hls4ml demo

- **Today we will go through a few notebooks from the hls4ml tutorial**
  - more info at the hls4ml documentation
- **Part 1:** get started with hls4ml and train a basic model and run the conversion, simulation & c-synthesis steps

  `notebook: part1_getting_started.ipynb`

- **Part 2:** learn how to tune inference performance with post-training quantization and parallelization

  `notebook: part2_advanced_config.ipynb`

- **Part 3:** perform model compression and observe its effect on the FPGA resources/latency

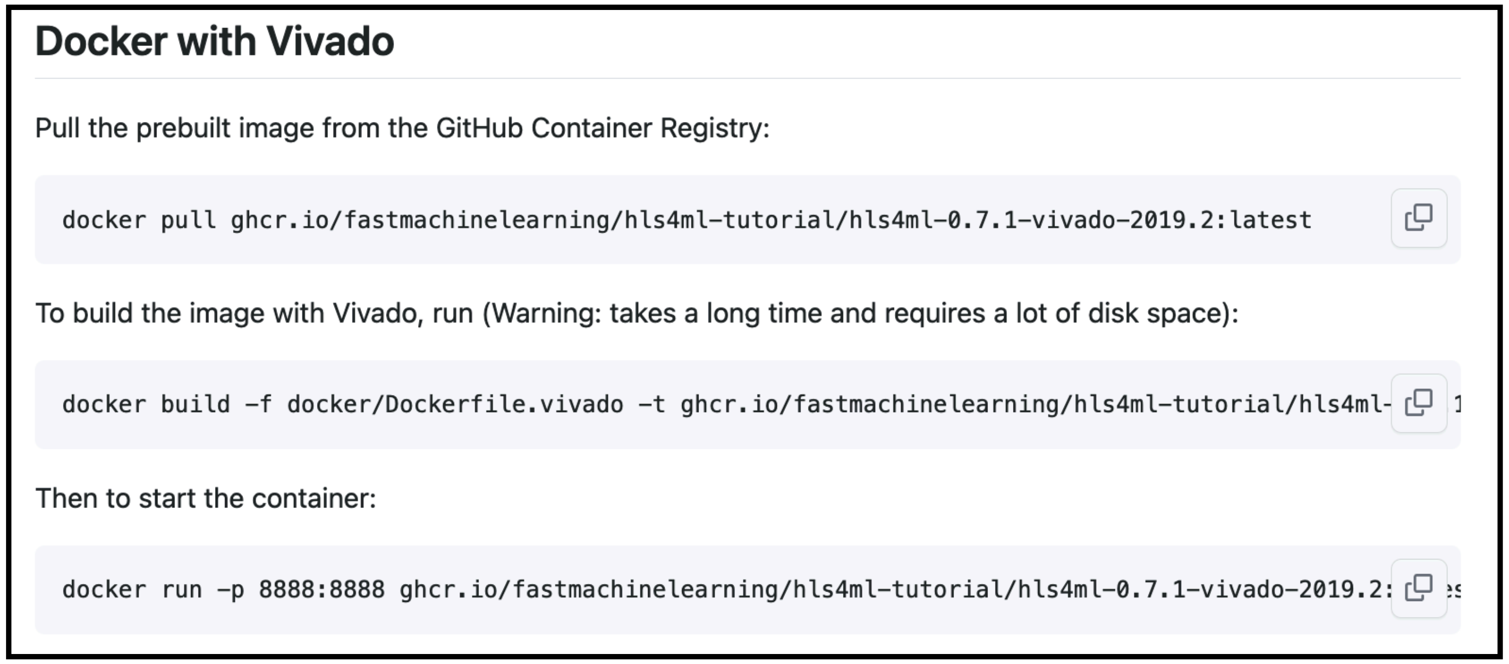
  `notebook: part3_compression.ipynb`

- **Part 4:** train using QKeras "quantization-aware training" and study impact on FPGA metrics

  `notebook: part4_quantization.ipynb`

Caveats

- hls4ml needs FPGA vendor HLS compiler tools
- In the past we were used to have on demand custom colab servers with Xilinx Vivado HLS installed for education purposes but now not anymore :(
- We do though provide prebuilt docker images with Vivado that can be pulled and built (see README)
- However, it takes long time and a lot of disk space to build the image
- As there's not enough time today, I'll walk you through the notebooks and teach you some HLS basics which can hopefully be a starting point for you to go deeper by yourself in the next days of the school or in the future



Docker with Vivado

Pull the prebuilt image from the GitHub Container Registry:

```
docker pull ghcr.io/fastmachinelearning/hls4ml-tutorial/hls4ml-0.7.1-vivado-2019.2:latest
```

To build the image with Vivado, run (Warning: takes a long time and requires a lot of disk space):

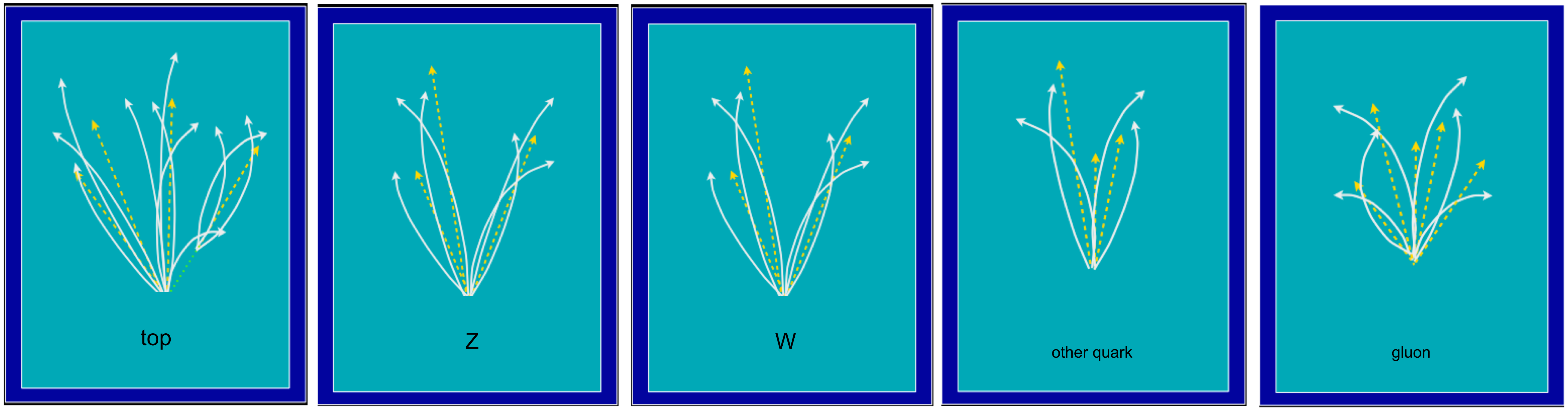
```
docker build -f docker/Dockerfile.vivado -t ghcr.io/fastmachinelearning/hls4ml-tutorial/hls4ml-
```

Then to start the container:

```
docker run -p 8888:8888 ghcr.io/fastmachinelearning/hls4ml-tutorial/hls4ml-0.7.1-vivado-2019.2:
```

Physics case: jet tagging

Study a **multi-classification task to be implemented on FPGA:** discrimination between highly energetic (boosted) ***q, g, W, Z, t*** initiated *jets*



$t \to bW \to bqq$ — 3-prong jet

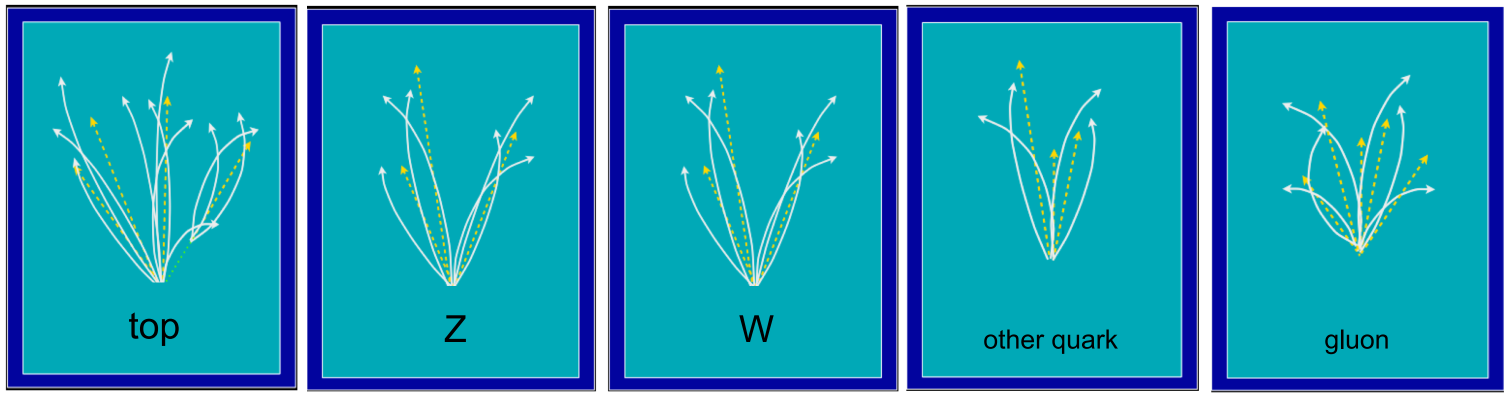
$Z \to qq$ — 2-prong jet

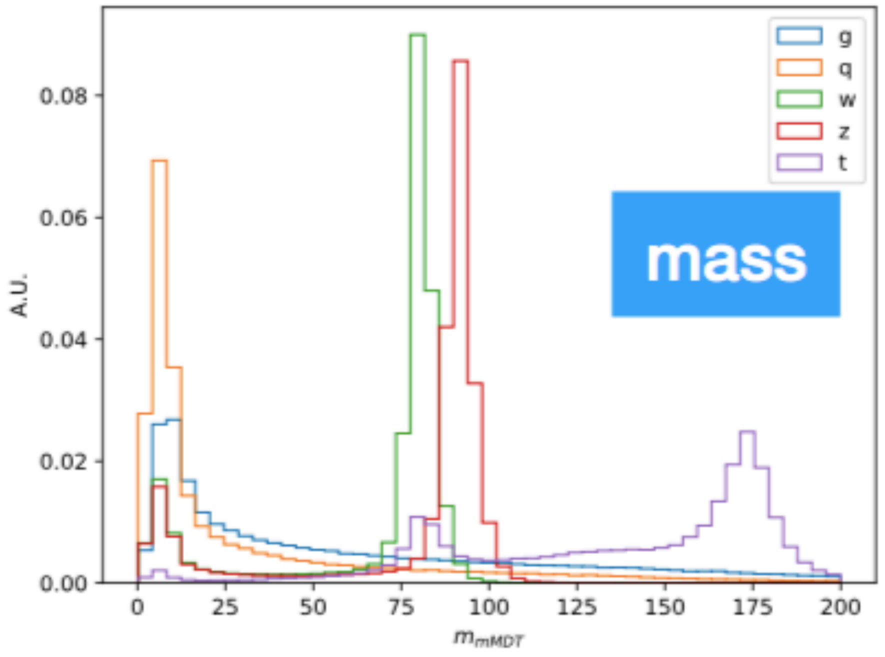
$W \to qq$ — 2-prong jet

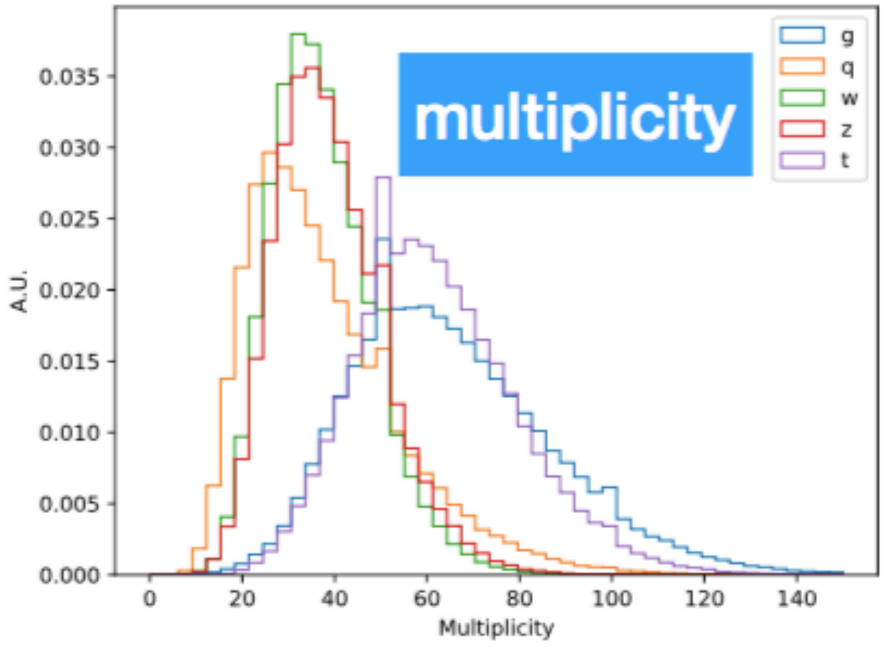
Reconstructed as one massive jet with substructure

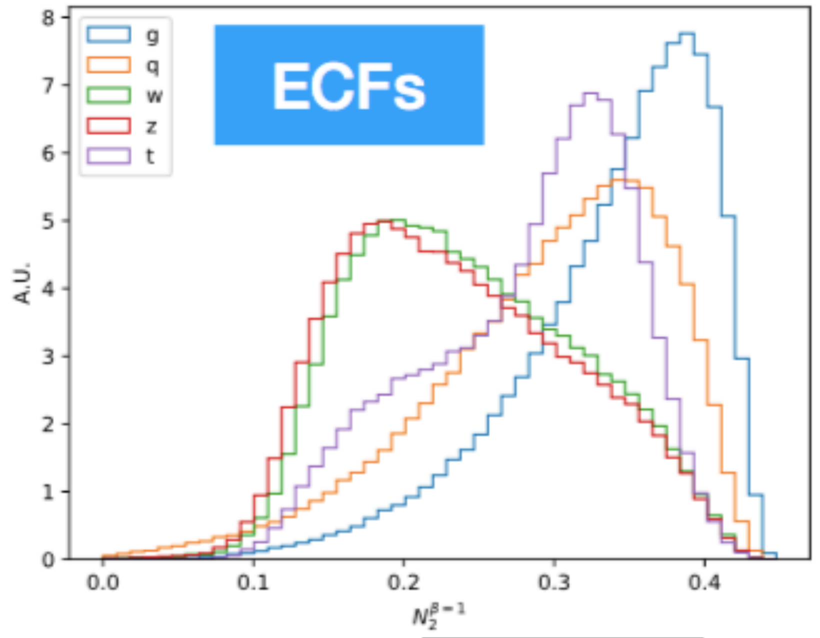
q/g background — no substructure and/or mass ~ 0

Physics case: jet tagging









Input variables: several observables known to have high discrimination power from offline data analyses and published studies [*]

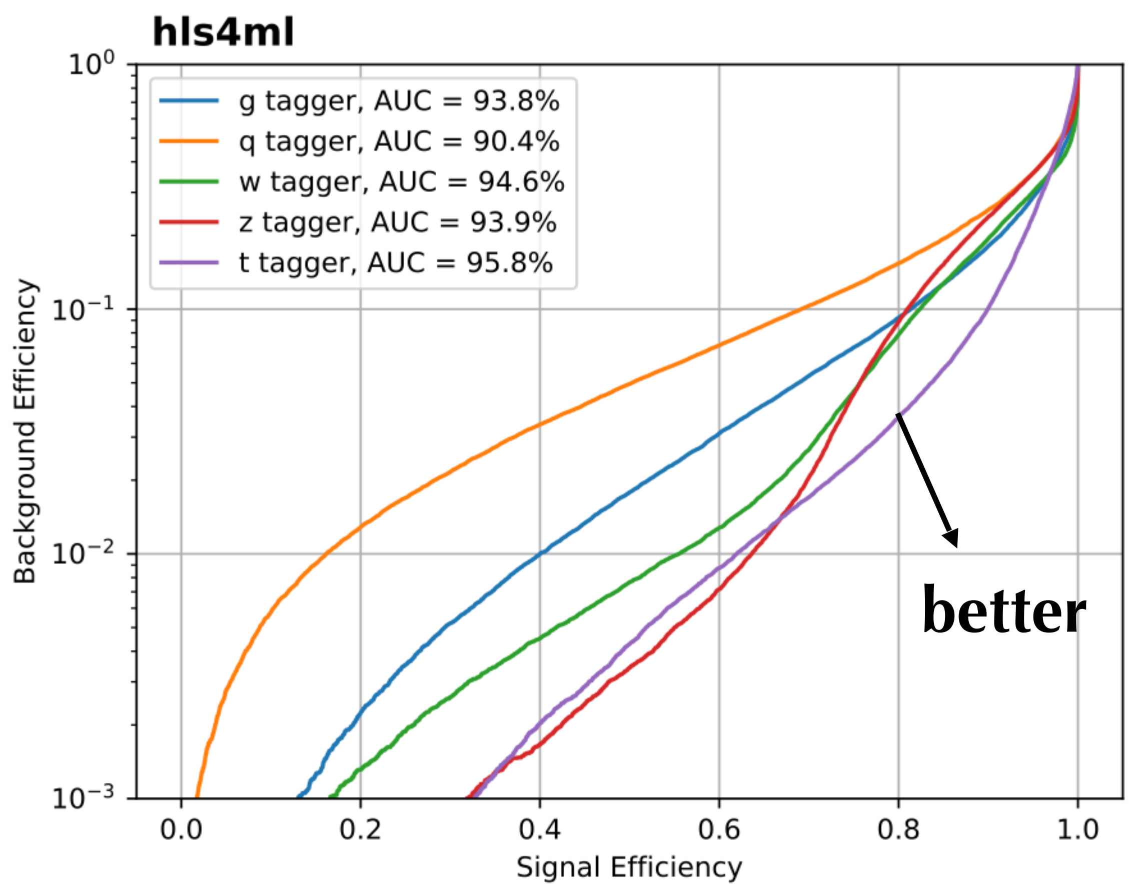
m_{mMDT}
$N_2^{\beta=1,2}$
$M_2^{\beta=1,2}$
$C_1^{\beta=0,1,2}$
$C_2^{\beta=1,2}$
$D_2^{\beta=1,2}$
$D_2^{(\alpha,\beta)=(1,1),(1,2)}$
$\sum z \log z$
Multiplicity

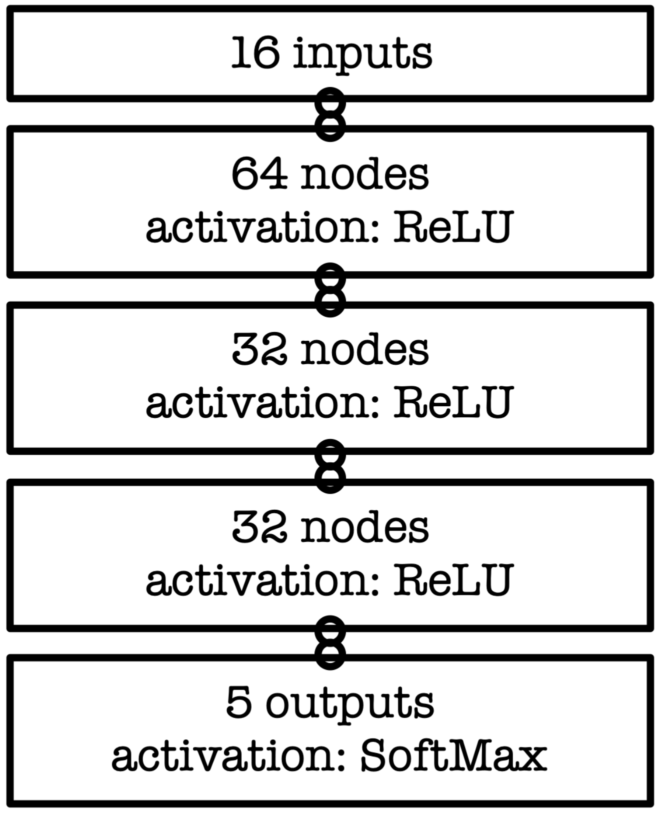
[*] D. Guest at al. PhysRevD.94.112002, G. Kasieczka et al. JHEP05(2017)006, J. M. Butterworth et al. PhysRevLett.100.242001, etc..

Physics case: jet tagging

- We'll train the five class multi-classifier on a sample of ~1M events with two boosted WW/ZZ/tt/qq/gg anti-k_T jets [doi:10.5281/zenodo.3602254, OpenML]

- Fully connected neural network with 16 expert-level inputs:
  - Relu activation function for intermediate layers
  - Softmax activation function for output layer





AUC = area under ROC curve (100% is perfect, 50% is random)

Introduction to HLS

- **High-Level Synthesis (HLS)** creates firmware blocks from C++
  - FPGA (concurrent) programming requires taking care of a temporal component
  - this is controlled in HLS through **pre-processor directives** (pragmas)
- We use **Vivado HLS** which is the **Xilinx** version
  - there are actually lots of different HLS's, but they basically all do the same thing
  - for example: Intel Quartus HLS
- **References:**
  - https://docs.xilinx.com/v/u/en-US/ug902-vivado-high-level-synthesis
  - https://raw.githubusercontent.com/KastnerRG/pp4fpgas/gh-pages/main.pdf

HLS project overview

C++ test bench [2]
Test your HLS block on inputs, check outputs, etc…

HLS block [1]
Your task, written in C++, with directives to guide HLS to optimize the firmware for your task

[1] hls4ml_prj/firmware/myproject.cpp
[2] hls4ml_prj/myproject_test.cpp

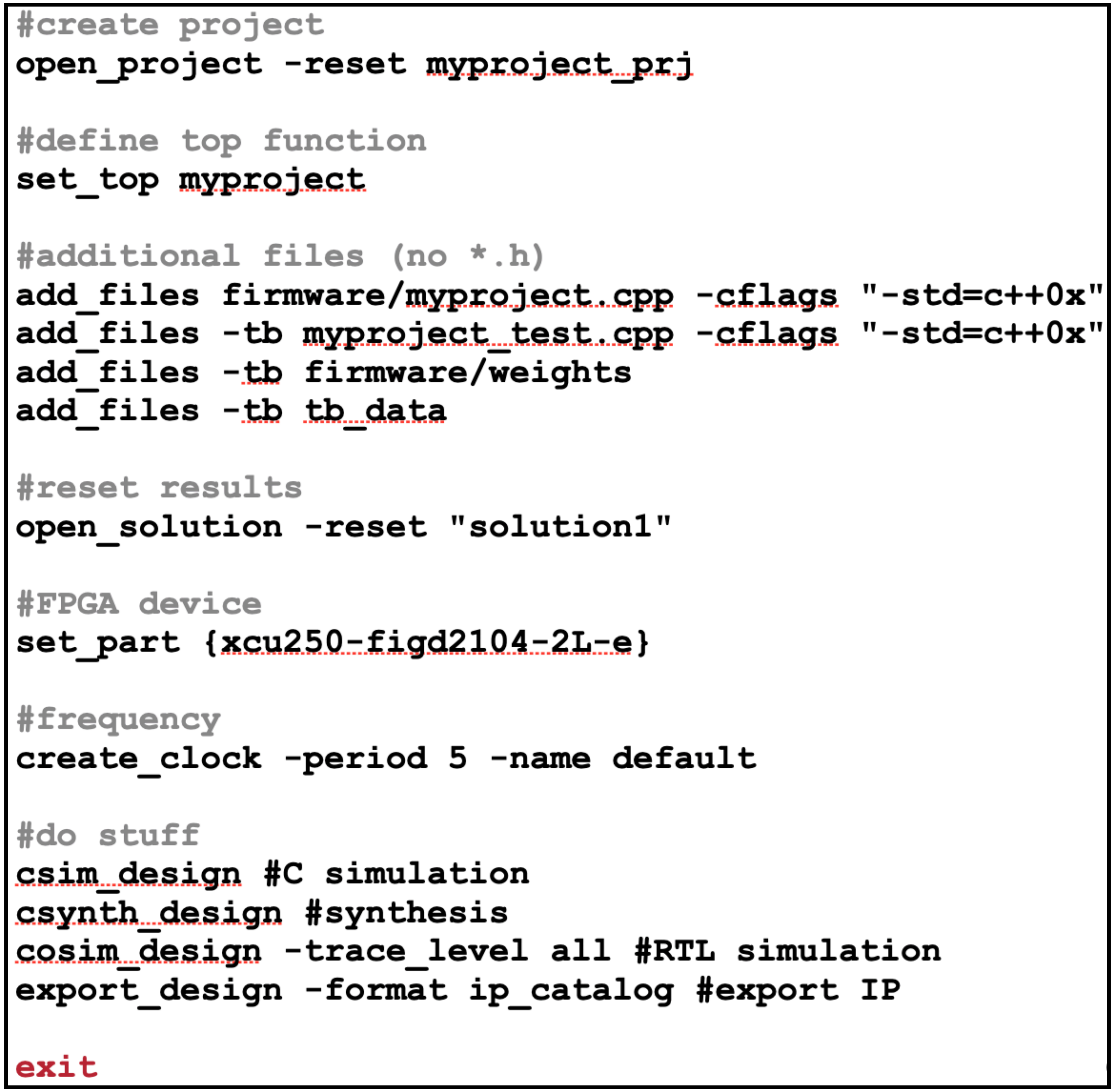
nb, in hls4ml API the **predict** method runs the test bench

Building the project: the TCL file

We build our project using the **TCL scripting language**

In hls4ml this file is created with **hls_model.build()** using parameters from configuration specified at hls_model creation time

Steps for building project →



```
#create project
open_project -reset myproject_prj

#define top function
set_top myproject

#additional files (no *.h)
add_files firmware/myproject.cpp -cflags "-std=c++0x"
add_files -tb myproject_test.cpp -cflags "-std=c++0x"
add_files -tb firmware/weights
add_files -tb tb_data

#reset results
open_solution -reset "solution1"

#FPGA device
set_part {xcu250-figd2104-2L-e}

#frequency
create_clock -period 5 -name default

#do stuff
csim_design #C simulation
csynth_design #synthesis
cosim_design -trace_level all #RTL simulation
export_design -format ip_catalog #export IP

exit
```

ex: hls4ml_prj/build_prj.tcl

Building the project: the TCL file

```
#do stuff
csim_design #C simulation
csynth_design #synthesis
cosim_design -trace_level all #RTL simulation
export_design -format ip_catalog #export IP
```

- **csim_design**:
  - C simulation of test bench and HLS block
  - both compiles and runs your code at the C++ level to make sure it's working
- **csynth_design**:
  - synthesizes your project for the FPGA and makes an estimate of resource usage and timing
  - generates RTL-level design in Verilog/VHDL to be synthesized (also called *logic synthesis*)
- **cosim_design**: a simulation of the RTL design to verify its functionality
- **export_design**: exports project with well-defined interfaces enabling it to be incorporated into a larger design
  - also called *IP block* (with IP = Intellectual Property)

Building the project: the TCL file

Additional step before deploying the design on the FPGA:

- **logic synthesis**: it is run with Vivado (not Vivado HLS) — different tcl script [*]
  - it is translates the RTL design into a netlist, i.e. the list of logical elements and the connections between them
  - the netlist is then associated with specific resources in a target device (*place and route*)
  - resulting configuration captured in a bitstream containing a a binary representation of the configuration of each FPGA resource including a logic elements, wire connections, and on-chip memories
  - over 1 billion configuration bits on modern FPGAs!
  - command: `vivado -mode batch -source design.tcl`

[*] https://docs.xilinx.com/viewer/book-attachment/KslPHBCdt0FBb__VT~0IFg/wAZ0CgY5heu3u6xaiteGeA

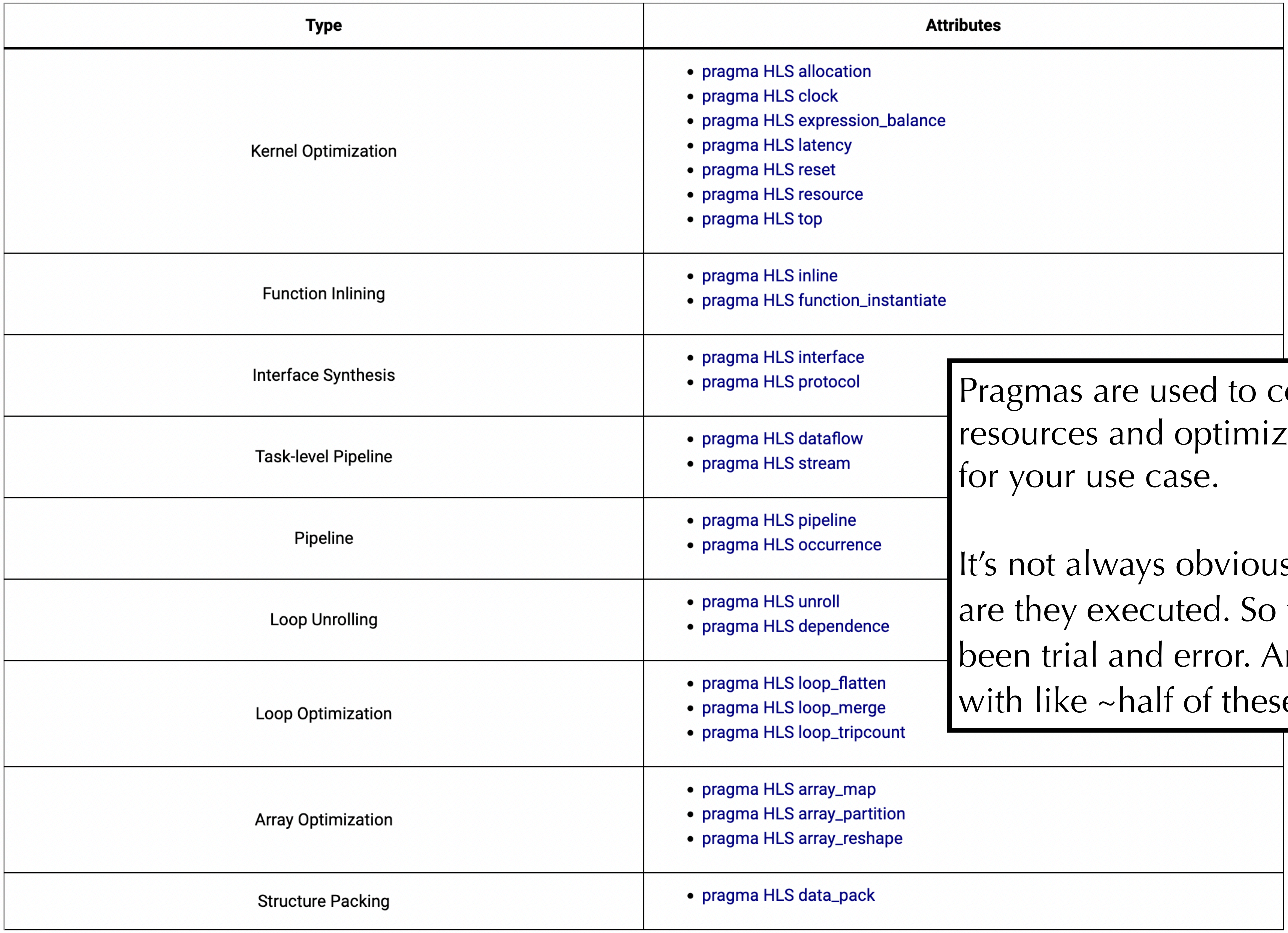
Running

```
vivado_hls -f <mytcl>.tcl
```

In hls4ml this is run automatically by the **hls_model.build()**
you can specify some options, ex: hls_model.build(**csim=False**)

Pragmas

https://www.xilinx.com/htmldocs/xilinx2017_2/sdaccel_doc/topics/pragmas/ref-pragma_HLS_resource.html



Type	Attributes
Kernel Optimization	• pragma HLS allocation • pragma HLS clock • pragma HLS expression_balance • pragma HLS latency • pragma HLS reset • pragma HLS resource • pragma HLS top
Function Inlining	• pragma HLS inline • pragma HLS function_instantiate
Interface Synthesis	• pragma HLS interface • pragma HLS protocol
Task-level Pipeline	• pragma HLS dataflow • pragma HLS stream
Pipeline	• pragma HLS pipeline • pragma HLS occurrence
Loop Unrolling	• pragma HLS unroll • pragma HLS dependence
Loop Optimization	• pragma HLS loop_flatten • pragma HLS loop_merge • pragma HLS loop_tripcount
Array Optimization	• pragma HLS array_map • pragma HLS array_partition • pragma HLS array_reshape
Structure Packing	• pragma HLS data_pack

Pragmas are used to control the timing/resources and optimize the HLS block for your use case.

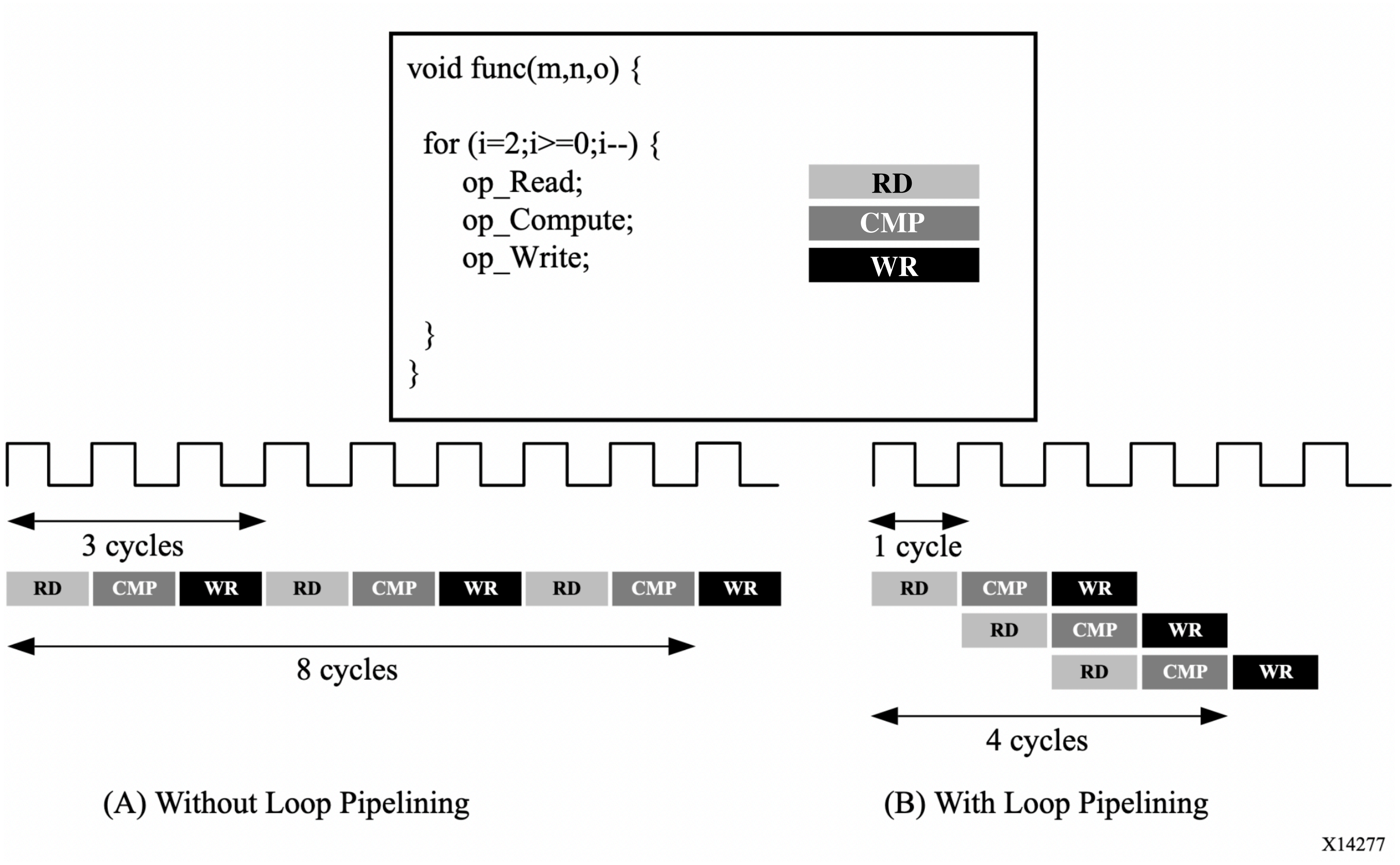
It's not always obvious in what priority are they executed. So far it's mostly been trial and error. And I'm unfamiliar with like ~half of these..

Pipelining

What is the **Initiation Interval (II)**:

- for a function, II is the number of clock cycles before it could accept new inputs
- for a loop, II is the number of clock cycles before the next iteration of a loop starts to process data

In hls4ml is what we call the **Reuse Factor**



II = 3

Latency = 8

II = 1

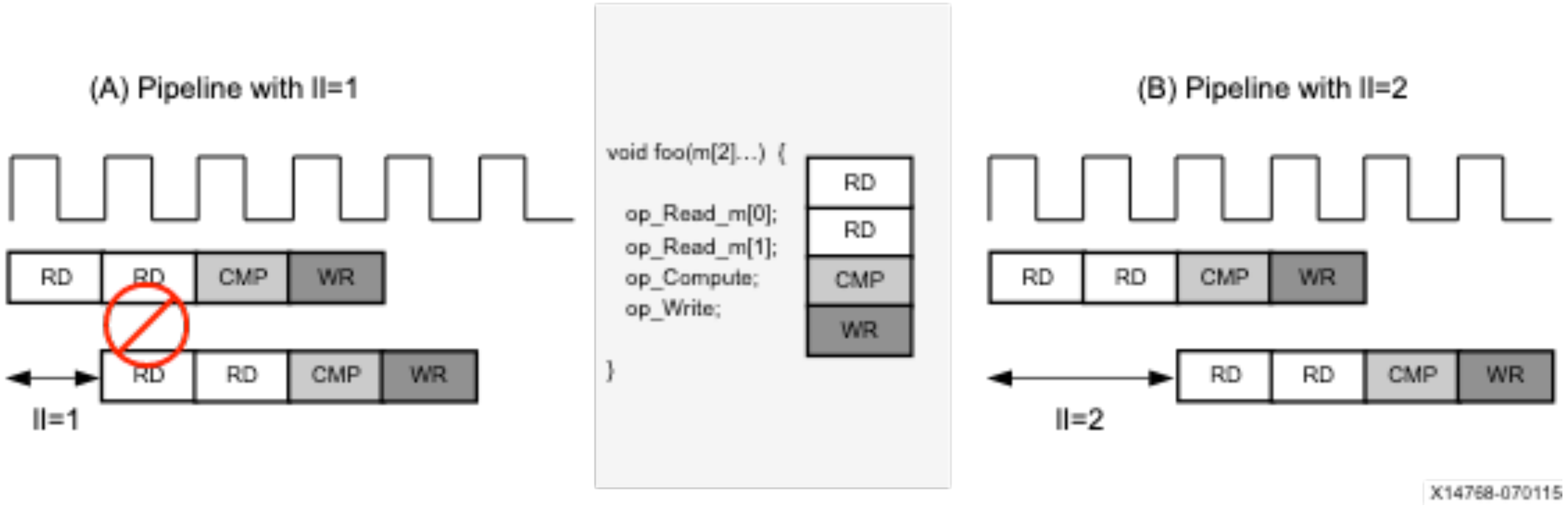
Latency = 4

(A) Without Loop Pipelining

(B) With Loop Pipelining

Partitioning arrays

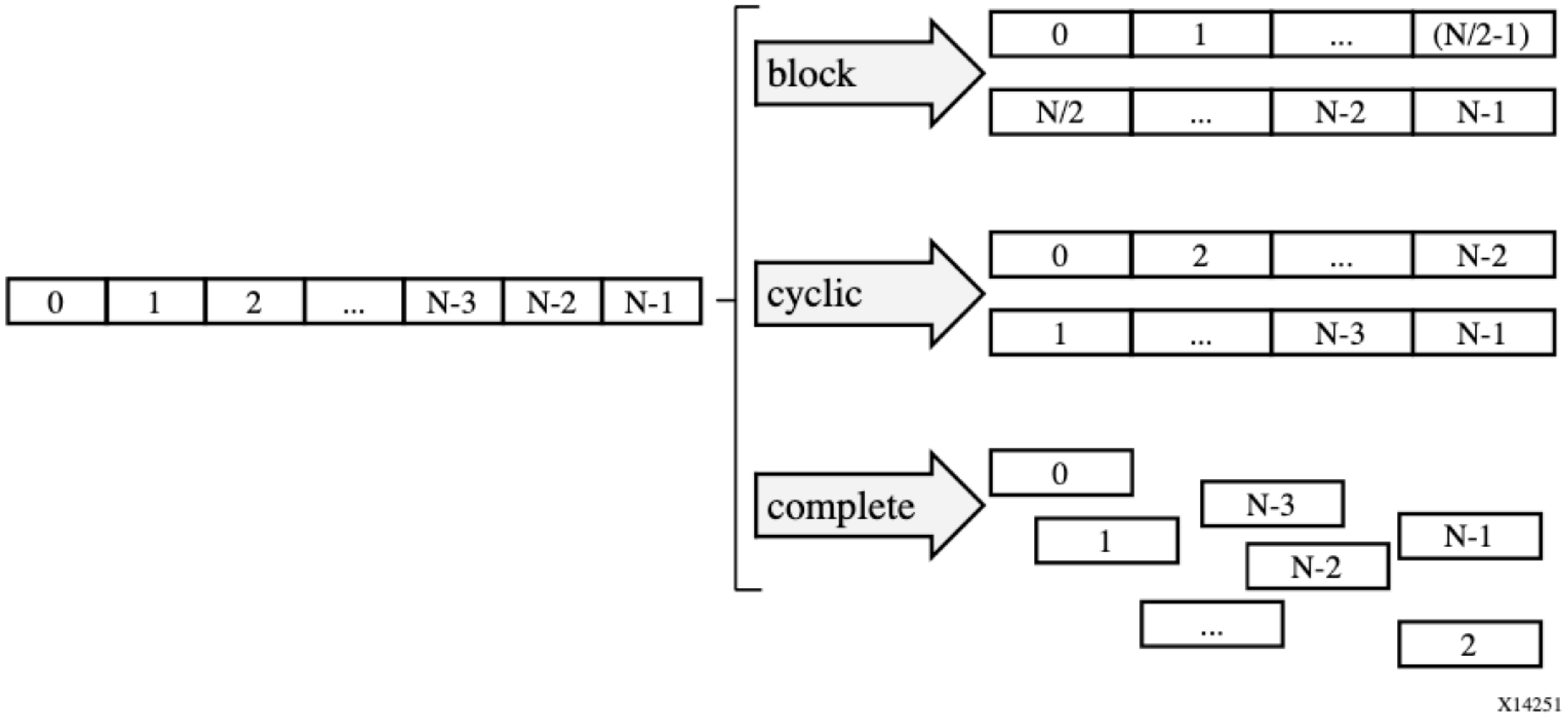
- For pipelined (parallelized) architectures, partitioning your array appropriately is important:
  - arrays are BRAMs by default unless they are partitioned (or reshaped) for pipelined data flow.
  - if there are two accesses to the same array in the loop body, it will need two read operation through the same memory port. So II becomes 2 and latency increases by 1 clock cycle!



Partitioning arrays

How do we solve this? The idea is to break one array into multiple small parts, so that we can access them at the same time.

Figure 57: **Array Partitioning**

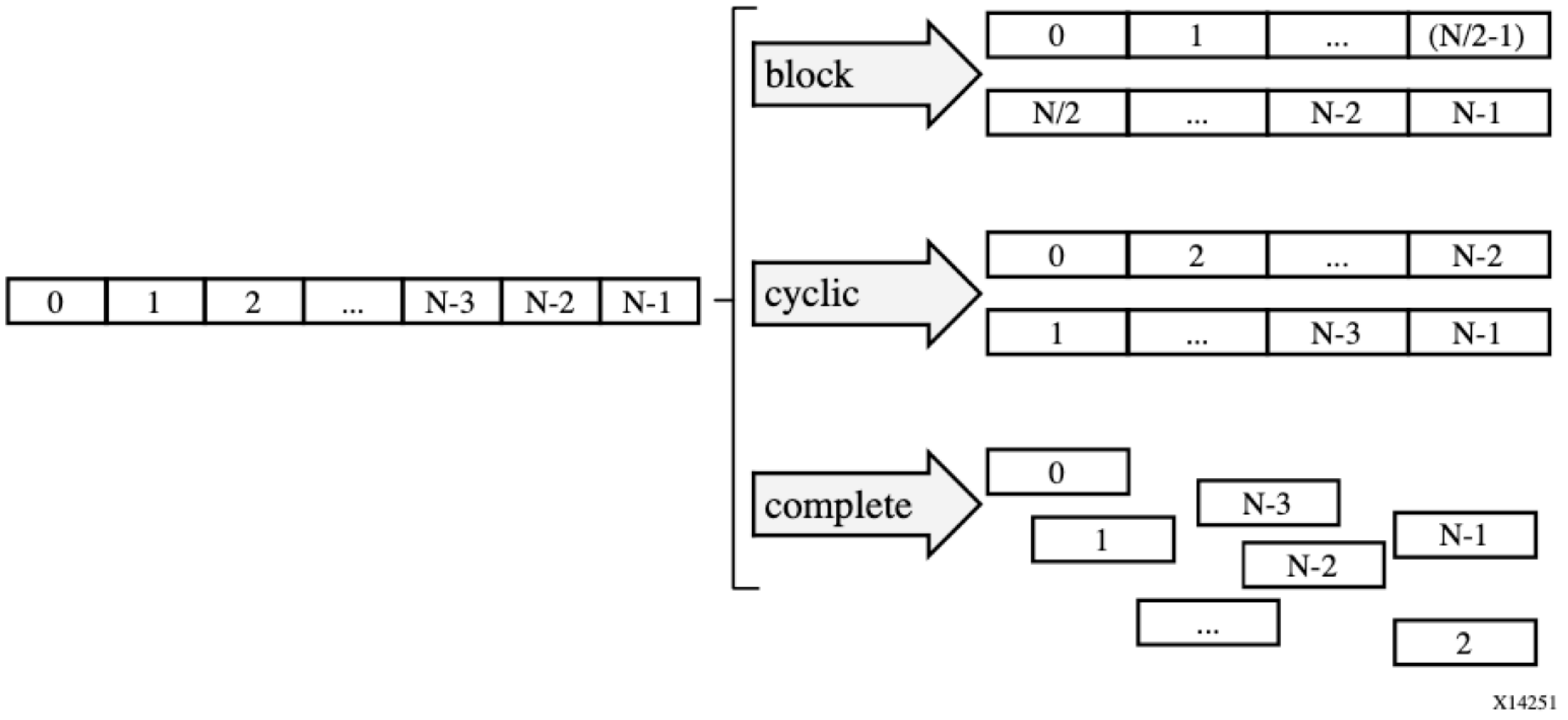


The **complete** scheme split the array into individual elements and so requires the smallest memory per each element → results in architecture with multiple small memories or registers (LUTs) instead of one large memory

Partitioning arrays

How do we solve this? The idea is to break one array into multiple small parts, so that we can access them at the same time.

Figure 57: **Array Partitioning**

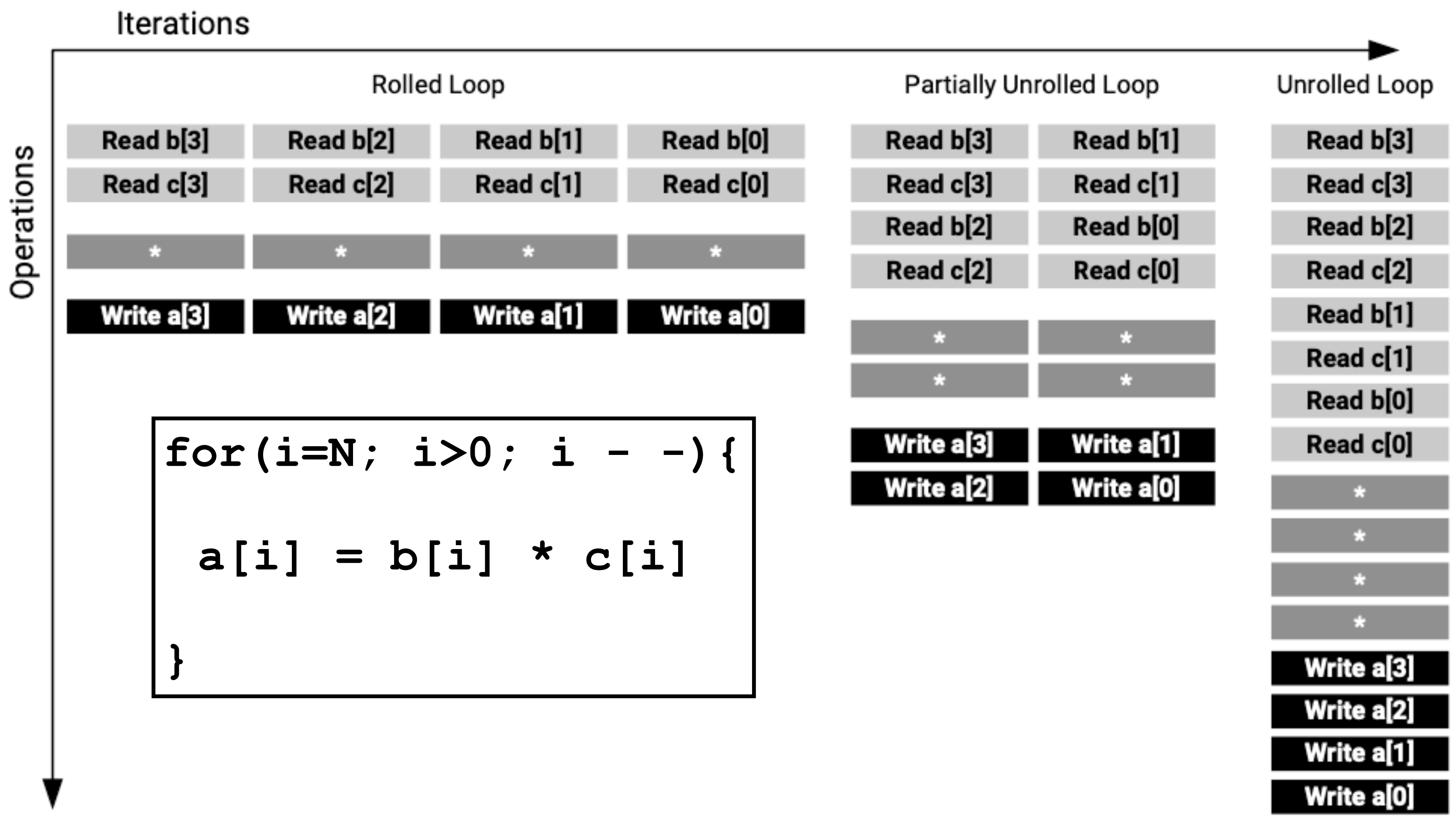


Example from hls4ml_prj/firmware/nnet_utils/nnet_dense_latency.h:

```
#pragma HLS ARRAY_PARTITION variable=biases complete
#pragma HLS ARRAY_PARTITION variable=mult complete
#pragma HLS ARRAY_PARTITION variable=acc complete
```

Loop unrolling

To optimize loops, people often suggest **unrolling** with `` `#pramga HLS unroll` ``: creates dedicated logic of the body loop for each of its iteration, so the entire loop can be run concurrently



```
for(i=N; i>0; i - -){

 a[i] = b[i] * c[i]

}
```

Loop unrolling in hls4ml

Unrolling can as well be obtained with the pipeline pragma:

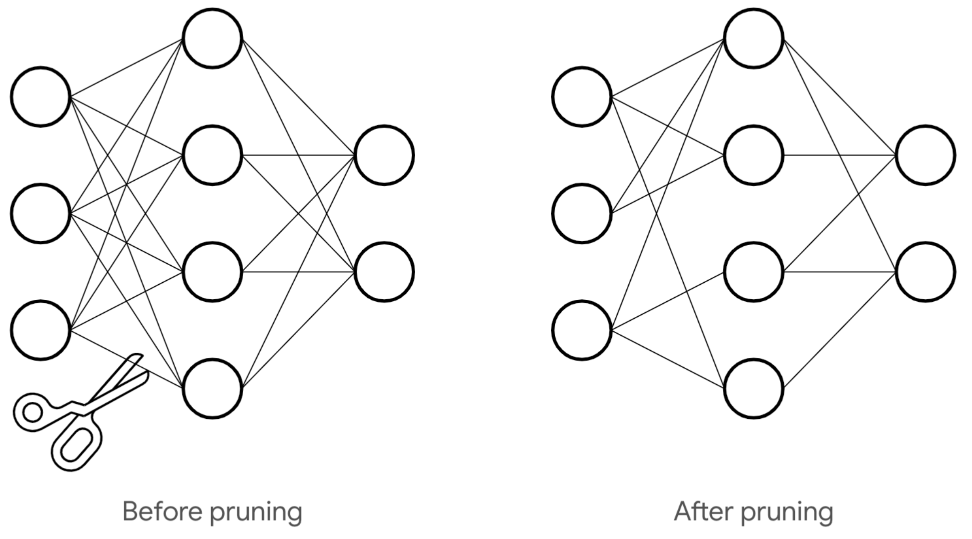
in hls4ml the **reuse_factor** defines how much to unroll and so defines the II and the latency

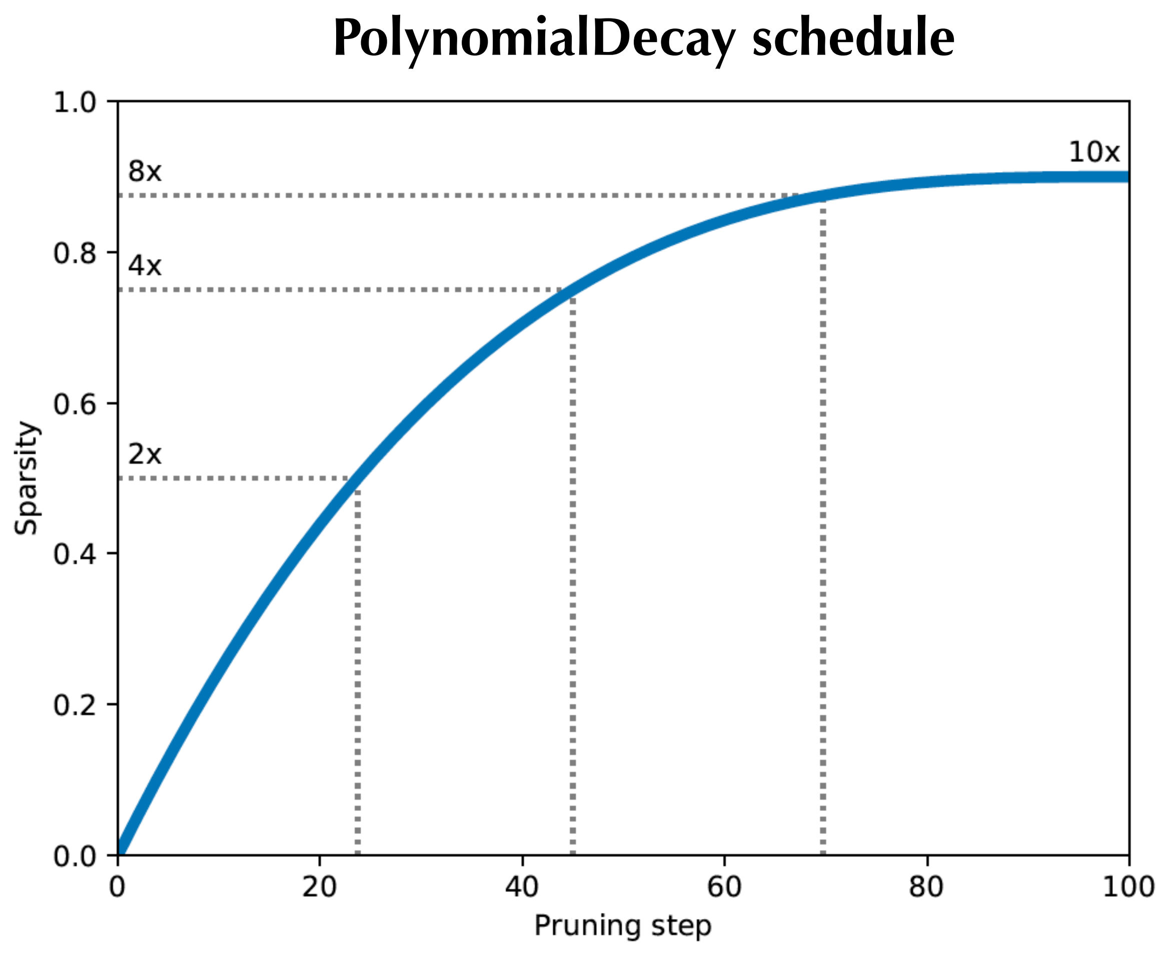
```
Example from hls4ml_prj/firmware/nnet_utils/nnet_dense_latency.h:
// For parallel inputs:
//   - completely partition arrays -- target fabric
//   - if we have an unroll factor, limit number of multipliers
#pragma HLS PIPELINE II=CONFIG_T::reuse_factor
```

Efficient NN design: compression

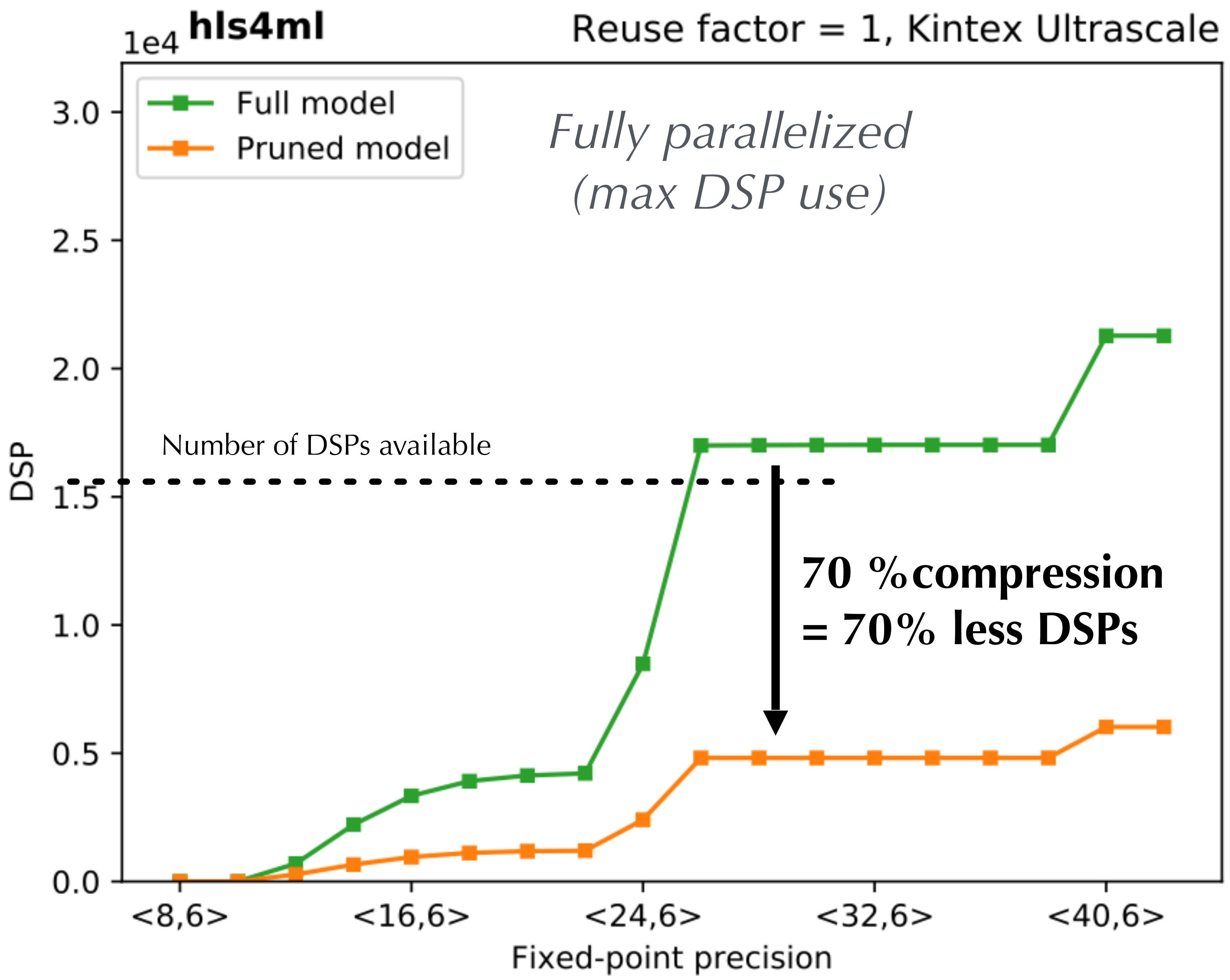
- Neural Network compression is a widespread technique to reduce the size, energy consumption, and overtraining of deep neural networks
- Several approaches in literature [arxiv.1510.00149, arxiv.1712.01312, arxiv.1405.3866, arxiv.1602.07576, doi:10.1145/1150402.1150464]
- Today we will test the tensorflow model sparsity toolkit
    - https://blog.tensorflow.org/2019/05/tf-model-optimization-toolkit-pruning-API.html

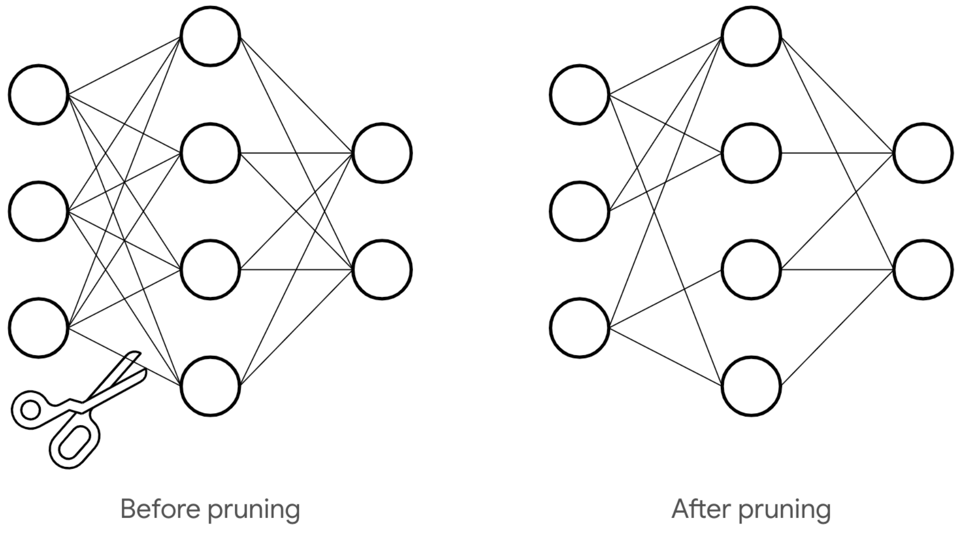
Main idea:
iteratively remove low magnitude weights, starting with 0 sparsity, smoothly increasing up to the set target as training proceeds





Efficient NN design: compression





Optional notebooks

- **Part 5:** boosted decision trees

  `notebook: part5_bdt.ipynb`

- **Part 6:** convolutional neural networks

  `notebook: part6_cnns.ipynb`

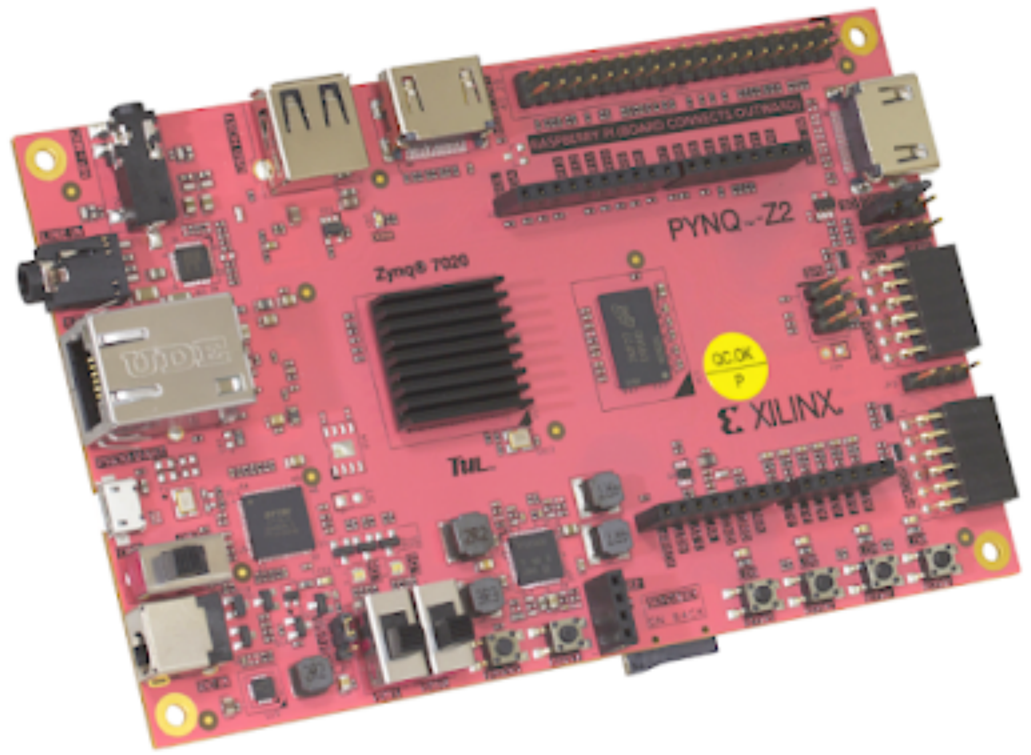
- **Part 7:** deployment on FPGA board (demo)

  `notebooks: part7*.ipynb`

The PYNQ-Z2 board

- In part7 you build a demonstration of a NN inference acceleration on the PYNQ-Z2 board
  - it does not connect through PCIe but it's a system on chip (SoC), i.e. all components are on the same board (including microcontrollers)
  - it can be easily programmed with python code / jupyter notebook
    → easy software interface and framework for rapid prototyping and development
- It uses a Xilinx Zynq Z7020 SoC device which is tiny wrt the one we have tested so far in this course
  - only ~200 DSPs instead of 12K !

Cost is less than 200$!!



http://www.pynq.io/home.html