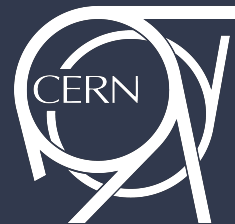


# Accelerating Deployment of FPGA-based AI in hls4ml with Parallel Synthesis through Model Partitioning

23rd International Workshop on Advanced Computing and Analysis Techniques in Physics Research

8–12 Sept 2025  
Hamburg, Germany



EP-SFT  
Software Frameworks and Tools



NextGen

Dimitrios Danopoulos, Vladimir Lončar

# Background on FPGAs

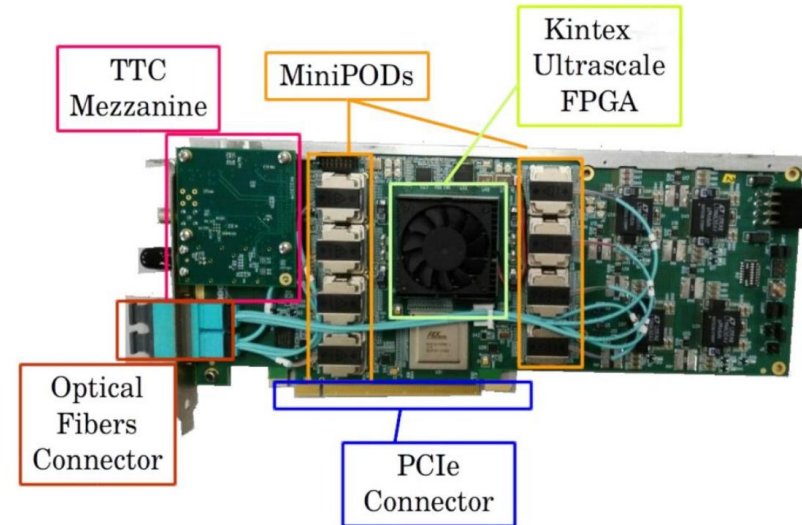
FPGAs are devices that can be re-configured after manufacturing allowing hw-level customization

They consist of

- configurable logic blocks (CLBs)
- Interconnects
- I/O blocks

They can achieve ultra-low latency in deterministic time

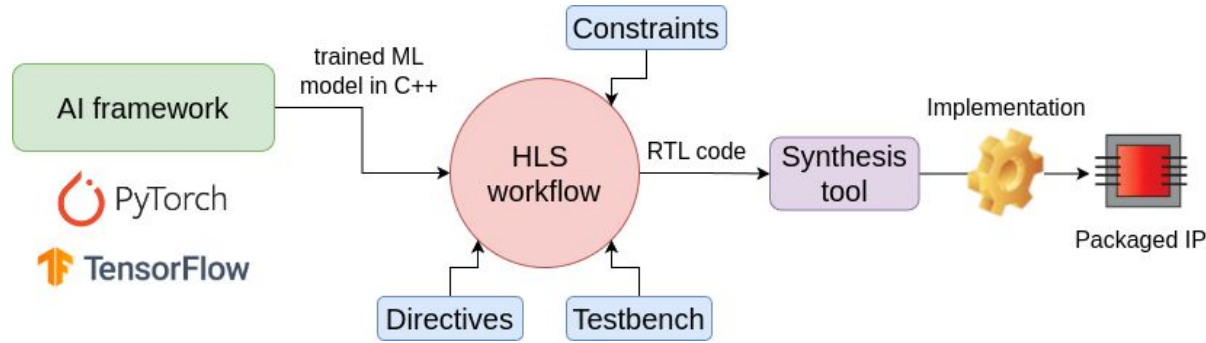
Due to their customizable architecture, they allow **ML-specific** configurations.



FPGA used in the ATLAS Detector [\[link\]](#)

# Designing ML models for FPGAs

Convert a trained ML model into a C++ representation for use in HLS workflow

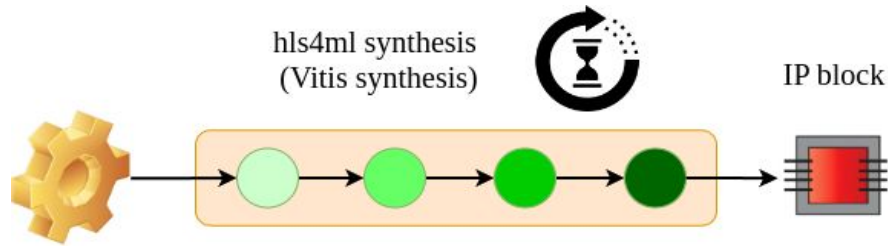


Using HLS to design ML models for FPGAs remains cumbersome and time-consuming:

1. HLS firmware design requires expertise
2. Achieving optimal performance is hard: a lot of parameters and strategies to consider
3. Debugging and verification are challenging due to the long synthesis times, even for small ML models

# Motivation and Background

**Problem:** The synthesis of complex ML models as a whole using hls4ml can be lengthy (often >24h)



Solution: Partition the model graph in hls4ml into smaller, independent subgraphs. Useful for:














- ❑ Large models
- ❑ Step-wise optimization
- ❑ Modular design flows

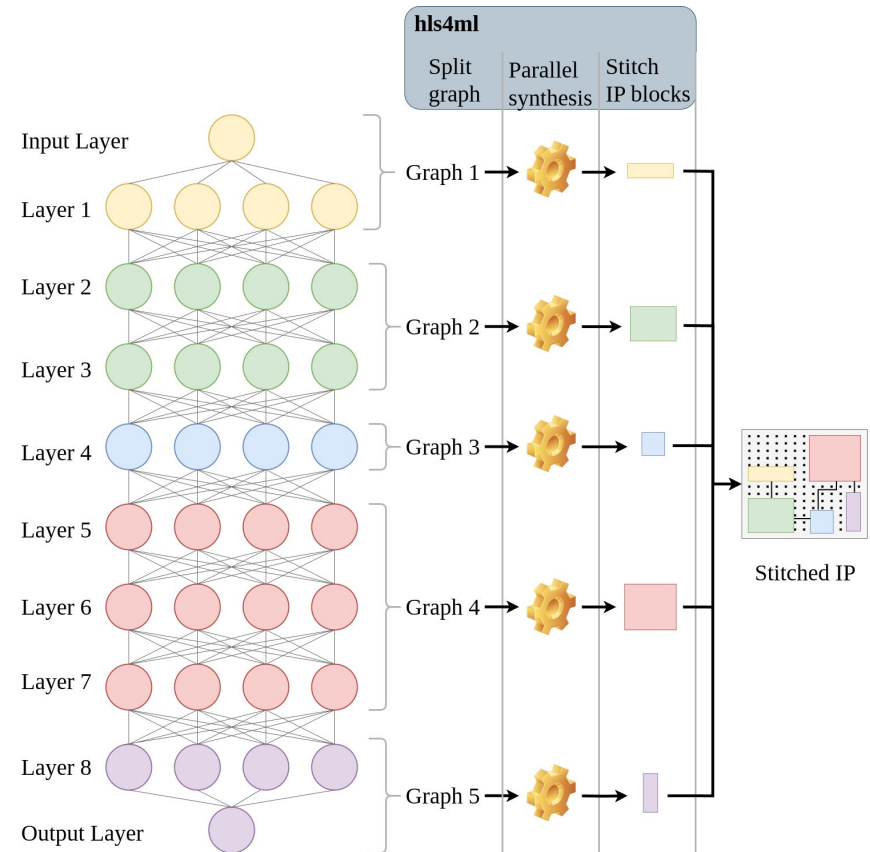
# Splitting the AI model graph for parallel synthesis

## Workflow:

- Select layers as splitting points
  - currently chosen by the user
  - future plans for being automated by hls4ml
- Parallel synthesis
  - Independently & concurrently synthesize each subgraph
  - Export each subgraph as an independent IP block.
- IP block stitching
  - automatic merging in Vivado to create the final IP

Synthesis  
status

graph1:		graph2:		graph3:	
graph1:		graph2:		graph3:	
graph1:		graph2:		graph3:	
graph1:		graph2:		graph3:	
graph1:		graph2:		graph3:	



# Splitting the Model Graph class in hls4ml



- User selects the layers to split the model at.
  - The `to_multi_model_graph` function facilitates splitting a base ModelGraph before the specified layer names.
- 

Top-level interface with the user

```
config = hls4ml.utils.config_from_keras_model(model, granularity='model')

hls_model = hls4ml.converters.convert_from_keras_model(
    model,
    hls_config=config,
    backend='vitis',
)

hls_multigraph_model = hls4ml.model.to_multi_model_graph(hls_model, ['layer3', 'layer7'])
```

What `to_multi_model_graph` does:

1. Ensures split layers exist and are valid (i.e., no merge layers).
2. Adjusts subgraph configurations (i.e., adding 'graphN' suffix to OutputDir and ProjectName.)
3. Passes precision information from the last layer of one subgraph to the next.
4. Creates ModelGraph objects for each subgraph.
5. Encapsulates the subgraphs into the new MultiModelGraph class.

# The `MultiModelGraph` class

We use a combination of composition and delegation

Composition: The **`MultiModelGraph`** class is composed of many instances of the `ModelGraph` class (subgraphs)






Delegation: We provide re-implemented the common methods of `ModelGraph` class (i.e., `build`, `compile`, `predict`, etc.)

Every subgraph (`ModelGraph` object) can always be accessed via indexing: `hls_model[i]`

➡ This allows us to interact with a specific graph and invoke methods of the `ModelGraph` if needed.

The build method re-implemented for MultiModelGraph

- Invokes Vitis for every subgraph build
- Tracks every graph's build status in real-time

graph1:			graph2:			graph3:	
graph1:			graph2:			graph3:	
graph1:			graph2:			graph3:	
graph1:			graph2:			graph3:	
graph1:			graph2:			graph3:	

Synthesis status monitoring

## Pseudocode of MultiModelGraph.build

```
1. func build(..., stitch_design=False, sim_stitched_design=False
2.     export_stitched_design=False, max_workers=None):
3.
4.     Initialize status for all graphs
5.
6.     func build_wrapper(graph):
7.         Update status to "Running"
8.         Try:
9.             Build the graph
10.            Update status to "Completed"
11.            Return build result
12.        On failure:
13.            Update status to "Failed"
14.            Raise exception
15.
16.    Run build_wrapper in parallel for all graphs
17.    Collect build reports into build_results
18.
19.    If stitching is enabled:
20.        Parse NN input/output layers
21.        Build stitched design
22.        Return stitched design report
23.
24.    Return build_results
```



After exporting each individual IP of every subgraph:

- We need to stitch them into a single cohesive IP
- Be able to simulate the stitched design if needed
- Export the final stitched IP

## **For stitching and exporting:**

- We have developed a stitcher script using Vivado TCL
- It automatically creates the necessary IP interconnections.
- Exposes input/output ports and exports the complete IP.
- Logs the entire stitching process in a verbose log file (for debugging).

## **For simulating stitched IP:**

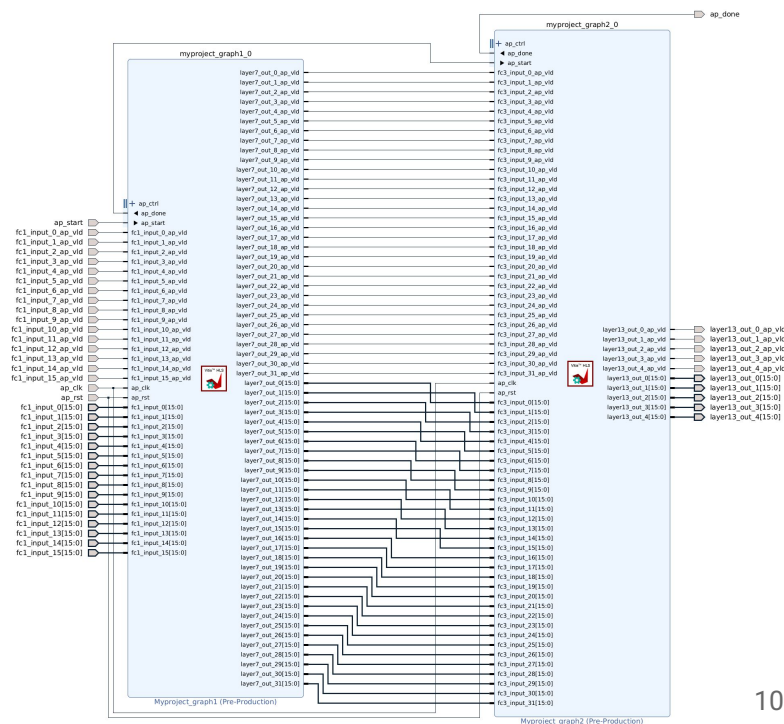
- Implemented simulation support using Xilinx XSIM.
- Dynamically generate a Verilog testbench based on the size of input/output layers and provided data.
- Input data are written into a file and read directly by the testbench.
- Runs simulation to validate the IP's functionality and latency.
- Logs IP output + latency to a file which is then processed by hls4ml

## MLP example 1 (partition pragma)

- Support for partitioned and streaming IP interfaces
- ~4x decrease in synthesis time based on current setup
- Greater reductions can be expected for larger models.

*Theoretically, the decrease ratio can match the No. of subgraphs*

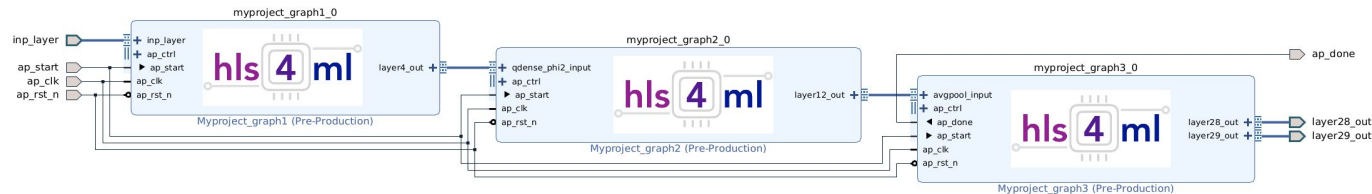
Model	DSP	FF	LUT	Latency
Original	2870	23719	122284	15 cycles
Graph 1	2071	16966	89807	5 cycles
Graph 2	799	6799	34177	9 cycles
Stitched	2870	23765	123984	14 cycles



- Split before & after the branch to keep all branches inside the graph

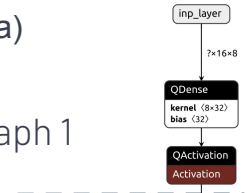
(Splitting in the middle of branches not yet supported)

Model	DSP	FF	LUT	Latency
Original	129	29083	79675	20 cycles
Graph 1	47	4039	12984	21 cycles
Graph 2	0	10906	42059	28 cycles
Graph 3	82	12235	23768	22 cycles
Stitched	129	<b>27180</b>	<b>78807</b>	19 cycles



MLP example 2  
(stream pragma)

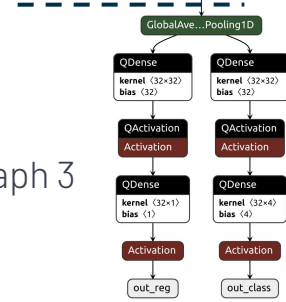
Graph 1



Graph 2



Graph 3

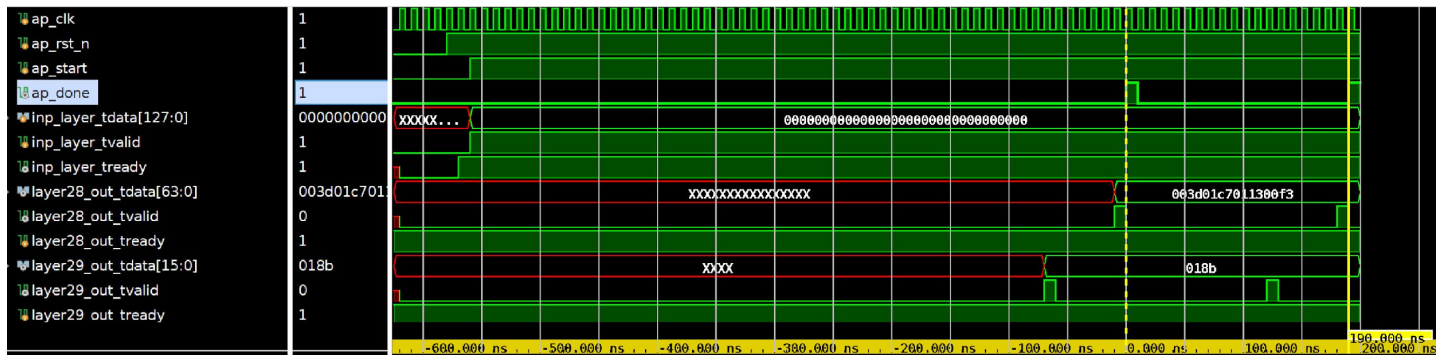


A Verilog testbench is generated automatically based on user-defined input.

Cold Start: In the first data sample pipeline stages are not fully filled, leading to inflated latency. We usually ignore it as real-world use has continuous streaming.

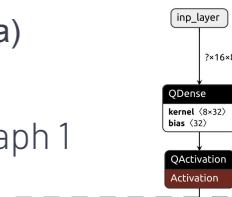
Steady-State Latency: The second data sample reflects the steady-state operation of the IP after the pipeline is active. We define latency as time between ap\_done signals.

Testbench logs the output of IP and latency (both cold start and steady-state)

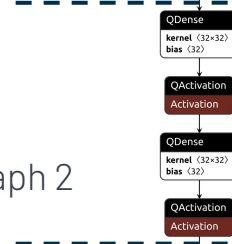


MLP example 2  
(stream pragma)

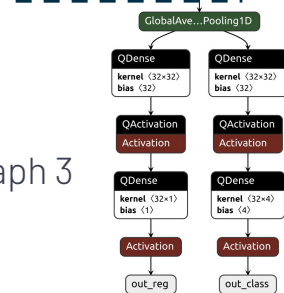
Graph 1



Graph 2



Graph 3



## ResNet20 (stream pragma)

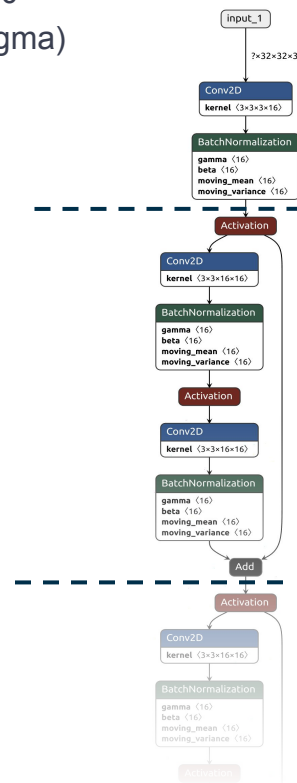
Handling larger models is easier now. Improved debugging and isolation of layers

- Model Tested: ResNet20      Dataset: Cifar10
- HLS config: 5 graphs / 4bit weights / 4 ReuseFactor

Monolithic synthesis time: **~10h**

Multigraph synthesis + stitch time: **~3h**

Model	DSP	BRAM	FF	LUT	Latency
Monolithic	10 (0%)	950 (17%)	196259 (5%)	509752 (29%)	4715 cycles
Stitched	10 (0%)	834 (16%)	187716 (5%)	501938 (29%)	4625 cycles



# The MultiModelGraph class



## Monolithic ResNet20 report

```
{'CSynthesisReport': {'TargetClockPeriod': '5.00',  
  'EstimatedClockPeriod': '3.599',  
  'BestLatency': '4715',  
  'WorstLatency': '4715',  
  'IntervalMin': '4627',  
  'IntervalMax': '4627',  
  'BRAM_18K': '950',  
  'DSP': '10',  
  'FF': '196259',  
  'LUT': '509752',  
  'URAM': '0',  
  'AvailableBRAM_18K': '5376',  
  'AvailableDSP': '12288',  
  'AvailableFF': '3456000',  
  'AvailableLUT': '1728000',  
  'AvailableURAM': '1280'}}
```

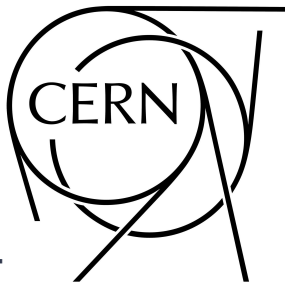
## Stitched ResNet20 report

```
{'StitchedDesignReport': {'TargetClockPeriod': '5.00',  
  'EstimatedClockPeriod': '3.599',  
  'BestLatency': 4625,  
  'WorstLatency': 7624,  
  'BRAM_18K': '834',  
  'DSP': '10',  
  'FF': '187716',  
  'LUT': '501938',  
  'URAM': '0',  
  'AvailableBRAM_18K': '5376',  
  'AvailableDSP': '12288',  
  'AvailableFF': '3456000',  
  'AvailableLUT': '1728000',  
  'AvailableURAM': '1280'}}
```

\* All the previously reported results are obtained from CSynthesis at this stage. The purpose was to demonstrate the graph partitioning algorithm.

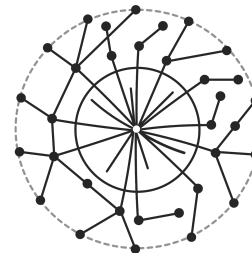
## Benefits at a glance:

- Time efficiency: Significantly reduces synthesis and build times.
- Scalability: Enables efficient handling of larger models by partitioning.
- Flexibility: Simplifies debugging of subgraphs without requiring full model resynthesis.
- Latency reduction: Potential improvements observed in some models during synthesis.
- AI framework-agnostic: the partitioning is handled internally by hls4ml.



EP-SFT

Software Frameworks and Tools



**NextGen**

# Thank you!

This work has been [partially] funded by the Eric & Wendy Schmidt Fund for Strategic Innovation through the CERN Next Generation Triggers project under grant agreement number SIF-2023-004.