Going from FINN to Galapagos

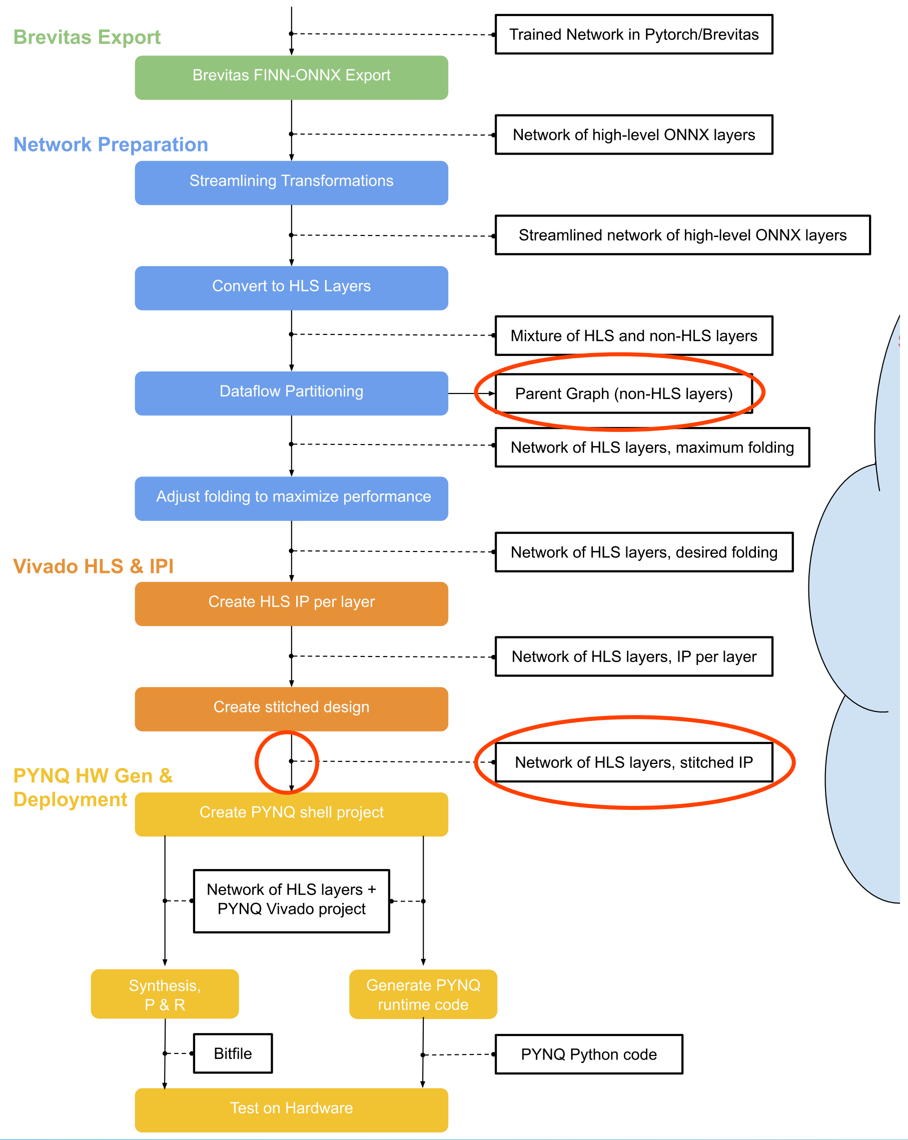
Objectives

- Explore connecting the FINN Quantized-Neural-Network (QNN) compiler with Galapagos, so that the compiled IP cores produced by FINN can be run on multiple devices

Background-FINN

- FINN: Open-Source compiler used to compile QNN’s into hardware IP cores that can be run on FPGAs
- Each generated IP core generally represents a single node in the QNN (popcount/FIFO, etc), allowing for the possibility of splitting compute across multiple devices
- Although QNN’s are supported, main focus is Binary-Neural-Networks (BNN’s)

Background-End2End Flow

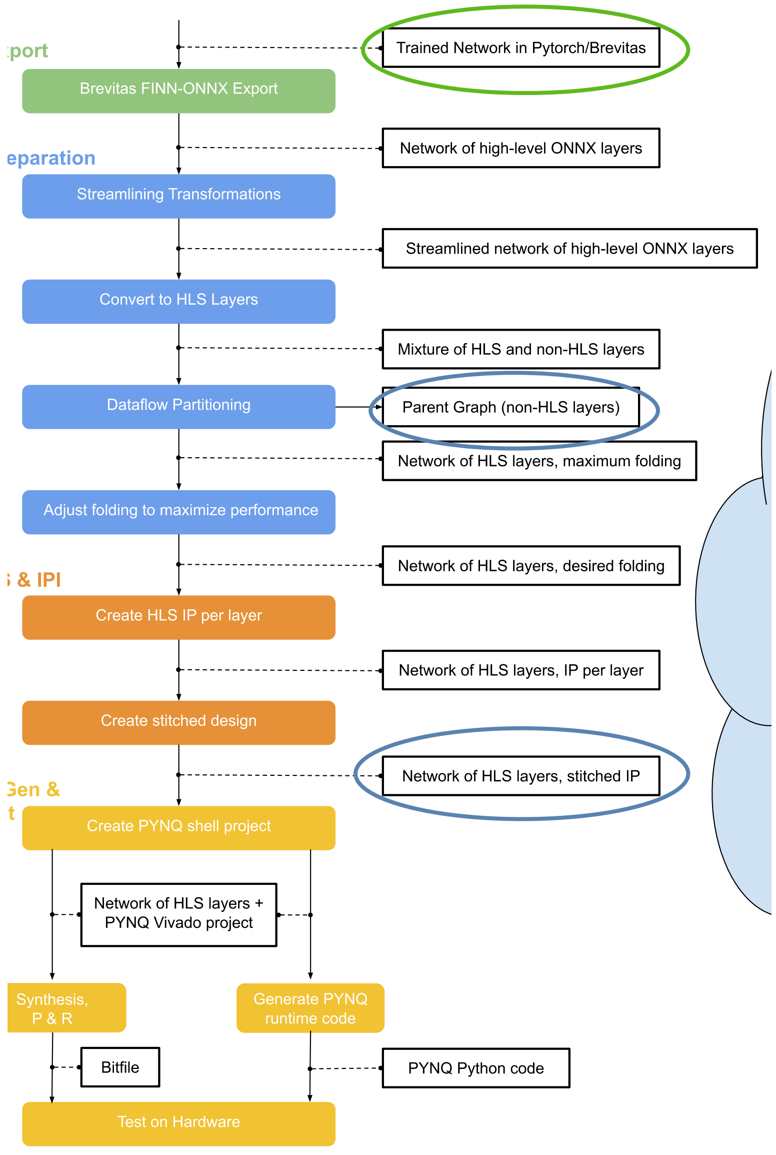


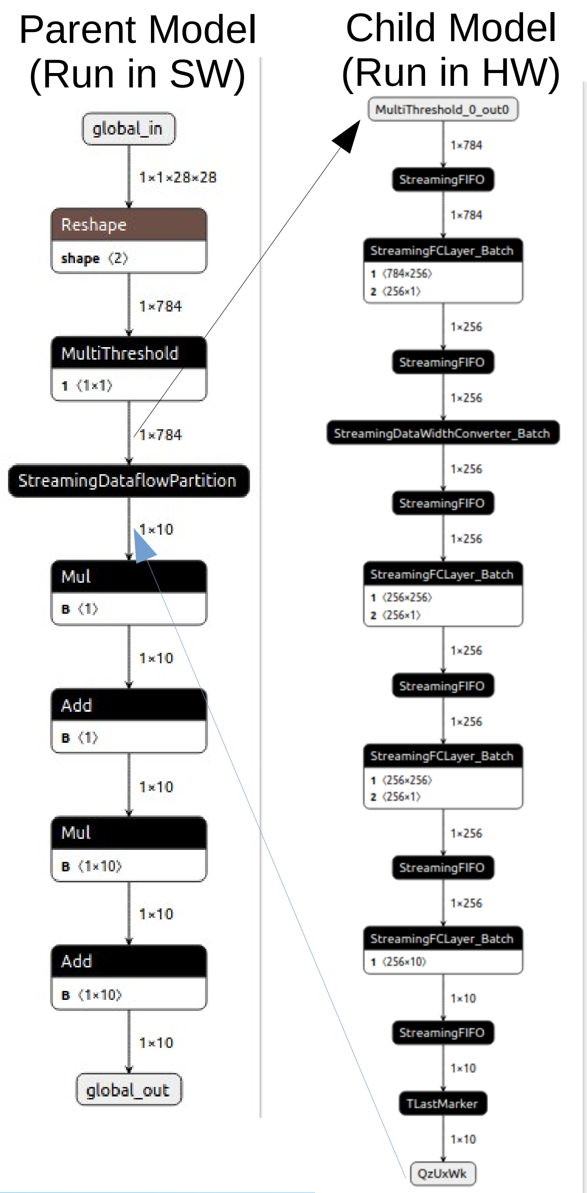
Brevitas Export
Trained Network in Pytorch/Brevitas
Brevitas FINN-ONNX Export
Network Preparation
Network of high-level ONNX layers
Streamlining Transformations
Streamlined network of high-level ONNX layers
Convert to HLS Layers
Mixture of HLS and non-HLS layers
Dataflow Partitioning
Parent Graph (non-HLS layers)
Network of HLS layers, maximum folding
Adjust folding to maximize performance
Network of HLS layers, desired folding
Vivado HLS & IPI
Create HLS IP per layer
Network of HLS layers, IP per layer
Create stitched design
Network of HLS layers, stitched IP
PYNQ HW Gen & Deployment
Create PYNQ shell project
Network of HLS layers + PYNQ Vivado project
Synthesis, P & R
Generate PYNQ runtime code
Bitfile
PYNQ Python code
Test on Hardware

End-to-End Flow:

- Input: Pretrained QNN PyTorch Model
- Outputs:
  - parent_model: ONNX model of all nodes that can't be accelerated
  - child_model: ONNX model of all nodes that can be accelerated
  - Vivado Project of child_model, with Block Design of the nodes converted to IP, stitched together
- FINN can also execute the parent_model, running the accelerated child_model on a Xilinx Pynq Board
- (Not Pictured) FINN also offers some functional verification tools to verify the outputs of different stages of the flow

Background-Input/Output Models


port
Trained Network in Pytorch/Brevitas
Brevitas FINN-ONNX Export
eparation
Network of high-level ONNX layers
Streamlining Transformations
Streamlined network of high-level ONNX layers
Convert to HLS Layers
Mixture of HLS and non-HLS layers
Dataflow Partitioning
Parent Graph (non-HLS layers)
Network of HLS layers, maximum folding
Adjust folding to maximize performance
& IPI
Network of HLS layers, desired folding
Create HLS IP per layer
Network of HLS layers, IP per layer
Create stitched design
Network of HLS layers, stitched IP
Gen &
Create PYNQ shell project
Network of HLS layers + PYNQ Vivado project
Synthesis, P & R
Generate PYNQ runtime code
Bitfile
PYNQ Python code
Test on Hardware



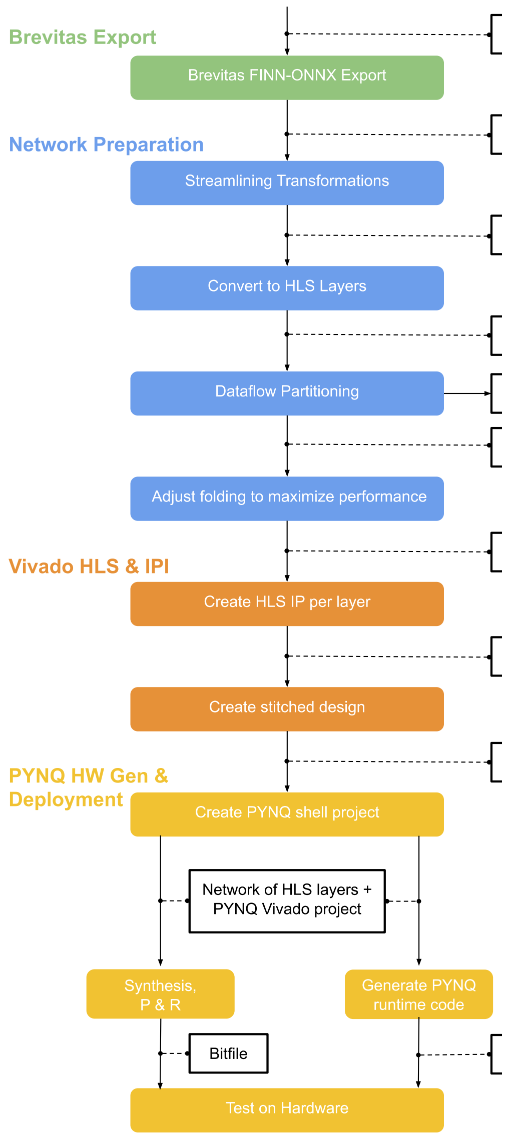
Input Model:

- Exported as Open-Neural-Network-Exchange (ONNX) format, with modifications
- Weights are quantized before export
- FINN-specific annotations added to enable binary quantization

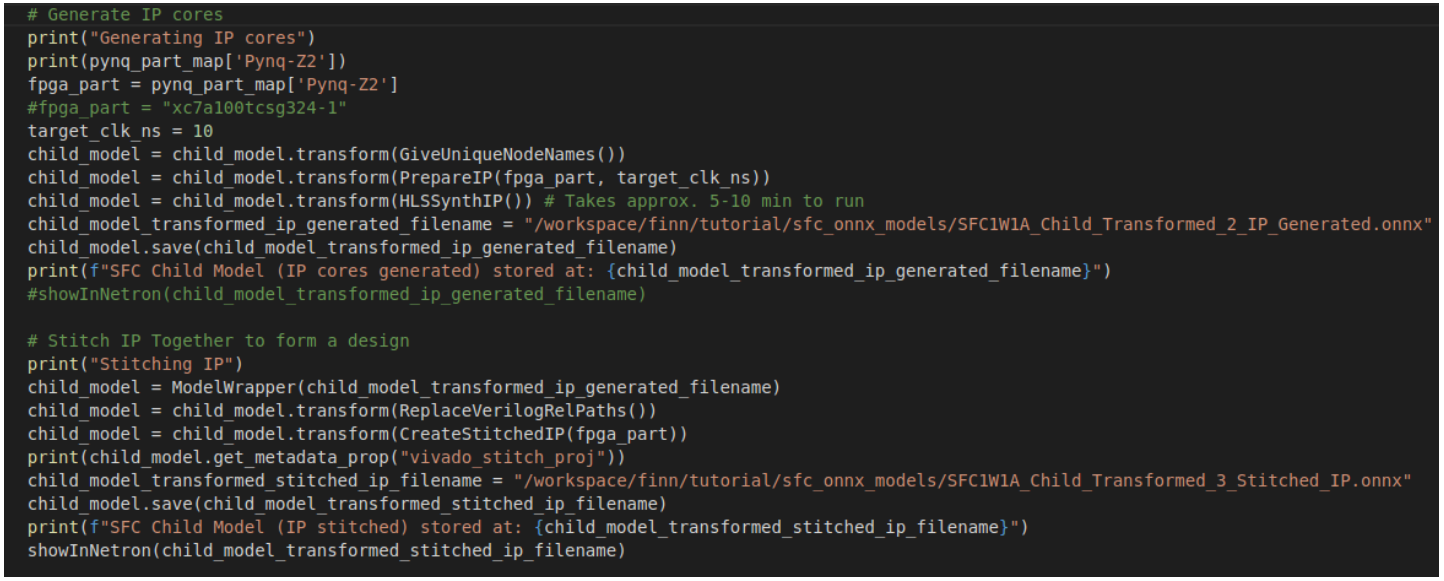
Parent/Child Models:

- Saved as ONNX format
- FINN's ONNX-Exec functions can deploy the child model to FPGA and run the parent model
- ONNX-Exec functions will switch to running child_model automatically when it arrives at the StreamingDataflowPartition
- To explore: What if there's more than one child model?

Background-Implementation

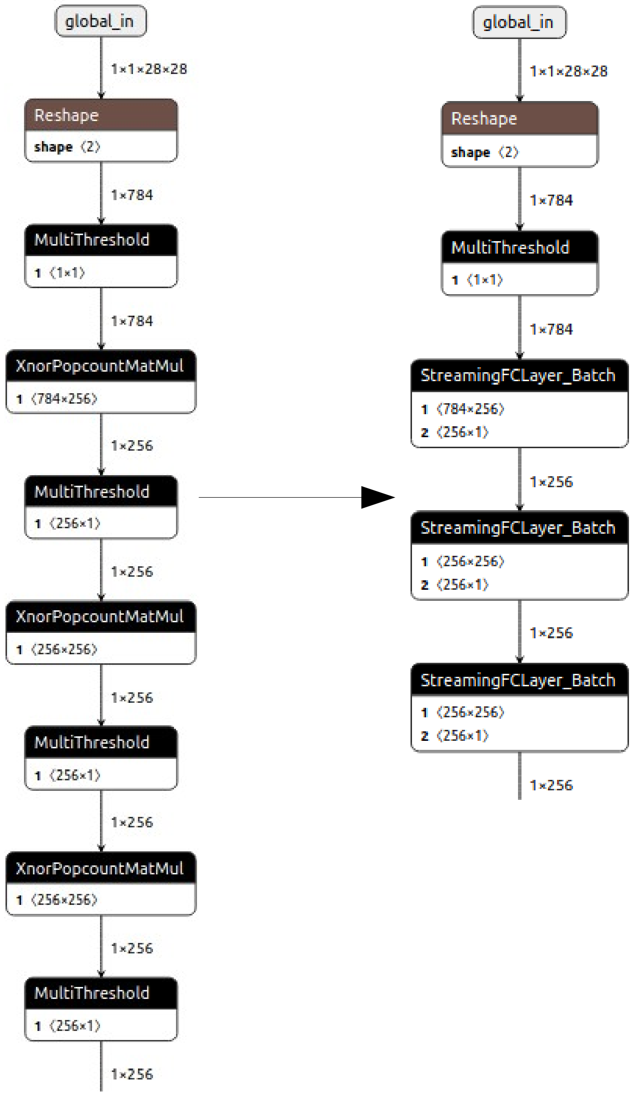


- FINN functions are called in Python
- ONNX model is treated like a class
- Steps in Network Preparation, Vivado HLS Generation and Deployment are performed by applying FINN Transform passes on the model
- Custom Transform/Analysis passes are supported
- Allows users to access the model/flow at any stage

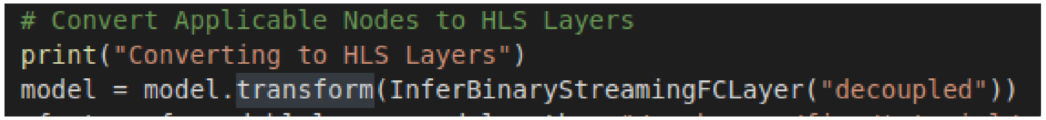


```python
# Generate IP cores
print("Generating IP cores")
print(pynq_part_map['Pynq-Z2'])
fpga_part = pynq_part_map['Pynq-Z2']
#fpga_part = "xc7a100tcsg324-1"
target_clk_ns = 10
child_model = child_model.transform(GiveUniqueNodeNames())
child_model = child_model.transform(PrepareIP(fpga_part, target_clk_ns))
child_model = child_model.transform(HLSSynthIP()) # Takes approx. 5-10 min to run
child_model_transformed_ip_generated_filename = "/workspace/finn/tutorial/sfc_onnx_models/SFC1W1A_Child_Transformed_2_IP_Generated.onnx"
child_model.save(child_model_transformed_ip_generated_filename)
print(f"SFC Child Model (IP cores generated) stored at: {child_model_transformed_ip_generated_filename}")
#showInNetron(child_model_transformed_ip_generated_filename)

# Stitch IP Together to form a design
print("Stitching IP")
child_model = ModelWrapper(child_model_transformed_ip_generated_filename)
child_model = child_model.transform(ReplaceVerilogRelPaths())
child_model = child_model.transform(CreateStitchedIP(fpga_part))
print(child_model.get_metadata_prop("vivado_stitch_proj"))
child_model_transformed_stitched_ip_filename = "/workspace/finn/tutorial/sfc_onnx_models/SFC1W1A_Child_Transformed_3_Stitched_IP.onnx"
child_model.save(child_model_transformed_stitched_ip_filename)
print(f"SFC Child Model (IP stitched) stored at: {child_model_transformed_stitched_ip_filename}")
showInNetron(child_model_transformed_stitched_ip_filename)
```

Background-HLS Layers

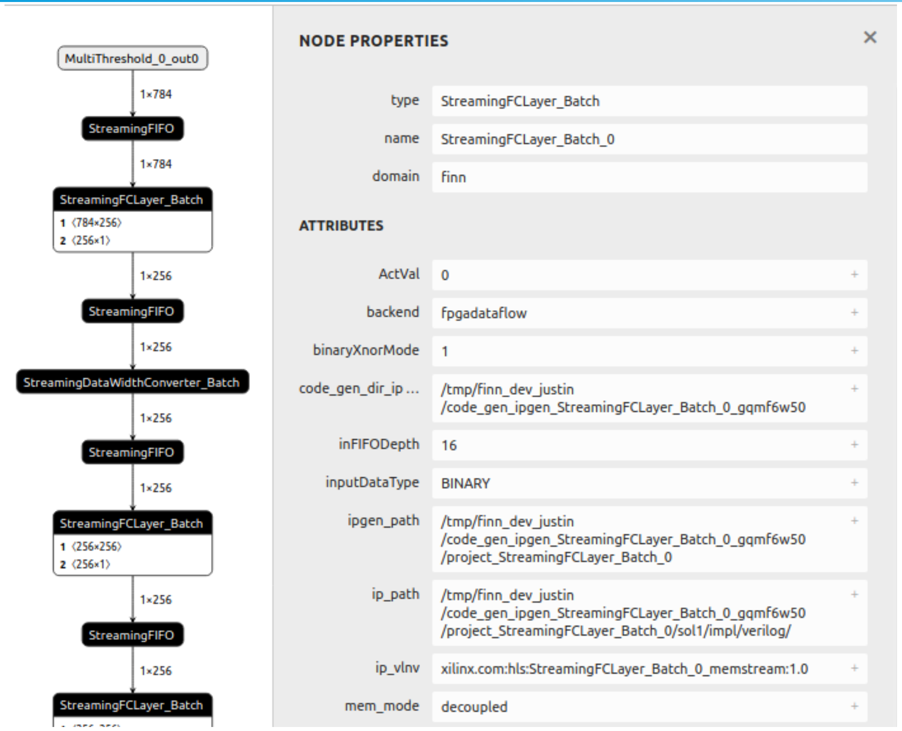


- Transform applied to ONNX Model
- FINN goes through the model, and replaces any applicable nodes with nodes indicating they will be run on HLS
- Not all nodes can be accelerated
  - Each node type that can be converted requires its own transform
  - IP cores are synthesized based on a library of HLS primitives



```
# Convert Applicable Nodes to HLS Layers
print("Converting to HLS Layers")
model = model.transform(InferBinaryStreamingFCLayer("decoupled"))
```

Background-IP Generation



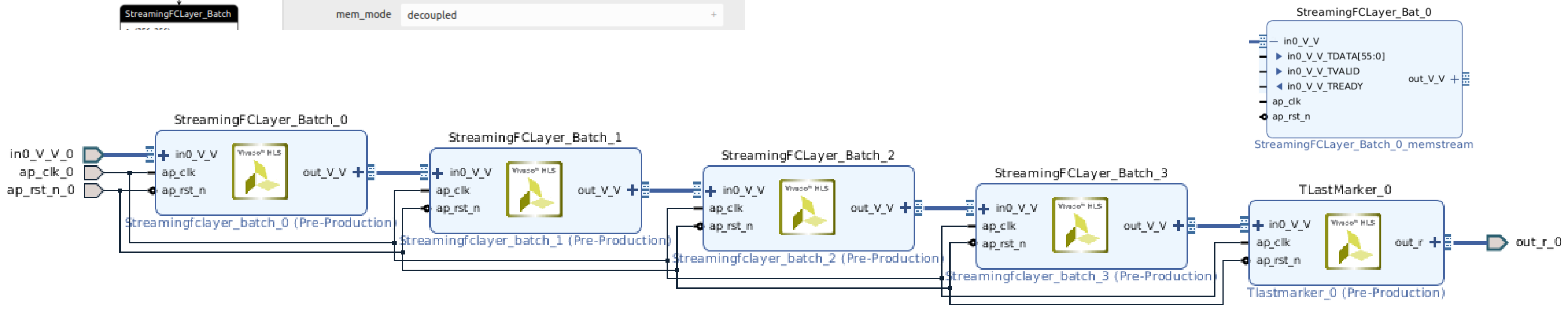
NODE PROPERTIES

type	StreamingFCLayer_Batch
name	StreamingFCLayer_Batch_0
domain	finn

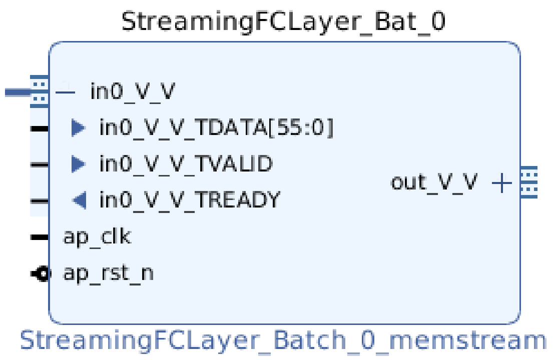
ATTRIBUTES

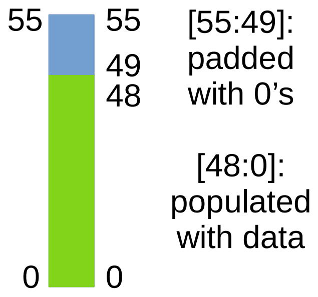
ActVal	0
backend	fpgadataflow
binaryXnorMode	1
code_gen_dir_ip ...	/tmp/finn_dev_justin /code_gen_ipgen_StreamingFCLayer_Batch_0_gqmf6w50
inFIFODepth	16
inputDataType	BINARY
ipgen_path	/tmp/finn_dev_justin /code_gen_ipgen_StreamingFCLayer_Batch_0_gqmf6w50 /project_StreamingFCLayer_Batch_0
ip_path	/tmp/finn_dev_justin /code_gen_ipgen_StreamingFCLayer_Batch_0_gqmf6w50 /project_StreamingFCLayer_Batch_0/sol1/impl/verilog/
ip_vlnv	xilinx.com:hls:StreamingFCLayer_Batch_0_memstream:1.0
mem_mode	decoupled

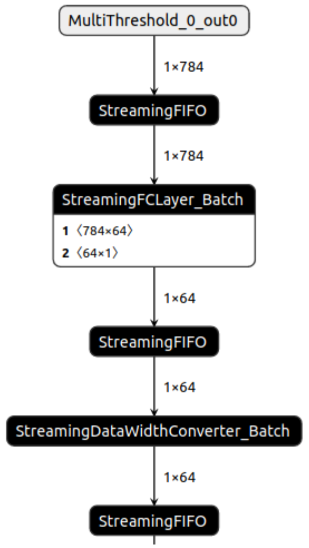
- Run on child_model only
- Each node in child_model becomes a unique IP Core (images shown are from different models)
- ONNX model stores path to IP Core files in ipgen_path attribute
- Each IP core that is generated is connected using the AXI-Stream protocol (but no last signal?)

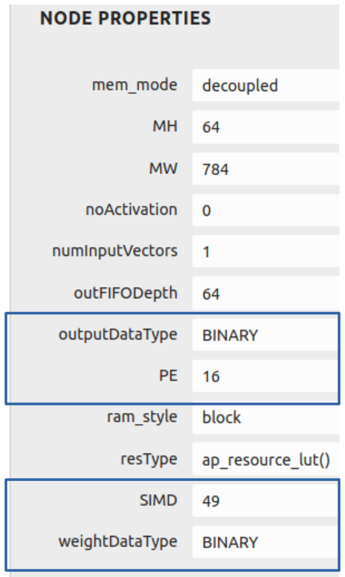


Background – FINN SFCL Cores





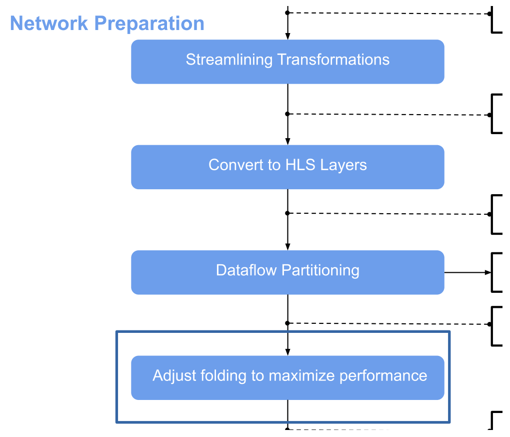


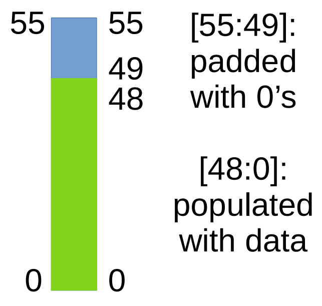

NODE PROPERTIES

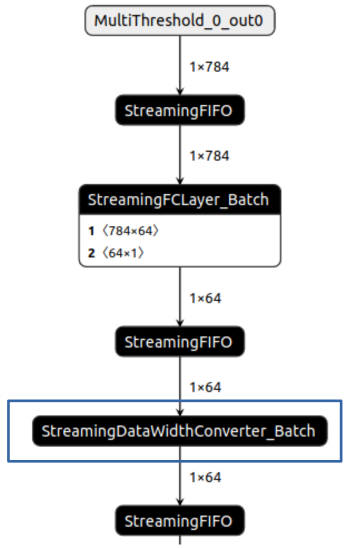
mem_mode	decoupled
MH	64
MW	784
noActivation	0
numInputVectors	1
outFIFODepth	64
outputDataType	BINARY
PE	16
ram_style	block
resType	ap_resource_lut()
SIMD	49
weightDataType	BINARY

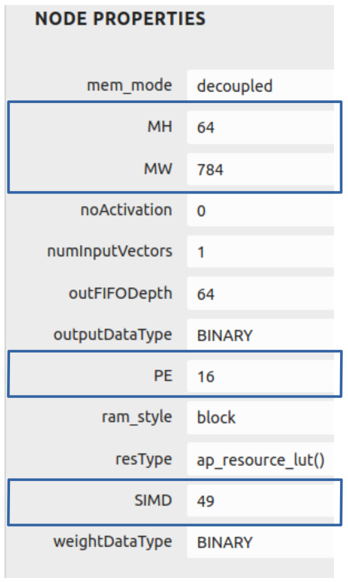
- StreamingFCLayer (SFCL) Cores perform the computations of one layer in the BNN
- Communication: AXI-Stream
- Communication Width:
  - Communication width is governed by Folding Factors, allows us to trade off between parallelism (performance) and on-chip area
  - **SIMD**: represents number of input lanes inside the core (number of input bits processed in one cycle) -> governs **input width**
  - **PE**: represents number of processing elements inside the core (number of output bits computed in one cycle) -> governs **output width**
  - Input data width = SIMD * weightDataType, output data width = PE * outputDataType, **rounded up to the nearest byte** due to AXI-Stream protocol
  - Factors are stored and can be found in the ONNX model (see left)
  - Bits are populated from LSB, additional bits are padded with 0's (see example above for 49-bit vector transmitted over 56-bit data bus)

BG: SFCL Comm Width Cont’d


Network Preparation
Streamlining Transformations
Convert to HLS Layers
Dataflow Partitioning
Adjust folding to maximize performance







NODE PROPERTIES

mem_mode	decoupled
MH	64
MW	784
noActivation	0
numInputVectors	1
outFIFODepth	64
outputDataType	BINARY
PE	16
ram_style	block
resType	ap_resource_lut()
SIMD	49
weightDataType	BINARY

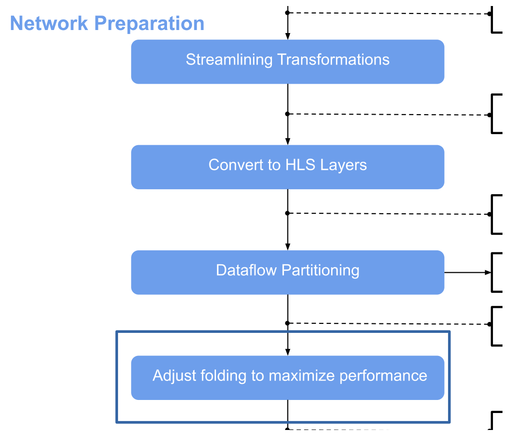
Communication Width Continued:

- Transfers will be **evenly populated** (eg. Each transfer on the left will contain 49 bits, rather than filling every transfer with 56 bits and having a final transfer of uneven length)
- Setting the Folding Factors is done by the user in FINN (**Adjusting Folding** stage of network prep)
- **Requirement:**
  - MW: Matrix Width, represents total width of input vector to this layer (in bits)
  - MH: Matrix Width, represents total width of output vector from this layer (in bits)
  - **MW % SIMD == 0, MH % PE == 0**
  - SIMD must divide input feature width, PE must divide output feature width with no remainder
- Output width of core A and input width of core B **do not have to match**, FINN will insert data width converters automatically

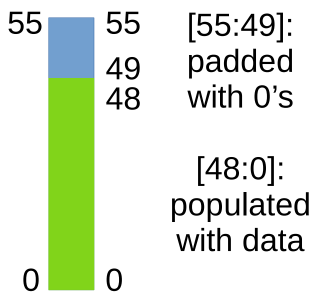
More info on Folding Factors:

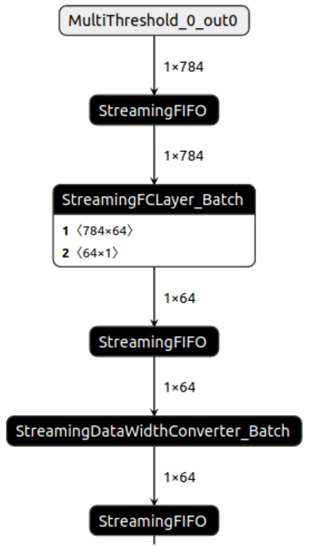
- https://github.com/Xilinx/finn/blob/dev/docs/finn-sheduling-and-folding.pptx
- https://arxiv.org/abs/1612.07119

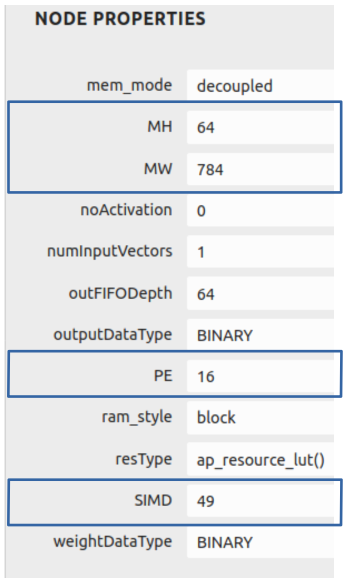
Background - SFCL Num Transfers



Network Preparation
Streamlining Transformations
Convert to HLS Layers
Dataflow Partitioning
Adjust folding to maximize performance







NODE PROPERTIES

mem_mode	decoupled
MH	64
MW	784
noActivation	0
numInputVectors	1
outFIFODepth	64
outputDataType	BINARY
PE	16
ram_style	block
resType	ap_resource_lut()
SIMD	49
weightDataType	BINARY

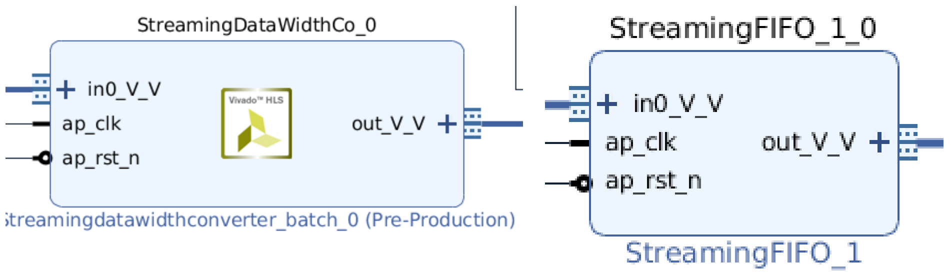
Number of AXI-Stream transfers:

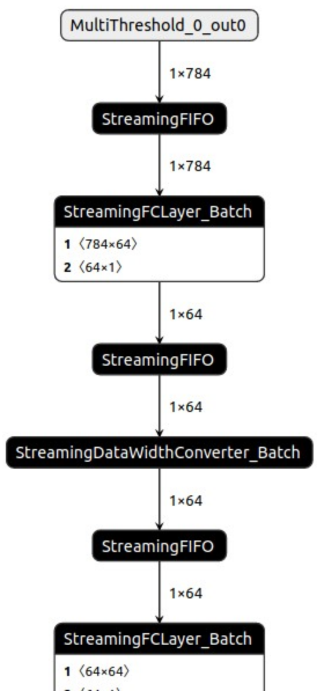
- Instead of using a TLAST signal, SFCL cores are programmed by FINN to recognize a fixed number of transfers as a single input vector, and will transmit a fixed number of transfers as an output packet
- Governed by SIMD, PE, MW, MH
- Each input MW is broken up into evenly populated transfers of SIMD bits, same with MH and PE.
- **Number of input transfers = MW / SIMD**
- **Number of output transfers = MH / PE**
- **Important:** Number of input and output transfers depends on SIMD and PE, **not** on the actual width of the data bus
  - Eg. The core on the left has MW of 784 bits, which must be divided by SIMD of 49 (16 transfers), not data width of 56

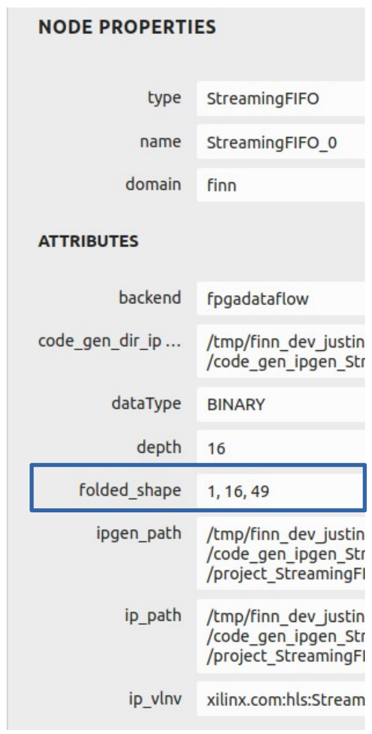
More info on Folding Factors:

- https://github.com/Xilinx/finn/blob/dev/docs/finn-sheduling-and-folding.pptx
- https://arxiv.org/abs/1612.07119

Background – Other FINN Cores






NODE PROPERTIES

type	StreamingFIFO
name	StreamingFIFO_0
domain	finn

ATTRIBUTES

backend	fpgadataflow
code_gen_dir_ip ...	/tmp/finn_dev_justin /code_gen_ipgen_Str
dataType	BINARY
depth	16
folded_shape	1, 16, 49
ipgen_path	/tmp/finn_dev_justin /code_gen_ipgen_Str /project_StreamingFI
ip_path	/tmp/finn_dev_justin /code_gen_ipgen_Str /project_StreamingFI
ip_vlnv	xilinx.com:hls:Stream

- Perform a variety of functions, like FIFO or data width conversion, which support the SFCL cores
- Communicate using AXI-Stream
- AXI-Stream Data width and number of transfers is determined by preceding/following SFCL cores (eg. This FIFO expects 16 transfers, 49 bits in each transfer, which will be rounded to 56 bits)

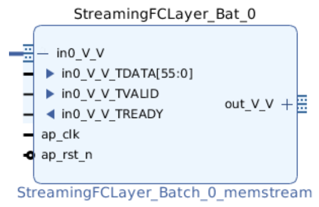
Plan P.1

- Target connection to occur in the Galapagos Middleware layer
  - Middleware allows for specification of individual kernels (IP cores, in FINN’s case), as well as specifying how kernels are connected together
  - FINN IP cores match Galapagos kernel I/O
    - Contain CLK, ARESETN, IN-stream, and OUT-stream (using AXI-Stream protocol)
  - Middleware Layer allows us to connect HW cores with the unaccelerated SW Cores

Plan: What are our Inputs?

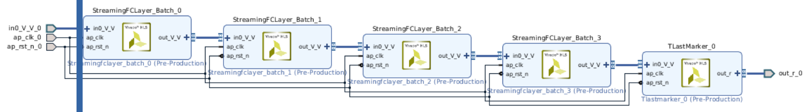
Individual IP Cores:

- Most flexible (any program that generates IP cores could reuse this bridge, not just FINN)
- Would require list of IP cores with details such as project/file location, **connections**, etc



IP Block Design:

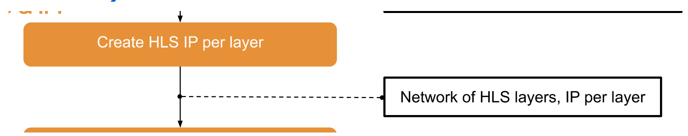
- All connections have already been made for us
- If we want to split IP cores into kernels, would require us to deconstruct the project
- May not be as flexible (deconstructing the project may require project-specific commands)



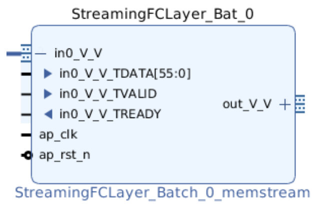
Selected due to flexibility and simplicity

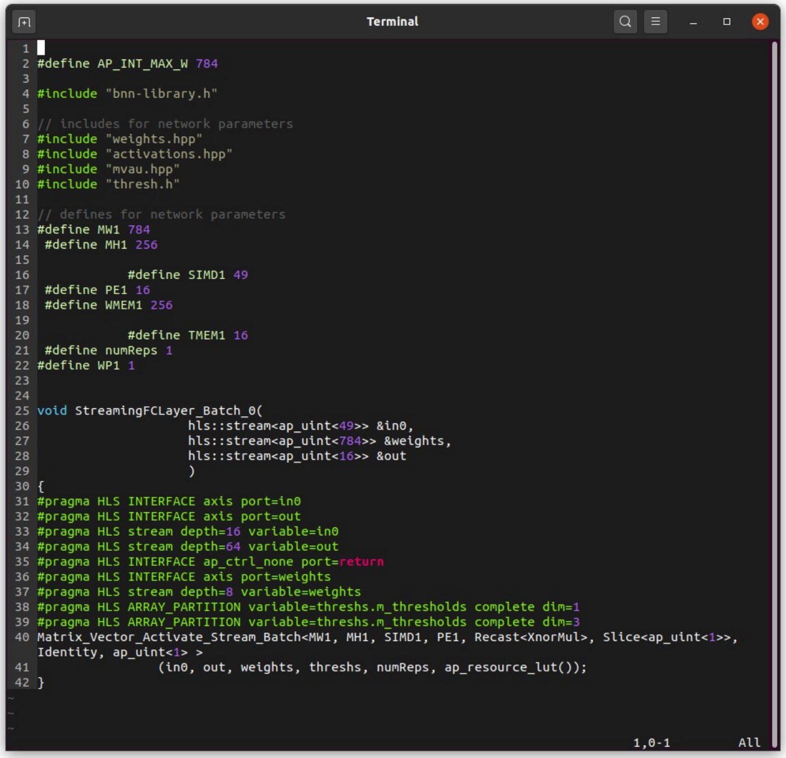
C++ Kernels

- Would allow us to run kernels on both CPU or HW
- Would require list of kernels with details such as file location, **connections**, etc
- We would need to perform HLS on all kernels that run on HW
- Some kernels might require special files/commands during HLS (haven't found any so far)



Plan: Inputs





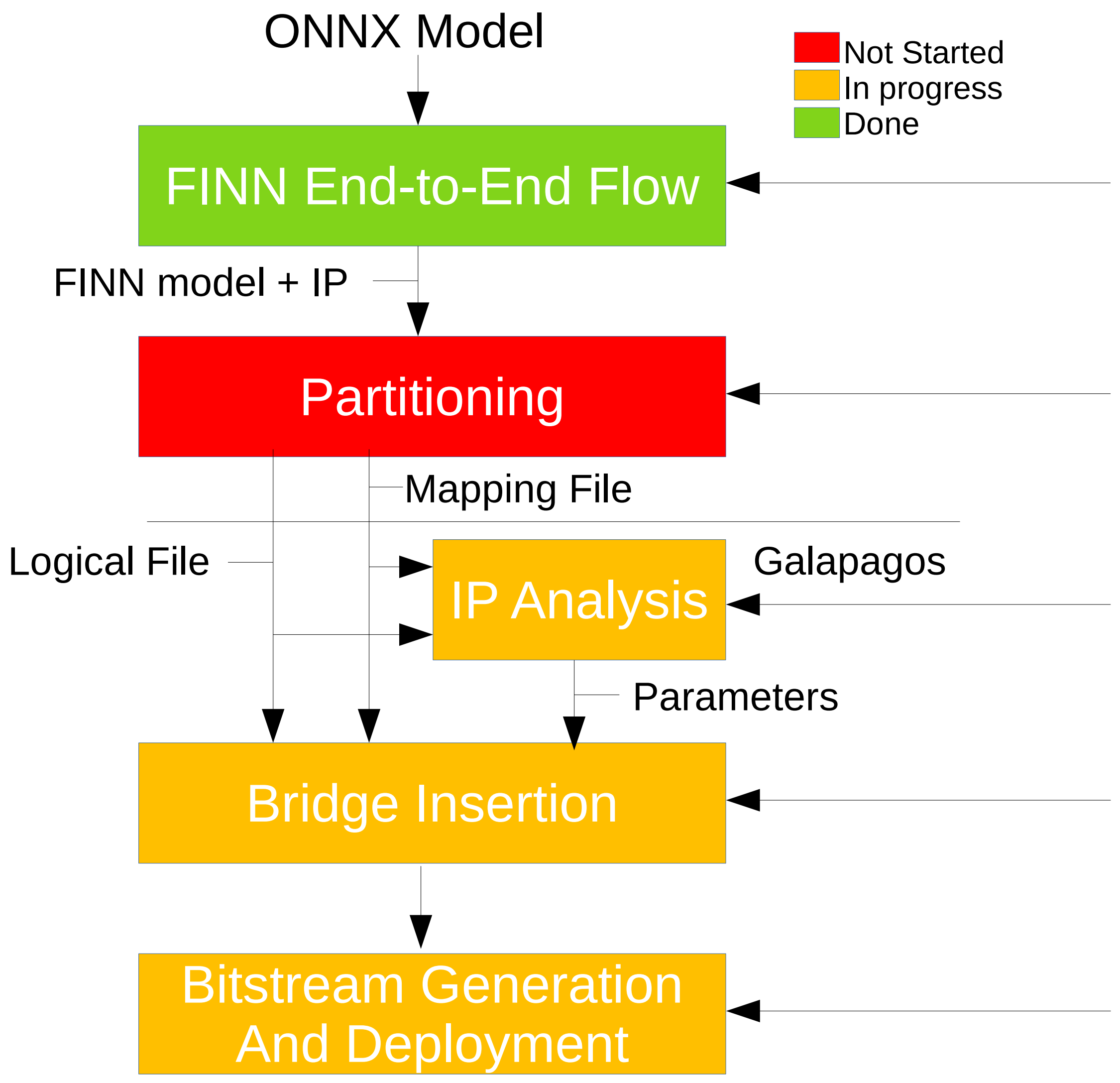
```

#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 256

            #define SIMD1 49
 #define PE1 16
 #define WMEM1 256

            #define TMEM1 16
 #define numReps 1
#define WP1 1


void 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, ap_uint<1> >
                (in0, out, weights, threshs, numReps, ap_resource_lut());
}
~
~
~
                                                                    1,0-1         All
```

- First stage: Connect and deploy IP cores using Galapagos
  - Easiest to start with
  - IP cores should (if the flow is used correctly) be generated correctly, saves us the trouble of debugging errors due to HLS
- Second/Long-term stage: Connect cores modelled by C++ files using Galapagos
  - Allows users to easily switch between running cores on CPU and HW
  - Would allow users to first test/debug models using software (libGalapagos)

Plan: Full Flow

1) Generate ONNX model and IP using FINN
2) Determine the layout and partitioning of IP cores across FPGAs, generating Galapagos Logical and Mapping File
3) Use Galapagos to stitch together IP and generate multiple bitstreams, connecting cores on different FPGAs together using Galapagos bridges

Full Flow Overview



User uses FINN End-to-End flow to transform and generate IP for the ONNX model

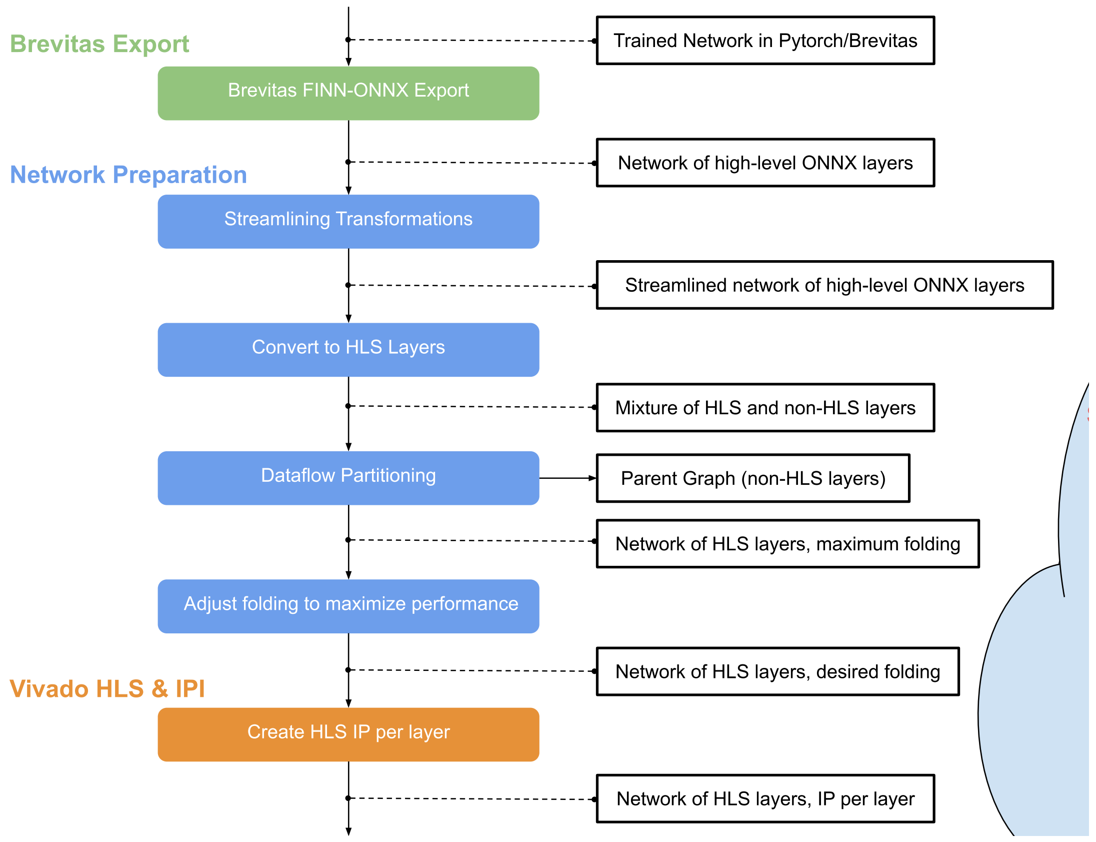
User decides how model will be partitioned across multiple devices, generating Logical file and Mapping file (could be automated)

Galapagos uses FINN model as well as Logical and Mapping file to figure out what bridges must be created, as well as their configuration

Galapagos stitches IP cores together and inserts Galapagos bridges, creating block diagrams for each device

Galapagos generates bitstreams and deploys them on each device

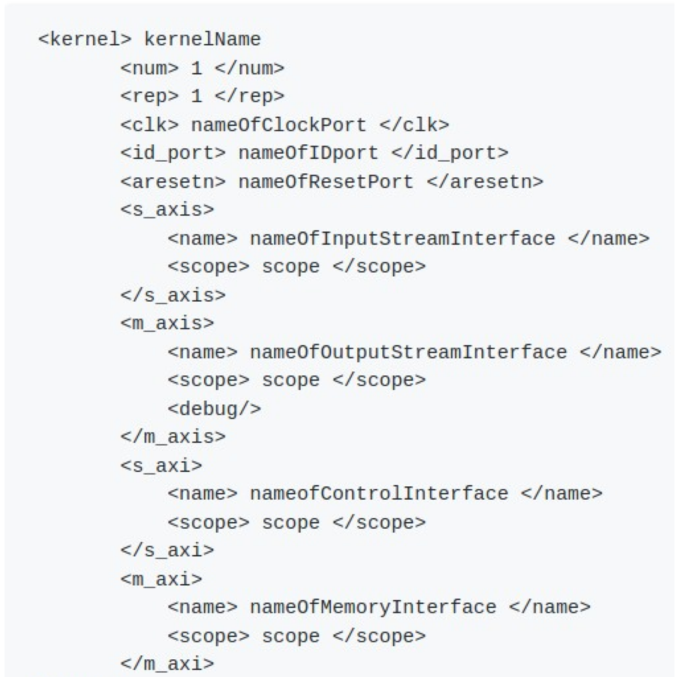
Full Flow P1: FINN Flow



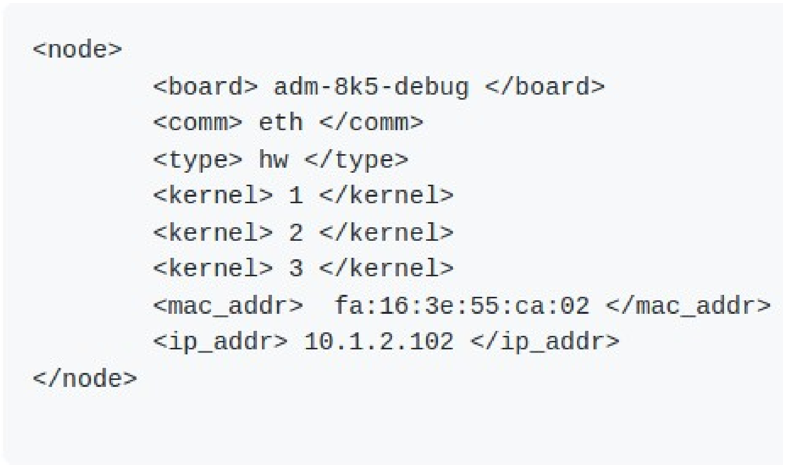
Brevitas Export
Trained Network in Pytorch/Brevitas
Brevitas FINN-ONNX Export
Network Preparation
Network of high-level ONNX layers
Streamlining Transformations
Streamlined network of high-level ONNX layers
Convert to HLS Layers
Mixture of HLS and non-HLS layers
Dataflow Partitioning
Parent Graph (non-HLS layers)
Network of HLS layers, maximum folding
Adjust folding to maximize performance
Vivado HLS & IPI
Network of HLS layers, desired folding
Create HLS IP per layer
Network of HLS layers, IP per layer

- User follows FINN's End-to-End flow, substituting the network that used in the tutorial for their network
- End to End Tutorials:
  - https://github.com/Xilinx/finn/tree/master/notebooks/end2end_example
  - Fully-Connected Networks and CNN examples provided
- User should follow the tutorial until IP has been created
- IMPORTANT: Towards the end of the flow the user will have to run the transform **PrepareIP(fpga_part, target_clk_ns).** Technically, fpga_part has to be one of the PYNQ parts that FINN officially supports. However, the generated IP also works with other boards and devices. Therefore, **put in any Xilinx part or device number**. If errors pop up saying that "/tmp/finn_dev_*" could not be found, use one of the approved parts. The IP should still be available in the IP catalog.
- STATUS:
  - FINN is maintained by Xilinx Research Labs, and is currently released in alpha phase

Full Flow P2: Partitioning



```
<kernel> kernelName
       <num> 1 </num>
       <rep> 1 </rep>
       <clk> nameOfClockPort </clk>
       <id_port> nameOfIDport </id_port>
       <aresetn> nameOfResetPort </aresetn>
       <s_axis>
           <name> nameOfInputStreamInterface </name>
           <scope> scope </scope>
       </s_axis>
       <m_axis>
           <name> nameOfOutputStreamInterface </name>
           <scope> scope </scope>
           <debug/>
       </m_axis>
       <s_axi>
           <name> nameofControlInterface </name>
           <scope> scope </scope>
       </s_axi>
       <m_axi>
           <name> nameOfMemoryInterface </name>
           <scope> scope </scope>
       </m_axi>
```

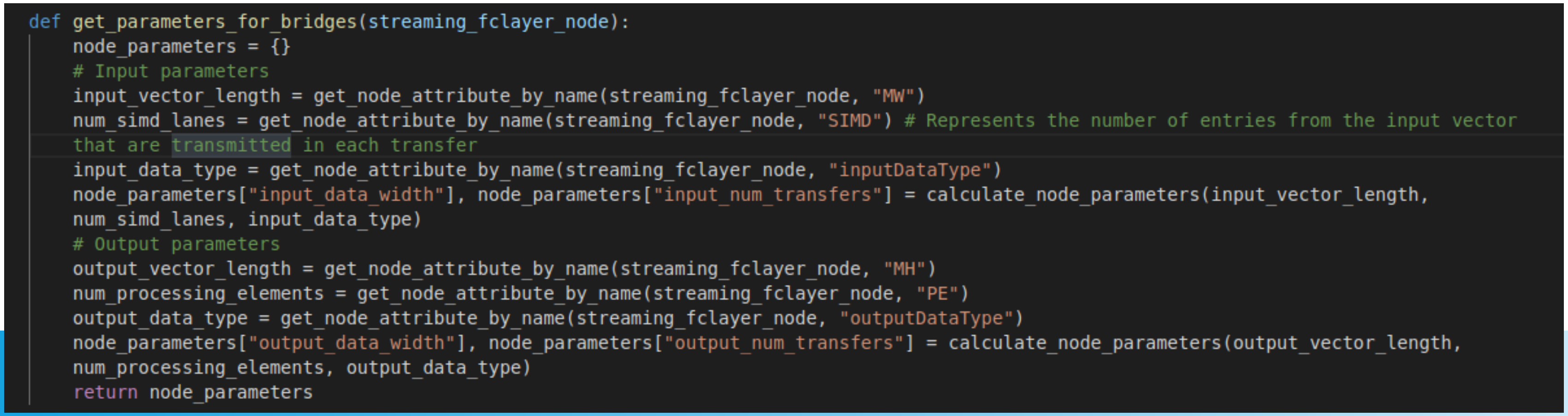


```
<node>
        <board> adm-8k5-debug </board>
        <comm> eth </comm>
        <type> hw </type>
        <kernel> 1 </kernel>
        <kernel> 2 </kernel>
        <kernel> 3 </kernel>
        <mac_addr>  fa:16:3e:55:ca:02 </mac_addr>
        <ip_addr> 10.1.2.102 </ip_addr>
</node>
```

- User uses their ONNX model to decide which cores should be partitioned onto each device
- Output of this stage should be a Galapagos Logical File and Mapping File
- Information about the Logical File and Mapping File can be found on the Galapagos repository: https://github.com/UofT-HPRC/galapagos
- STATUS:
  - Automation of this step has not been started

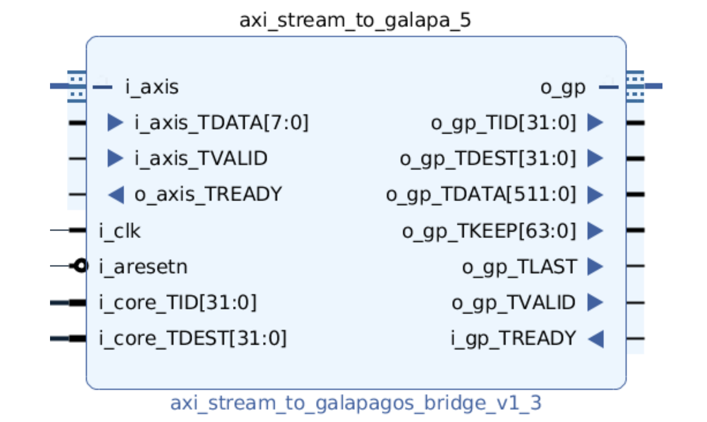
Full Flow P3: IP Analysis

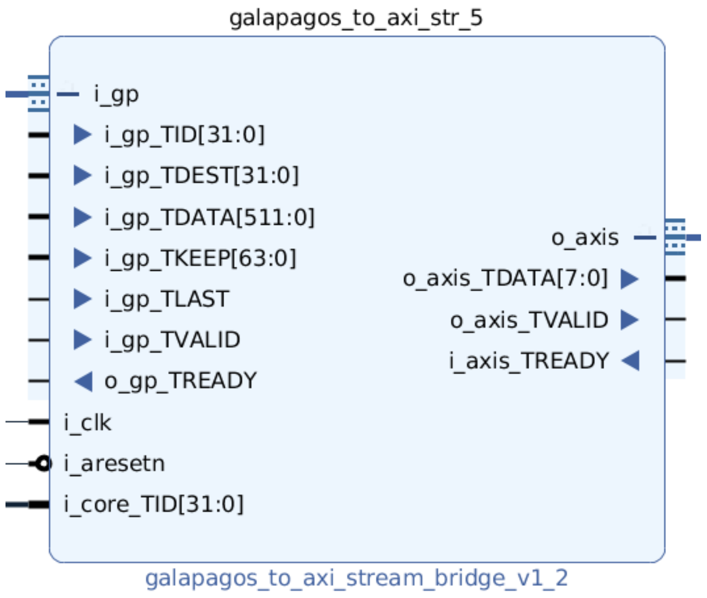
- Using python functions, Galapagos determines important parameters for Galapagos bridges
- IMPORTANT: These functions will look for IP stored in the “ipgen_path” attribute of each accelerated FINN node. If users copy IP to a different location, they **must** update these IP paths as well. The paths are absolute.
  - The script *finn_galapagos_software/finn_helper.py* contains functions that can be called inside the FINN docker container, one of which copies an ONNX model and all of its IP to a target location, changing all necessary filepaths automatically.
- Currently, the important parameters to gather for Galapagos are:
  - AXI-Stream input data width (in bits) (based on SIMD)
  - AXI-Stream output data width (in bits) (based on PE)
  - Number of input transfers
  - Number of output transfers
- STATUS:
  - Functions for gathering these parameters have been written in *finn_galapagos_software/finn_galapagos.py*, but a script needs to be created to automate their use. See the documentation in that folder for details.



```
def get_parameters_for_bridges(streaming_fclayer_node):
    node_parameters = {}
    # Input parameters
    input_vector_length = get_node_attribute_by_name(streaming_fclayer_node, "MW")
    num_simd_lanes = get_node_attribute_by_name(streaming_fclayer_node, "SIMD") # Represents the number of entries from the input vector
    that are transmitted in each transfer
    input_data_type = get_node_attribute_by_name(streaming_fclayer_node, "inputDataType")
    node_parameters["input_data_width"], node_parameters["input_num_transfers"] = calculate_node_parameters(input_vector_length,
    num_simd_lanes, input_data_type)
    # Output parameters
    output_vector_length = get_node_attribute_by_name(streaming_fclayer_node, "MH")
    num_processing_elements = get_node_attribute_by_name(streaming_fclayer_node, "PE")
    output_data_type = get_node_attribute_by_name(streaming_fclayer_node, "outputDataType")
    node_parameters["output_data_width"], node_parameters["output_num_transfers"] = calculate_node_parameters(output_vector_length,
    num_processing_elements, output_data_type)
    return node_parameters
```

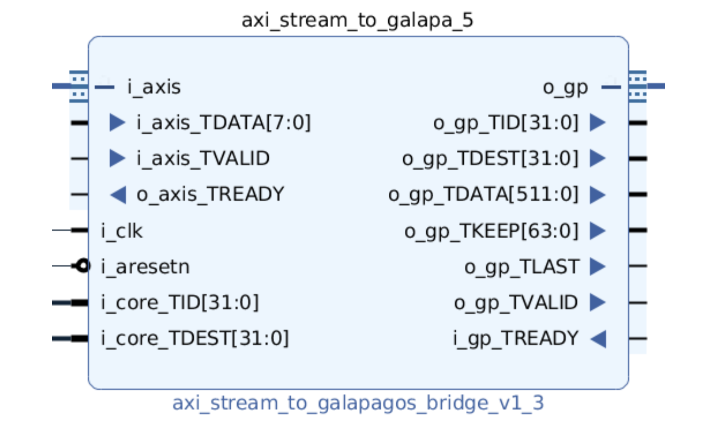
Full Flow P4: Bridge Insertion

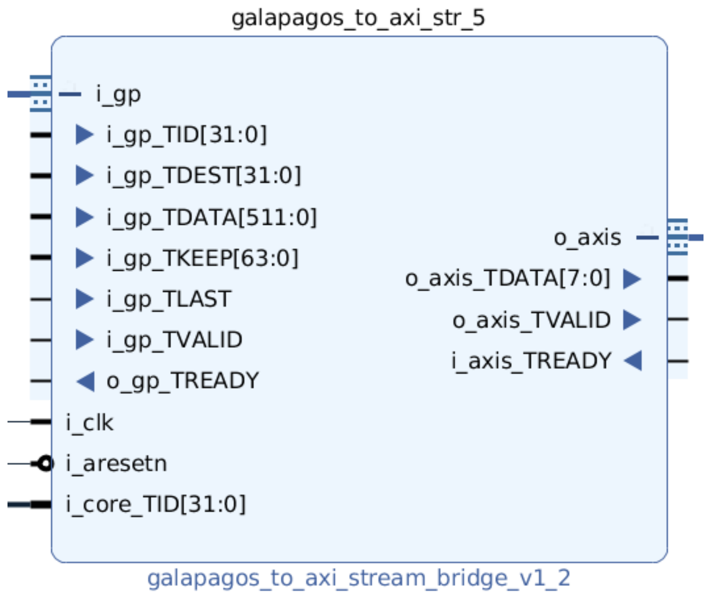




- Galapagos is used to convert the logical file and mapping file into stitched IP.
- Galapagos will use the logical file and mapping file to stitch IP cores together.
- Galapagos will also instantiate Galapagos bridges to connect IP cores on different FPGAs together, via a network switch.
- STATUS:
  - Bridges have been created to convert the AXI-Stream transfers from FINN cores into Galapagos packets, and vice versa.
  - These bridges will be attached between FINN cores and existing Galapagos network cores to enable connection between FPGAs.
  - Insertion of these bridges and generation of the stitched project have not been started

Full Flow P4: Bridges





- 2 Bridges: axi_stream_to_galapagos, galapagos_to_axi_stream
- Connect FINN cores with Galapagos network bridges
- User/Galapagos connects const cores to specify TID and TDEST

AXI-Stream to Galapagos Parameters:

- AXI_STREAM_DATA_WIDTH (bits)
- NUM_AXI_STREAM_TRANSFERS: number of transfers sent from the FINN core it's connected to that makes up a single output vector
- GALAPAGOS_DATA_WIDTH (bits)

Galapagos to AXI-Stream Parameters:

- AXI_STREAM_DATA_WIDTH (bits)
- GALAPAGOS_DATA_WIDTH (bits)
- NUM_GALAPAGOS_TRANSFERS: number of transfers sent from the AXIS_GP core it's connected to that makes up a single output vector (calculated by dividing FINN packet width by GALAPAGOS_DATA_WIDTH and rounding up)
- AXI_STREAM_DATA_WIDTH and NUM_TRANSFERS are calculated using IP Analysis

STATUS:

- Bridges have been simulated and tested on one FPGA, but should be tested on multiple FPGAs
- Additionally, the parameter GALAPAGOS_NUM_TRANSFERS should be replaced with AXI_STREAM_NUM_TRANSFERS for consistency

Full Flow P5: Deployment

- Galapagos generates bitstreams and deploys bitstreams onto FPGAs
- STATUS:
  - Galapagos currently builds Vivado projects, but bitstream deployment is not supported.
  - A full run needs to be tested, where the full flow is completed
  - Also, a process must be derived to connect the FINN cores deployed on FPGAs with unaccelerated FINN cores that must be run on CPU. Galapagos supports running software kernels on CPUs, so utilization of that should be investigated. Additionally, long-term goals of FINN are to increase the number of cores that get accelerated, so hopefully this option will not be necessary.

To do

These are tasks that still need to be completed for the Full Flow to function.

Full Flow P2: Partitioning:

- Automation of partitioning task

Full Flow P3: IP Analysis:

- Creation of IP Analysis script that uses built functions to automatically output bridge parameters based upon an ONNX model and the partitioning

Full Flow P4: Bridge Insertion:

- Test bridges with Galapagos hardware, and send packets across different FPGAs
- Modify bridges to not require GALAPAGOS_NUM_TRANSFERS parameter, for consistency
- Integrate the bridges into Galapagos so that Galapagos can insert the bridges automatically when it detects FINN cores

Full Flow P5: Deployment:

- Test of the full flow (from model generation to bitstream deployment)
- Automate deployment of bitstreams
- Connect unaccelerated cores of the FINN model to cores deployed on the FPGA.

Future Work

These are optional tasks for extending this flow.

- Support for C++ HLS kernels: Instead of taking FINN IP cores as an input, take in FINN C++ kernels which are transformed into FINN IP cores via HLS
  - This will allow FINN IP cores to be run on both software and hardware, and for simulations of the entire project to occur using libGalapagos
- Perform performance, power and area measurements, comparing deployment using FINN and deployment using Galapagos
- Extend these protocols to run other cores that communicate using AXI-Stream