BDT for Tau Identification in the ATLAS Level-1 Trigger

David Reikher

Fast Machine Learning for science

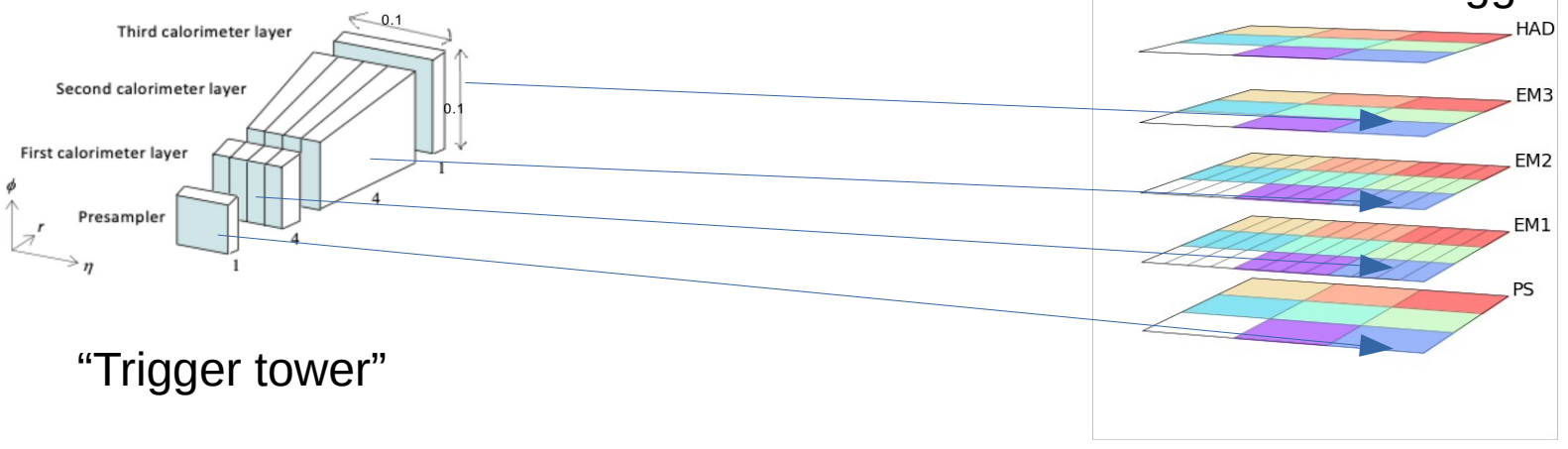
Imperial College London

September 25, 2023

Tau Identification

ATLAS Trigger must identify taus for tau-based physics analyses
Challenging experimental signature of taus based on calorimeter energy depositions
Two candidate algorithms – heuristic, ML-based

Input to algorithm:
“Trigger object” (TOB)
- A block of 3x3 trigger towers



“Trigger tower”

Further details on ATLAS trigger: arXiv:2305.16623

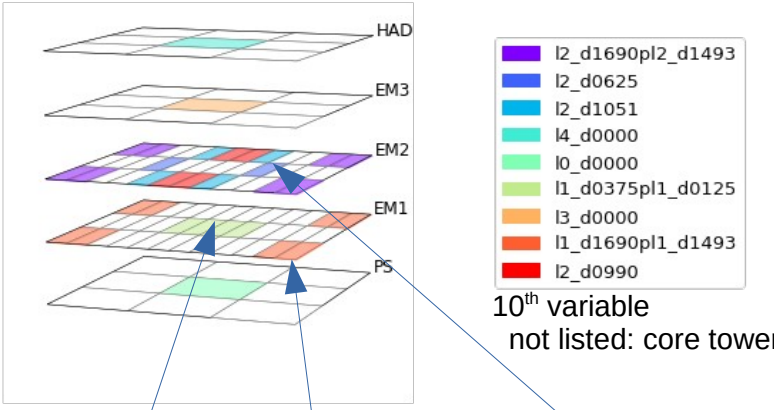
Comprised of 99 “supercells”
=> 99 16-bit numbers

Input Variables

Tried different complex variables,
But HLS implementation suggests prohibitive resource usage and latency

Variables must be:
- Quick to compute (max. 3 cycles)
- Light in resources

10 selected variables using SHAP ranking*



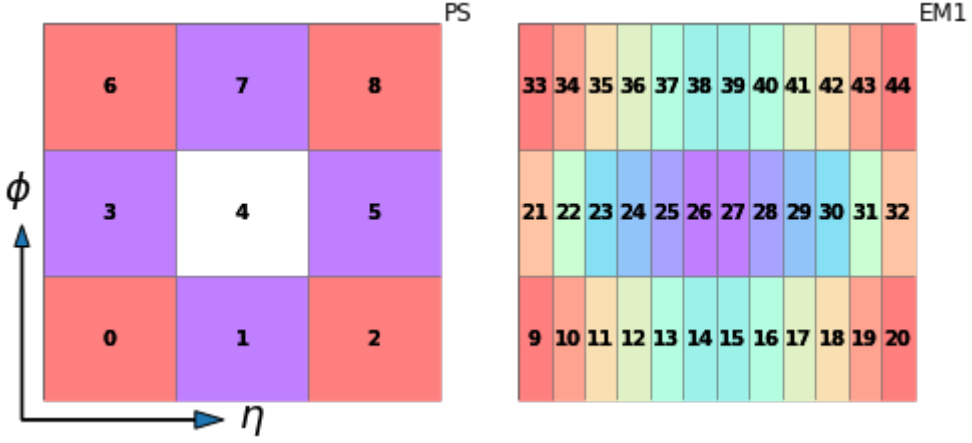
10th variable
not listed: core tower sum

2-stage adder

1-stage adder

3-stage adder

Supercells located symmetrically around TOB axis



* See slides 13,14

Hardware and Firmware

- Target device: Xilinx Virtex™ 7 FPGA
- Clock cycle: 5 ns
- Latency budget - 12 cycles
- Resource budget – enough for shallow BDT
  - Max. depth=2, ntrees=32, pruned

Tools

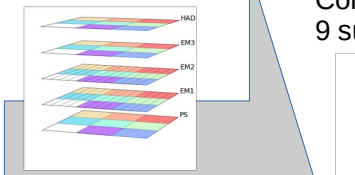
- BDT model implementation (Python version)

– Convert trained XGBoost to VHDL

- Synthesis, implementation, simulation, bitfiles

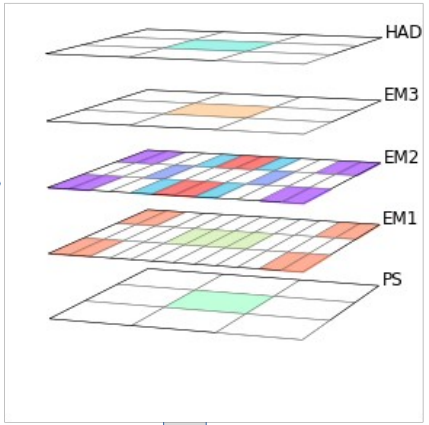
- Backend to Conifer, useful for diagnostics

Tau Algorithm Outline



Trigger Object
99
calorimeter
supercells

Compute 10 input variables =
9 sums over same color + 1 core tower sum



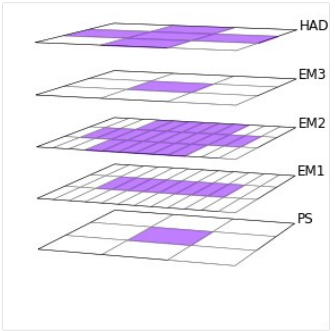
BDT
Max depth=2
Ntrees=32
(N_{Nodes}=220)

Parameters

Compare with Thresholds

BDT Score, E_T, Conditions

Use subset of variables
around core to estimate TOB energy



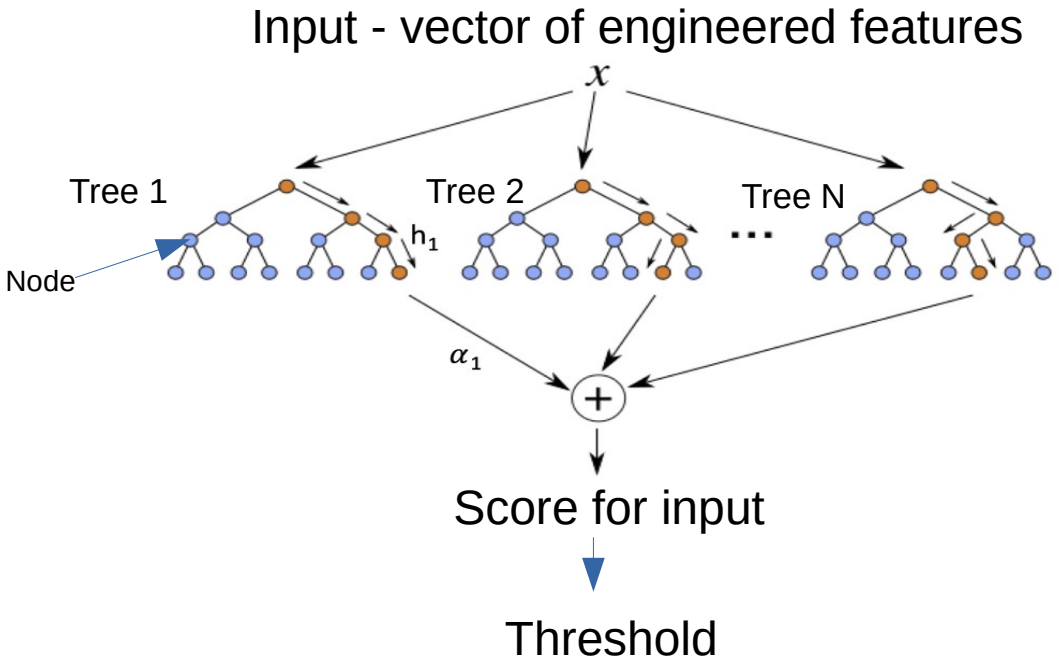
Status

- Performance at least as good as heuristic algorithm
- Resource usage on FPGAs is lower than heuristic algorithm
- Advanced testing stage
  - Fully implemented in firmware and simulation software
  - Ongoing tests on detector hardware
  - Ongoing checks on data collected during 2022-2023

Thank you!

Some more interesting details in the backup :)

BDT on FPGA



On FPGA:
- Compute all nodes in parallel
- Traverse each tree
- Sum N scores

Hyperparameters:
- Number of trees
- Max. depth of tree
- How much pruning

These determine N_{nodes}
- determines resource usage and whether the design meets timing

Resource usage vs hyperparameters

Resource Utilization

More Nodes (low g, high M or N) => More resource usage

Higher M – exponentially higher resource usage ($N_{nodes} \sim 2^{M+1}$)

Higher N – linearly higher resource usage ($N_{nodes} \sim N \times N_{avg.\ nodes\ in\ tree}$)

Latency

Higher max_depth – higher latency (~ linear with M)

Higher nt – higher latency (addition of N numbers ~$\log_2(N)$)

Ability of integrated BDT design to meet timing

Depends mainly on number of nodes in BDT

From experiments, if N_{nodes}>~220, full firmware doesn't meet timing with Vivado 2022.1

N – number of trees, M – max. depth

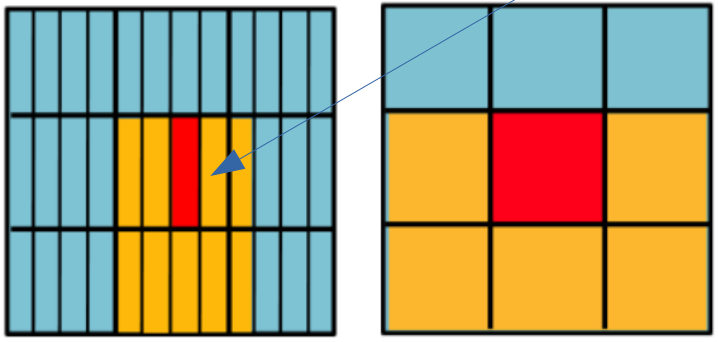
How the tau discriminants are used

Heuristic

BDT

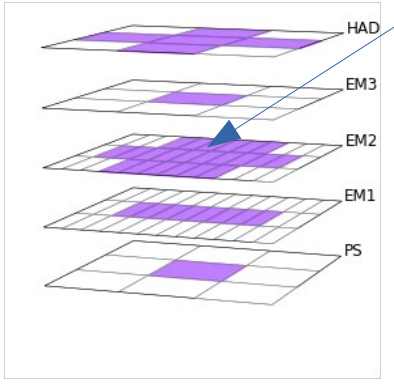
"Stage 1": E_T Cut

Thresh. 1 - Cut on TOB E_T (cluster-based)

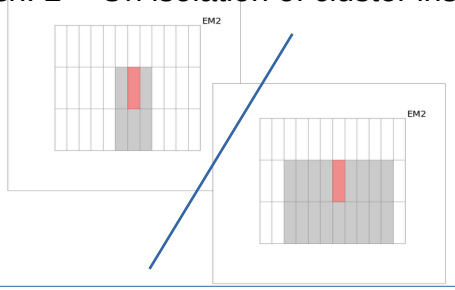


Fine layer (EM1,EM2) Coarse Layer (PS, EM3, HAD)

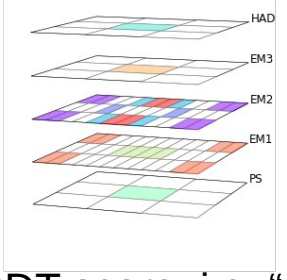
Thresh. 1 - Cut on TOB E_T (fixed around core)



Thresh. 2 – On isolation of cluster inside EM2



"Stage 2": Fine tuning



Thresh. 2 – On BDT score, i.e. "complex isolation" **

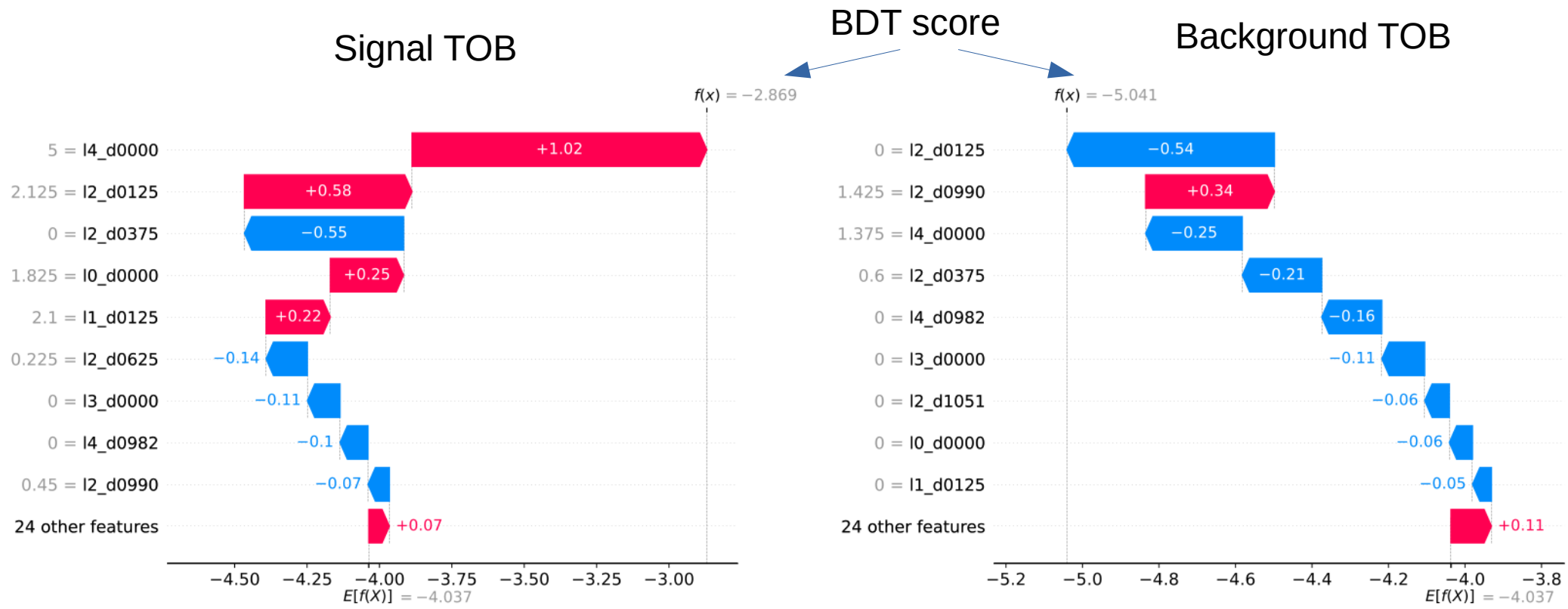
** BDT <=> set of cuts on E_T depositions a certain distance from TOB center

Constraints

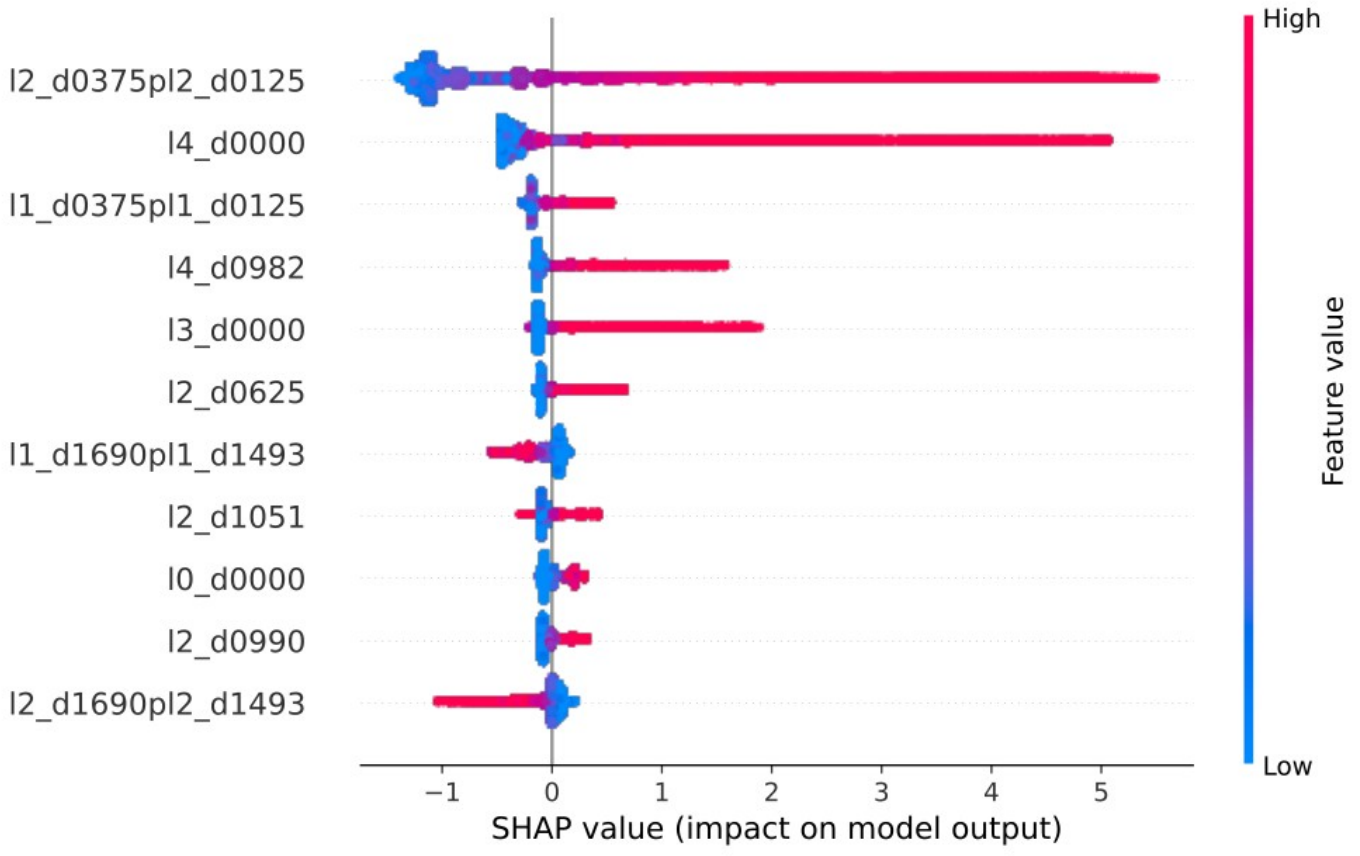
Based on already existing heuristic algorithm

- Latency
  - 12-cycles @ 200 Mhz
  - Fully pipelined
- Input:
  - 99 supercells, digitized to 25 MeV (16 bit)
  - 8 parameters for thresholds
    - 6 discriminant thresholds (8 bit)
    - 2 energy thresholds (16 bit)
- Output:
  - E_T estimate in TOB (16 bit)
  - 2 L/M/T working points (2 bits each)
  - Does TOB have maximum in central tower (1 bit)
- Resources
  - Same or less than heuristic algorithm

Explanation using SHAP values



SHAP ranking of selected variables



Some challenges and resolutions

Challenge I – switching models

- Quickly update BDT in firmware if re-training is required
- Modify summation schemas for TOB energy estimation

Sums - “AdderTree” VHDL entity

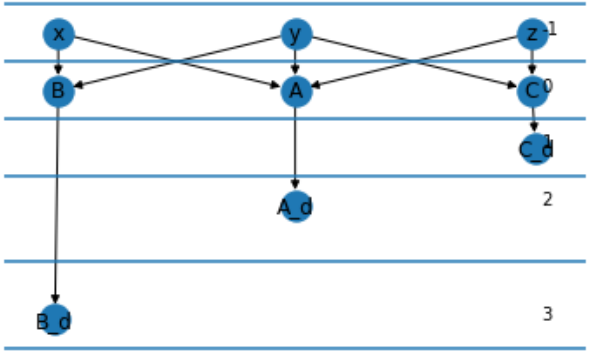
A=x+y+z
B=x+y
C=y+z

Latency requirements:

Input	Ready at cycle
A	2
B	4
C	1

graph_vhdl

```
entity AdderTree is
  port (
    CLK : in std_logic;
    IN_Words : in DataWords(2 downto 0);
    OUT_Words : out DataWords(2 downto 0);
    OUT_Overflows : out std_logic_vector(2 downto 0)
  );
end AdderTree;
```

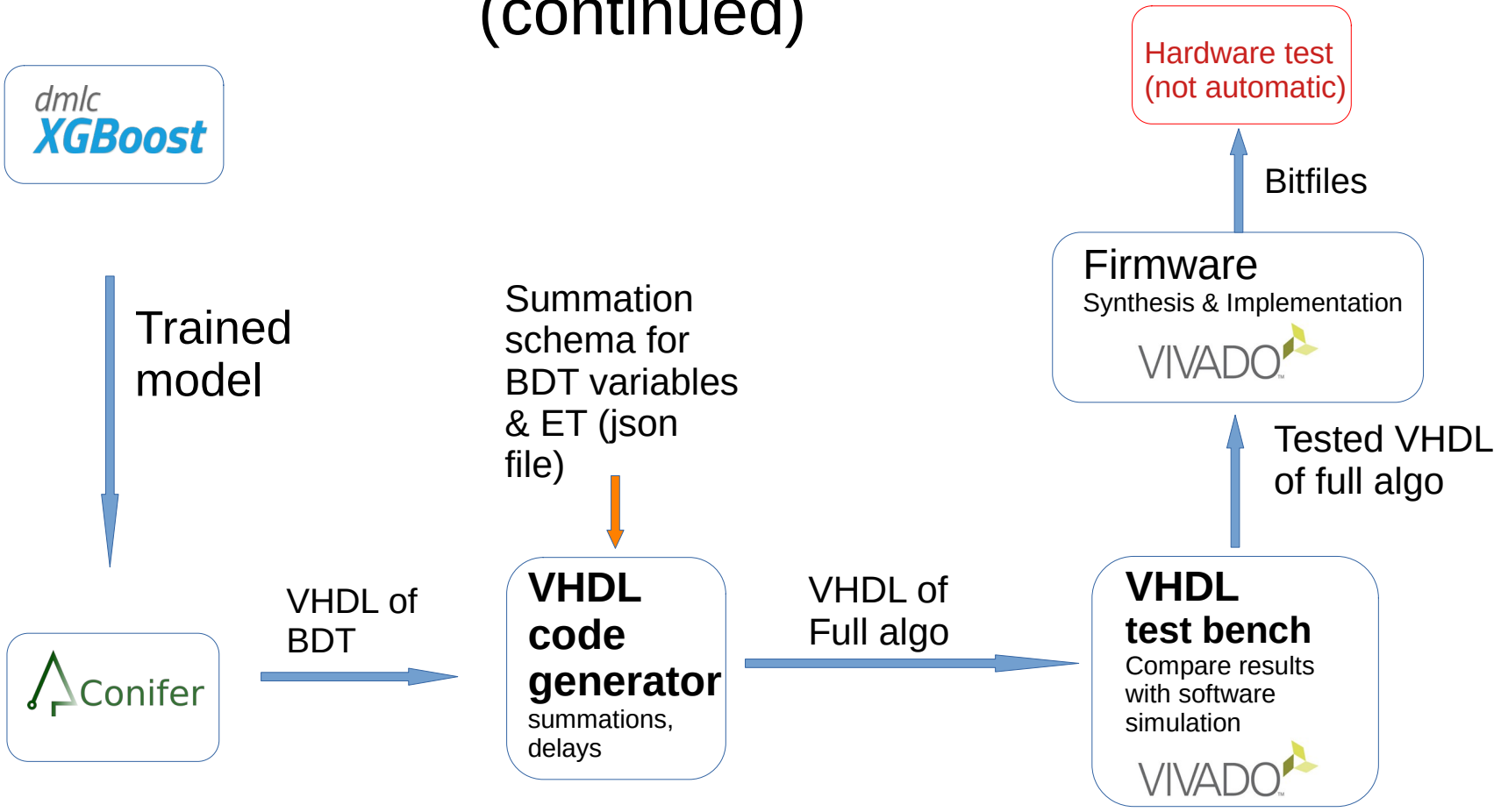


Similar “DelayTree” VHDL entity implementing all delays

Bonus: Clean VHDL code. All sums and delays in two separate auto-generated entities.

Challenge I – switching models
(continued)

Fully Automatic Pipeline:



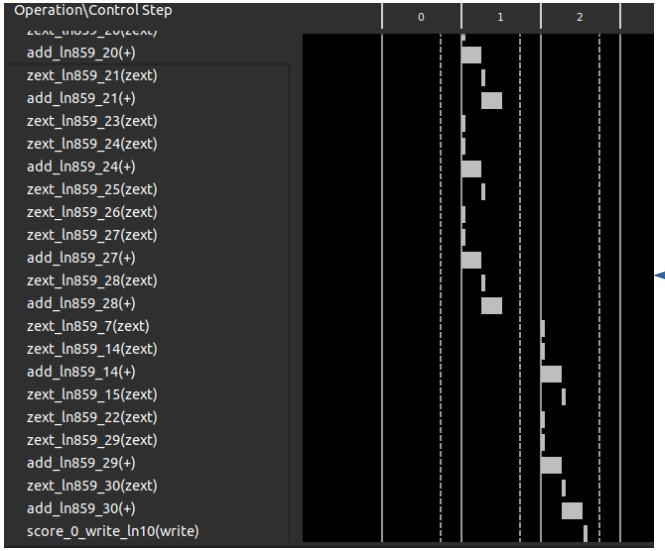
Challenge II – meeting timing

Problem:
Difficulty meeting timing for eFEXFirmware with BDT tau algorithm.

Possible Solution:
BDT produced by Vitis HLS is dense. To reduce logic density, increasing clock uncertainty parameter may help (using Vivado TCL command):

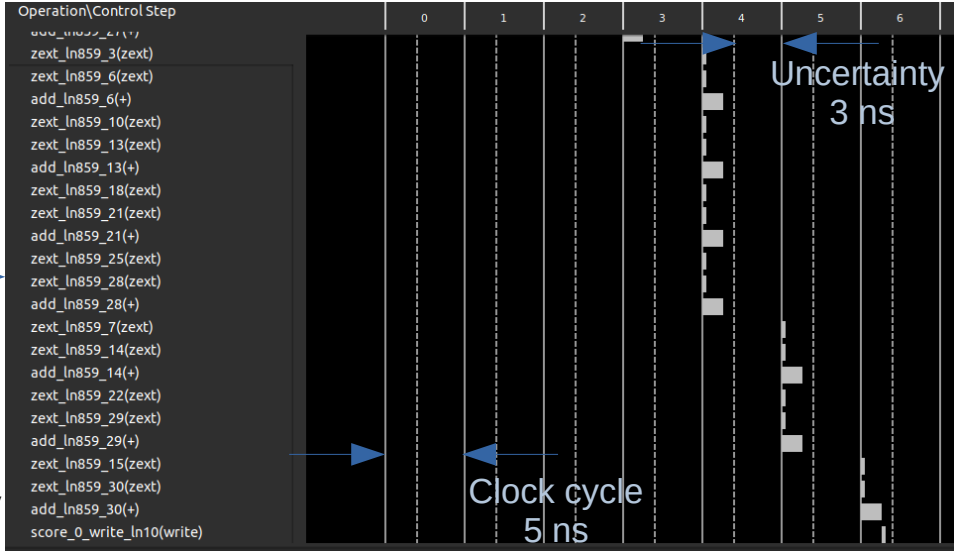
`set_clock_uncertainty 3`

Default (uncertainty=0.27 x period)



- Lower BDT latency
- “**Denser” logic**

- Higher BDT latency
- **“Thinner” logic**



[Vitis HLS “Schedule Viewer”]

Challenge III – C++ algorithm simulation

Problem:

Must be implemented in multiple places.

- Automatic validation of algorithm outputs from VHDL simulation
- Automatic validation of algorithm outputs from hardware simulation
- Algorithm implementation for R&D (Python *)
- Detector simulation software

Solution:

C++ implementation with wrapper accepting pointers to inputs.

For each of the above cases, just set the pointers during initialization.

```
void LVL1::eFEXtauBDTAlgo::setSCellPointers() {
  for (int phi = 0; phi < 3; phi++) {
    for (int eta = 0; eta < 3; eta++) {
      // Coarse layers
      m_bdtAlgoImpl->setPointerToSCell(eta, phi, 0, &m_em0cells[eta][phi]);
      m_bdtAlgoImpl->setPointerToSCell(eta, phi, 3, &m_em3cells[eta][phi]);
      m_bdtAlgoImpl->setPointerToSCell(eta, phi, 4, &m_hadcells[eta][phi]);
    }
    for (int eta = 0; eta < 12; eta++) {
      // Fine layers
      m_bdtAlgoImpl->setPointerToSCell(eta, phi, 1, &m_em1cells[eta][phi]);
      m_bdtAlgoImpl->setPointerToSCell(eta, phi, 2, &m_em2cells[eta][phi]);
    }
  }
}
```


* Python can call C++ using e.g. ctypes

Important Lessons Learned for BDTs on FPGA

- During R&D always keep in mind resource and latency restrictions.
  - Experiment early on with synthesis and implementation of simple designs
- Understand how the tools work and do not be afraid to tweak them (e.g. conifer/hls4ml).
- ML algorithms likely will require re-training (possibly at the worst possible time)
  - Develop automated tools to produce and test FPGA design (VHDL/Verilog, etc.)
- Whenever possible avoid using floats
- Try to use the same C++ code for all software implementations (detector simulation software, hardware simulation, test bench, R&D, etc.)
- Use Vitis HLS to get (very) rough estimate of resource usage for logic