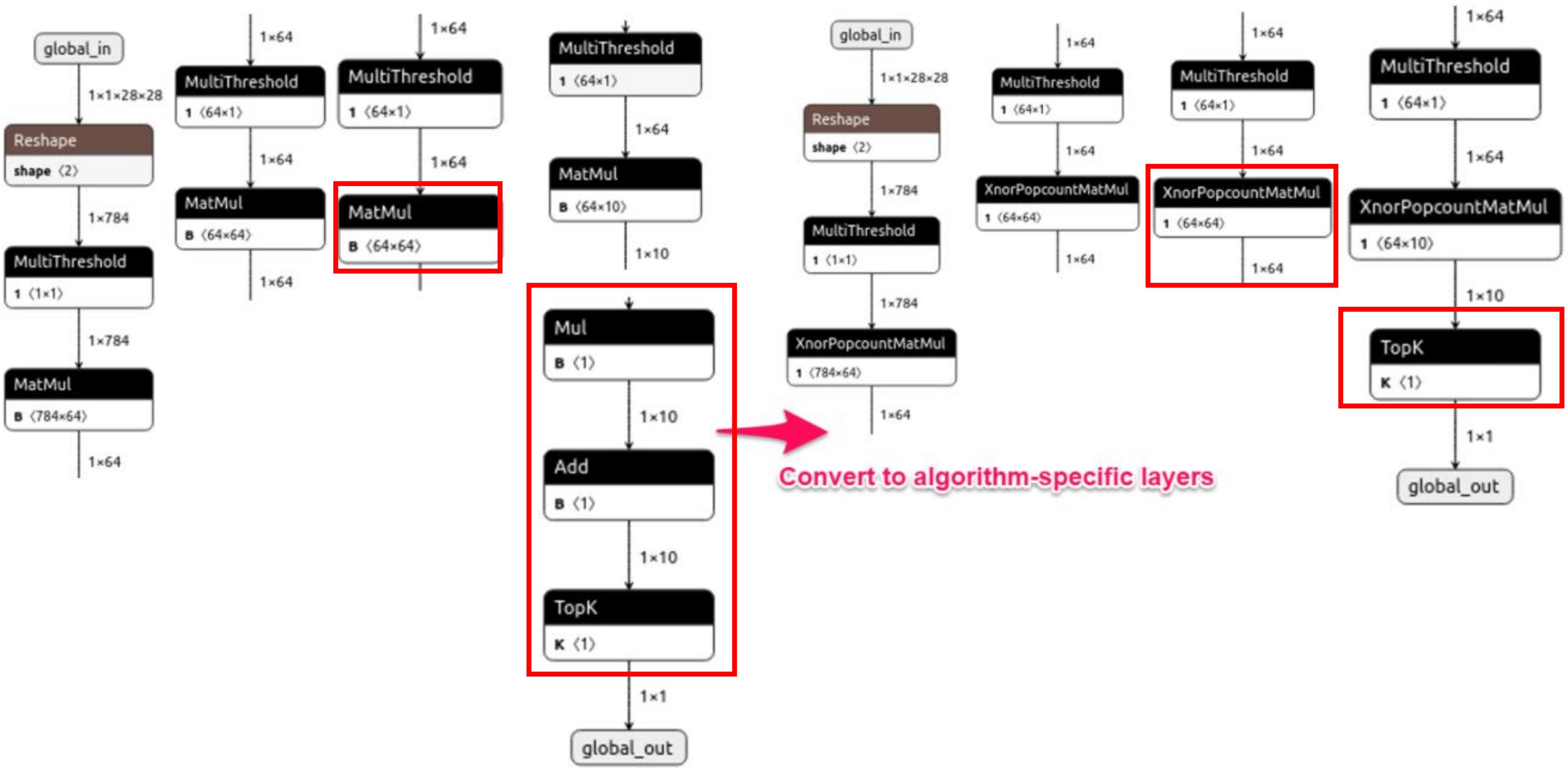


(IV) Streamlining(3/4)

- 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

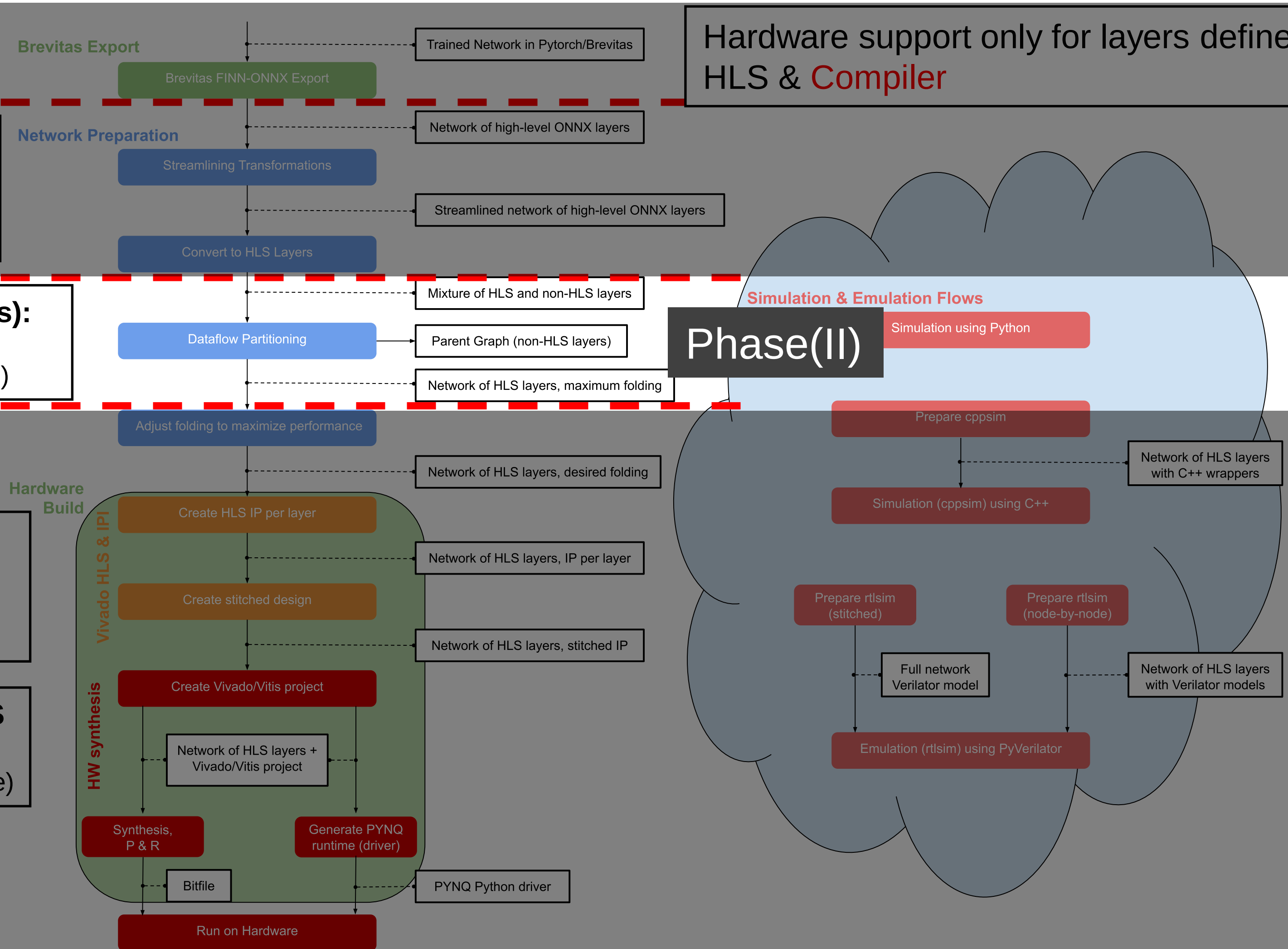
Hardware support only for layers defined in Finn-HLS & Compiler

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

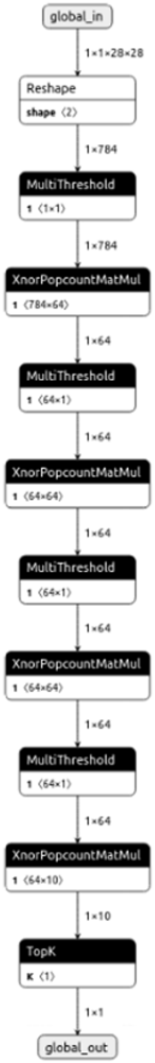
Coding (to-hls): Infer_xxx.py (adjusting nodes)

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

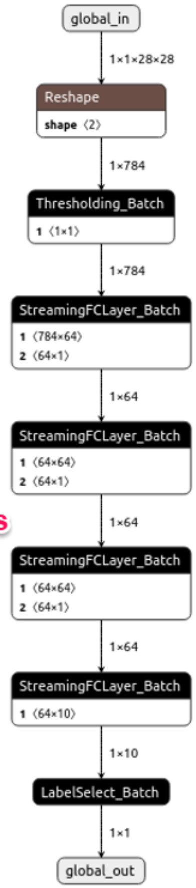
Coding: (HLS Library): xxx.cpp/.h (core)



(V) Conversion to HLS layers



Infer HLS Layers



NODE PROPERTIES

type	StreamingFCLayer_Batch
domain	finn.custom_op.fpgadataflow

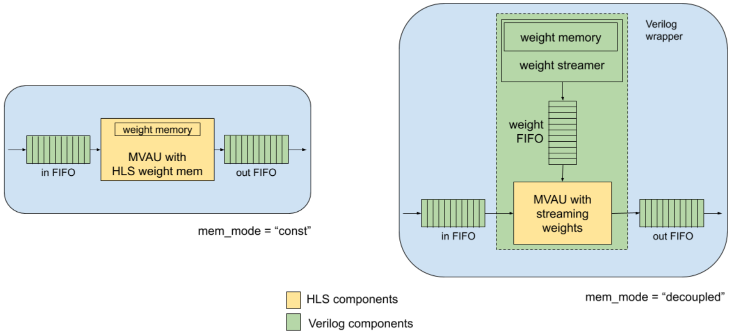
ATTRIBUTES

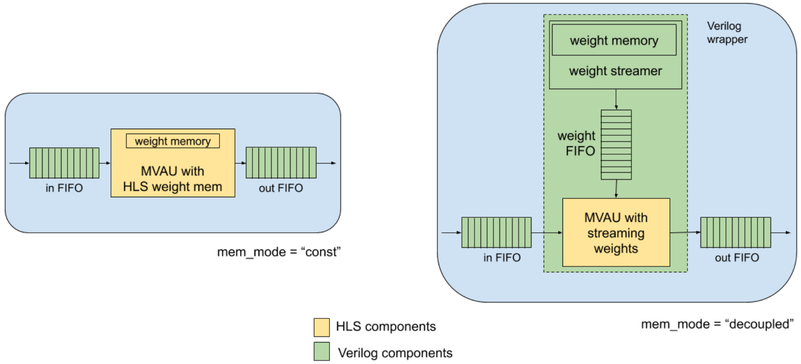
accDataType	UINT10
ActVal	0
backend	fpgadataflow
binaryXnorMode	1
inputDataType	BINARY
mem_mode	decoupled
MH	64
MW	784
noActivation	0
numInputVectors	1
outputDataType	BINARY
PE	1
SIMD	1
weightDataType	BINARY

INPUTS

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

OUTPUTS





mem_mode:

- const: 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)

XILINX FINN

(V) Conversion to HLS layers: Infer_xxx.py

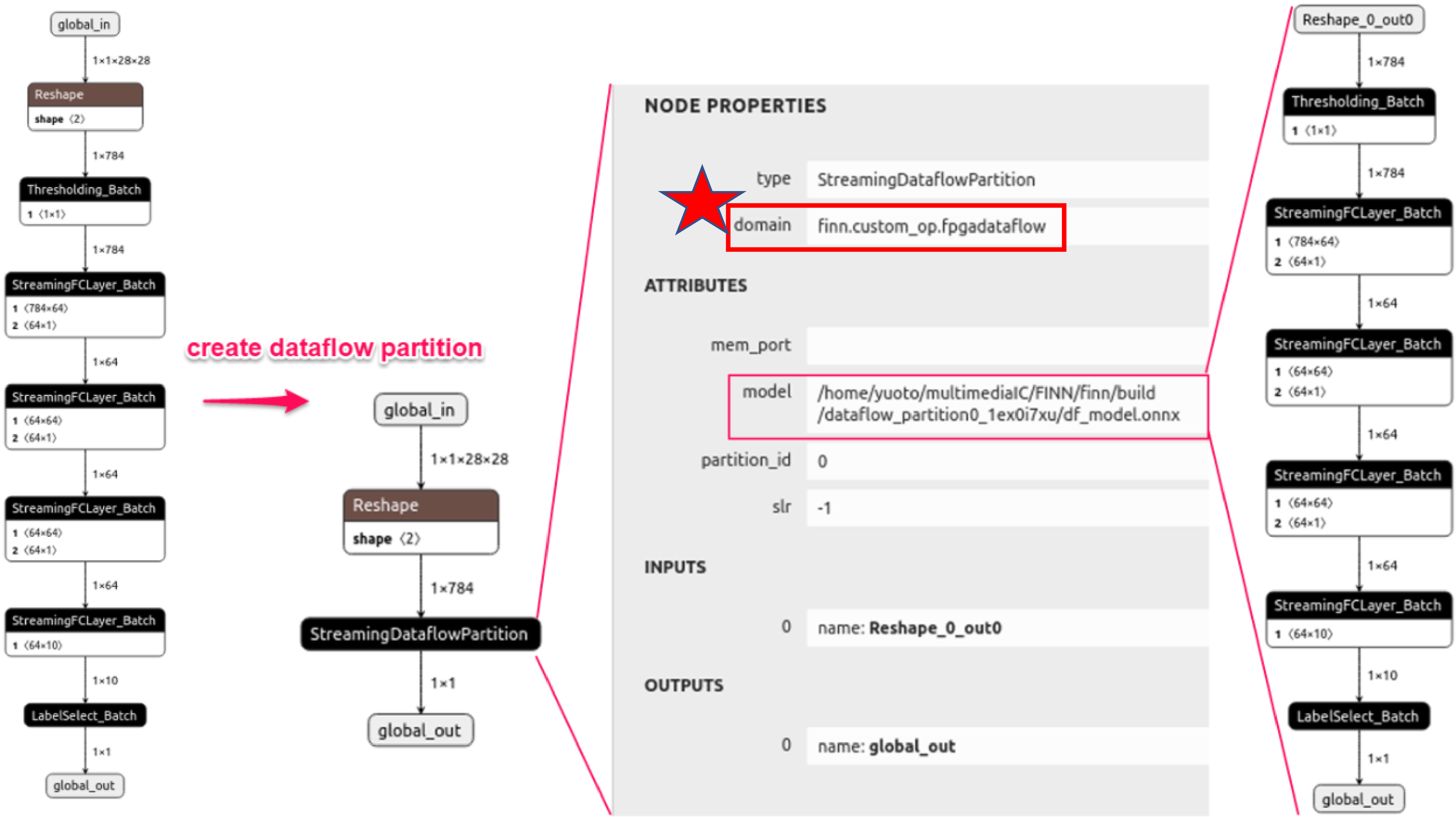
```python
import finn.transformation.fpgadataflow.convert_to_hls_layers as to_hls
model = ModelWrapper(build_dir+"/tfc_w1a1_ready_for_hls_conversion.onnx")
model = model.transform(to_hls.InferBinaryStreamingFCLayer("decoupled"))
# TopK to LabelSelect
model = model.transform(to_hls.InferLabelSelectLayer())
# 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")
```

```python
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
```

```python
# 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
```


BOL edu

(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)

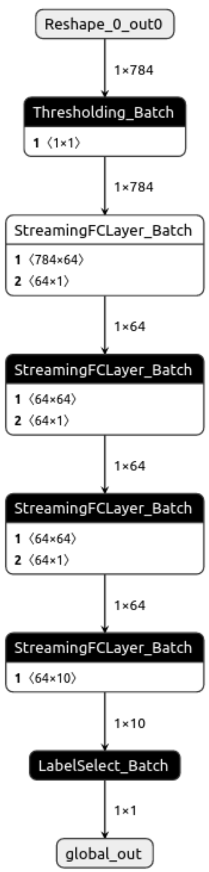
- Below:

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

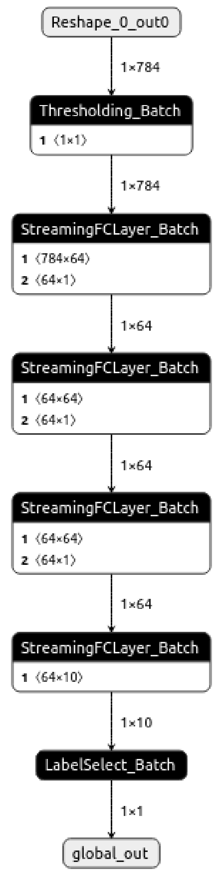
accDataType	UINT10
ActVal	0
backend	fpgadataflow
binaryXnorMode	1
inputDataType	BINARY
mem_mode	decoupled
MH	64
MW	784
noActivation	0
numInputVectors	1
outputDataType	BINARY
PE	1
SIMD	1
weightDataType	BINARY

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

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

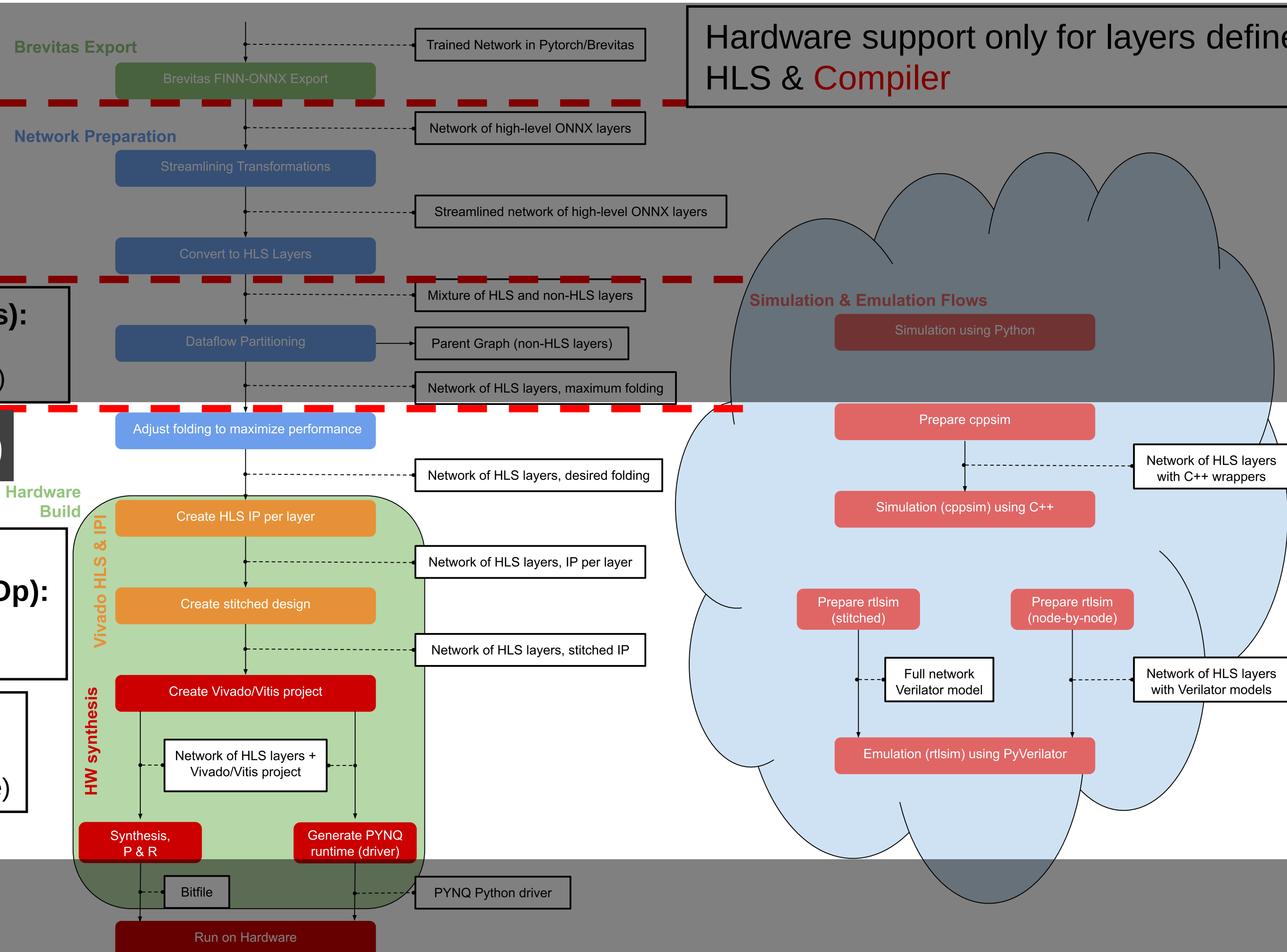
Don't touch: Since we are not concern of onnx operations

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

Phase(III)

Coding: (HLSCustomOp): xxx.py (gen_code)

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

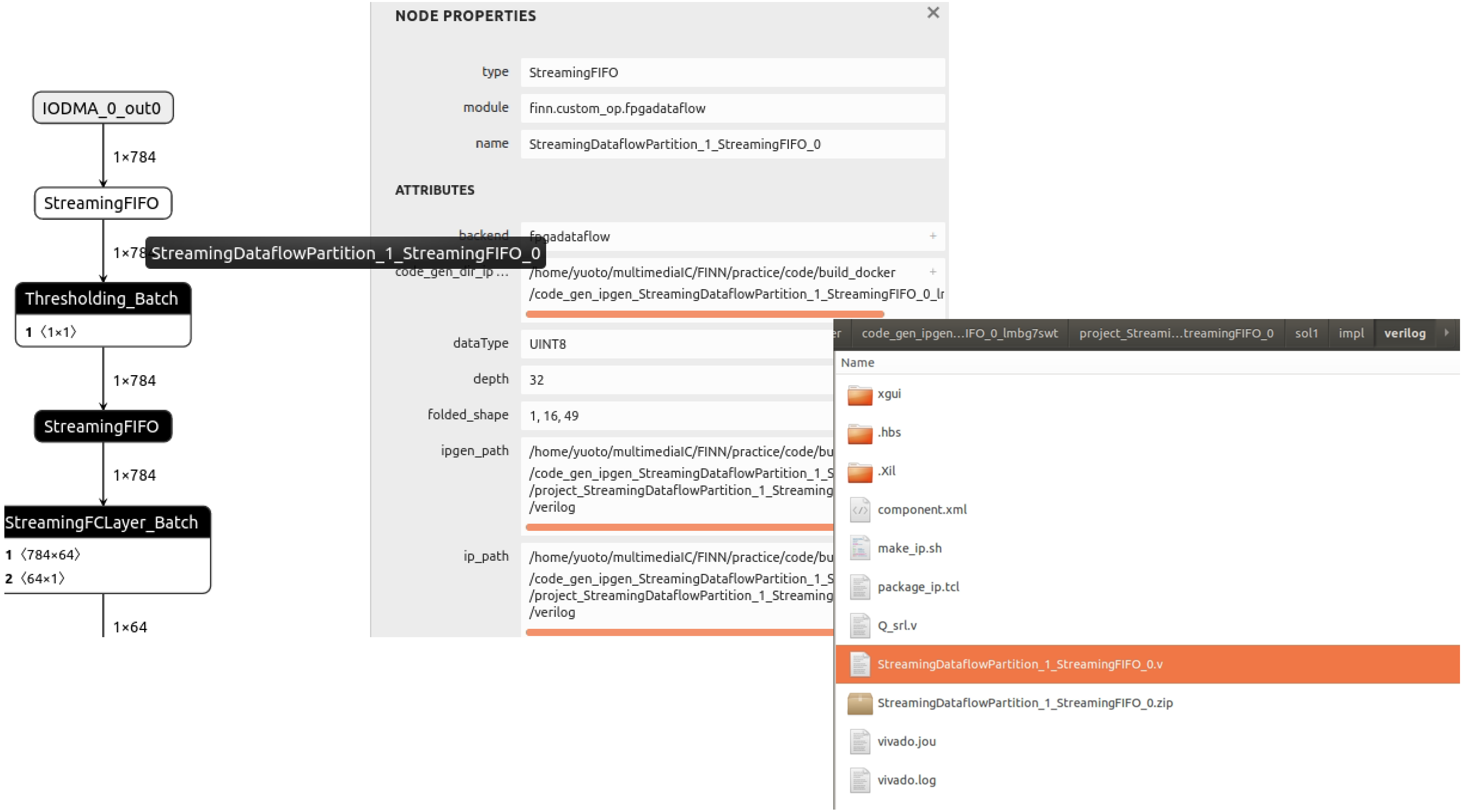


Phase (III): Hardware Build

- 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

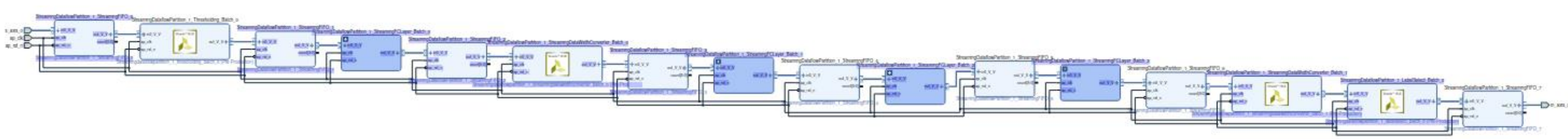

IODMA_0_out0
1×784
StreamingFIFO
1×784
StreamingDataflowPartition_1_StreamingFIFO_0
Thresholding_Batch
1 ⟨1×1⟩
1×784
StreamingFIFO
1×784
StreamingFCLayer_Batch
1 ⟨784×64⟩
2 ⟨64×1⟩
1×64
NODE PROPERTIES
type StreamingFIFO
module finn.custom_op.fpgadataflow
name StreamingDataflowPartition_1_StreamingFIFO_0
ATTRIBUTES
backend fpgadataflow
code_gen_dir_ip ... /home/yuoto/multimedialC/FINN/practice/code/build_docker
/code_gen_ipgen_StreamingDataflowPartition_1_StreamingFIFO_0_l
dataType UINT8
depth 32
folded_shape 1, 16, 49
ipgen_path /home/yuoto/multimedialC/FINN/practice/code/bu
/code_gen_ipgen_StreamingDataflowPartition_1_S
/project_StreamingDataflowPartition_1_Streaming
/verilog
ip_path /home/yuoto/multimedialC/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

4. MakeZYNQProject(2/2)

- Vivado project was automatically built and stitched

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

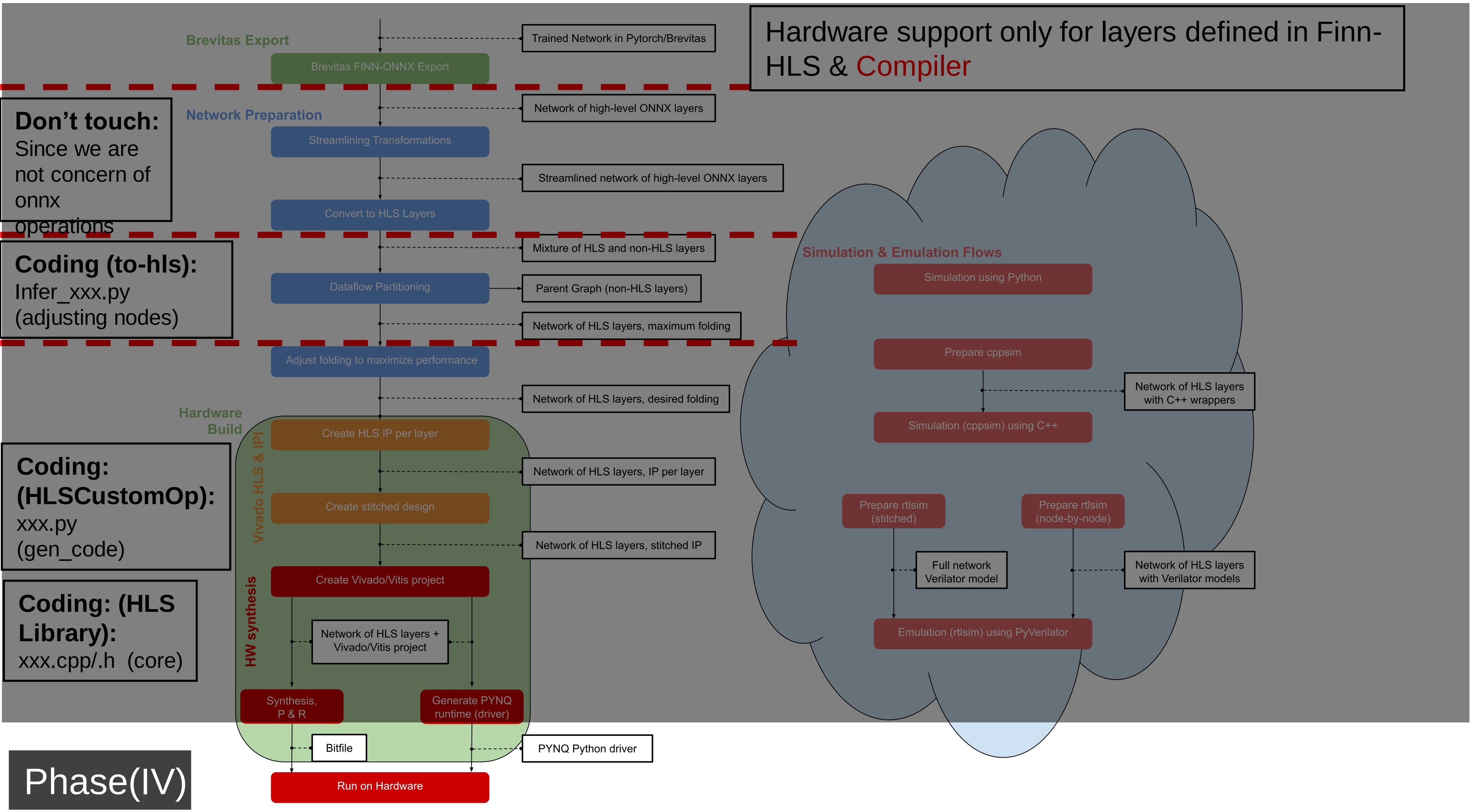
```
[key: "floorplan_json"
value: "/home/yuoto/multimediaIC/FINN/practice/code/build_docker/vitis_floorplan_zxz9em07/floorplan.json"
, key: "vivado_stitch_proj"
value: "/home/yuoto/multimediaIC/FINN/practice/code/build_docker/vivado_stitch_proj_ol2h0n49"
, key: "clk_ns"
value: "10"
, key: "wrapper_filename"
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): PYNQ deployment

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