Transferring HLS-Generated BDT Model into Existing Firmware in the ATLAS Level-1 Trigger

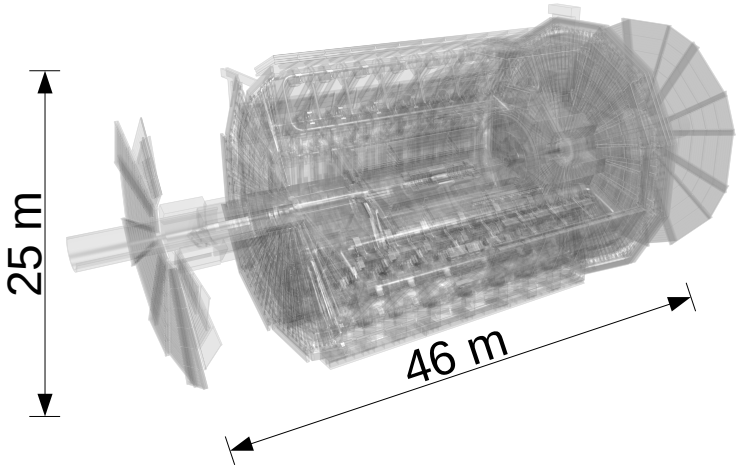
David Reikher

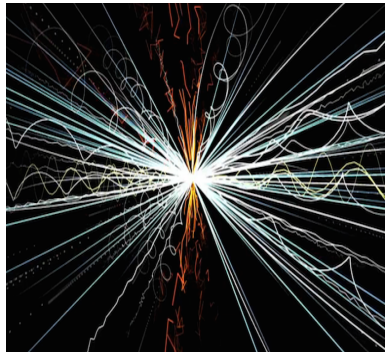
June 12, 2024

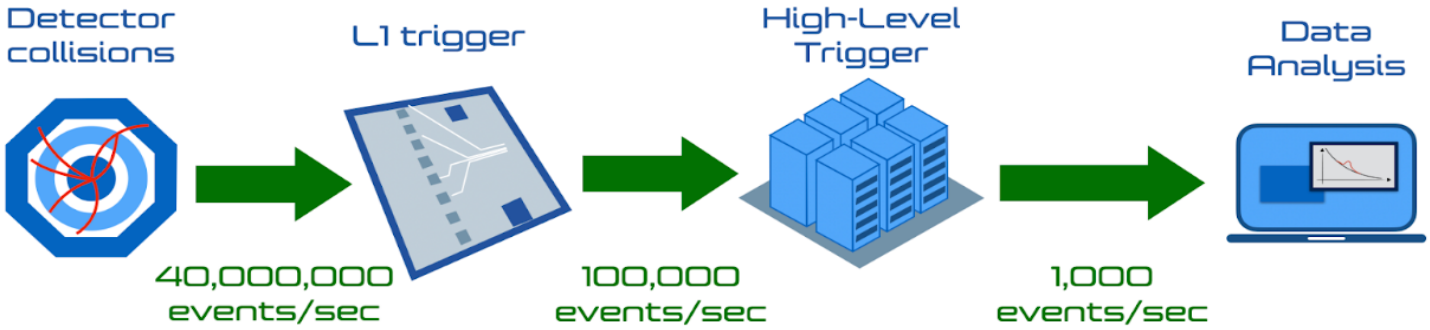
First FPGA Developer’s Forum

ATLAS L1 Trigger

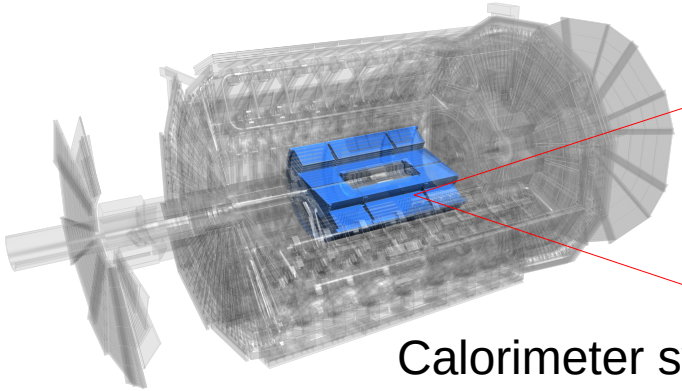
ATLAS detector



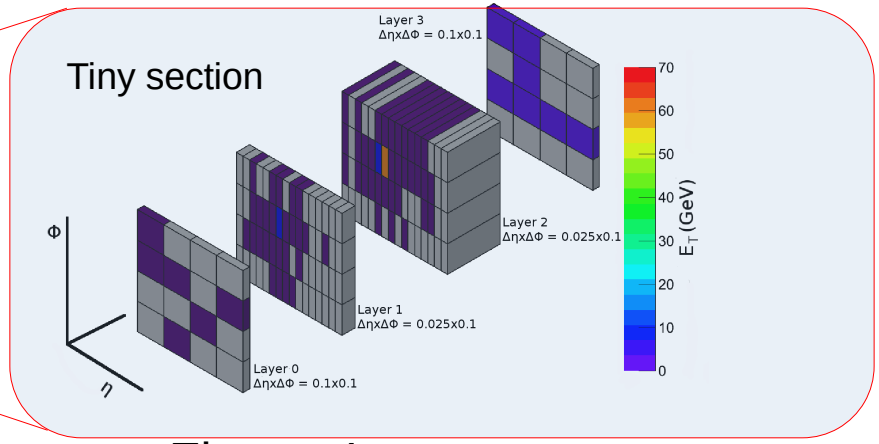




L1 Calorimeter Tau Trigger



Calorimeter system



Electron!
(but in our case it's **tau leptons**)

24 "eFEX" Modules

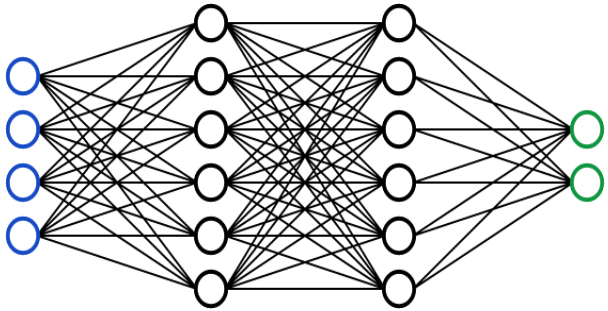
4 processing (+ 1 control) FPGAs

- Each FPGA runs 8 instances of tau ID algo.
- Each instance looks at a fixed region in the calorimeter
- Algorithm instances run at 200 Mhz
- Everything is fully pipelined

Machine Learning

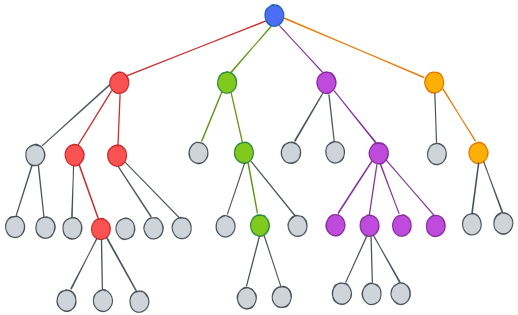
1. Trainable models that accept features and produce decisions
2. Better than heuristics at identifying cats/elementary particles in fuzzy settings

Neural Network



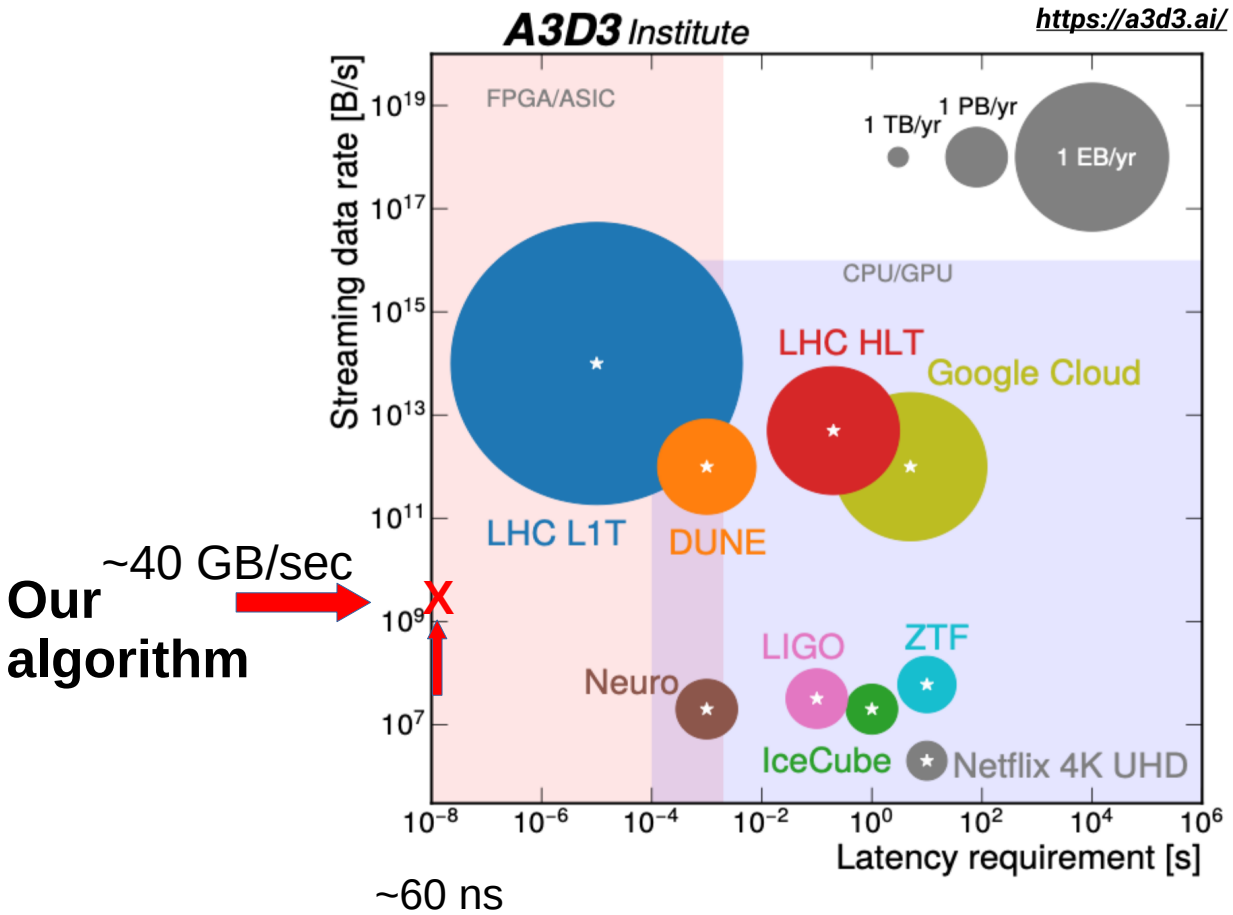
- Can accept very basic features
- Heavy on resources
- Lots of architectures
- Potentially higher performance

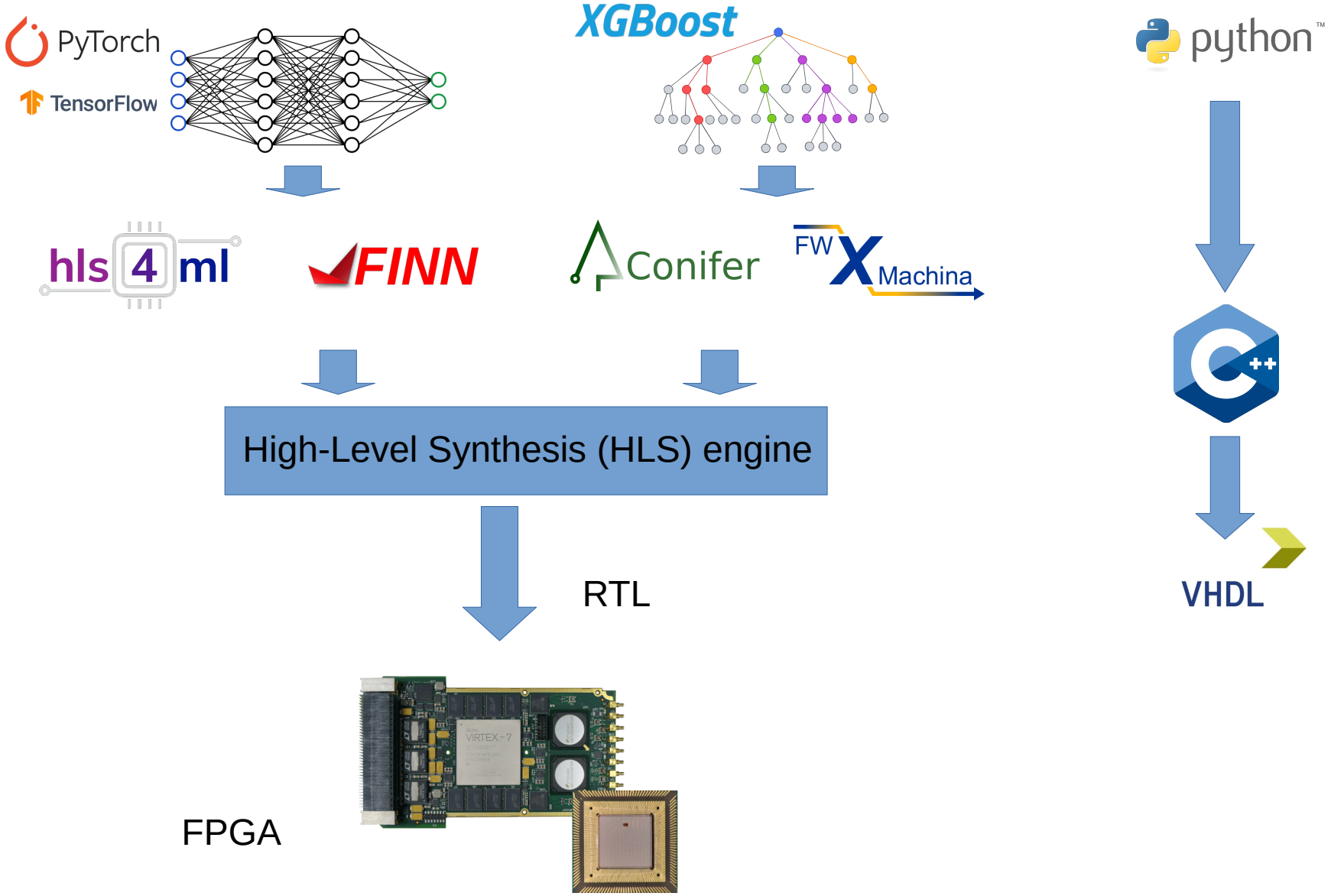
Boosted Decision Tree (BDT)



- Engineered features
- Light on resources
- Parallelizable evaluation => Fast
- Performance often like NN

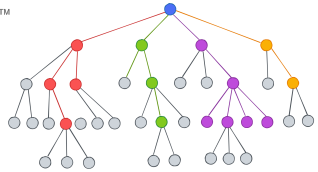
Latency & Data Rate




PyTorch
TensorFlow
XGBoost
python
hls 4 ml
FINN
Conifer
FW X Machina
High-Level Synthesis (HLS) engine
RTL
VHDL
FPGA

XGBoost

python™

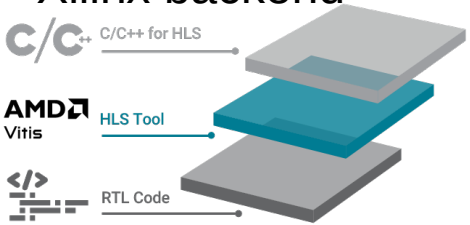


VHDL
BDT RTL code

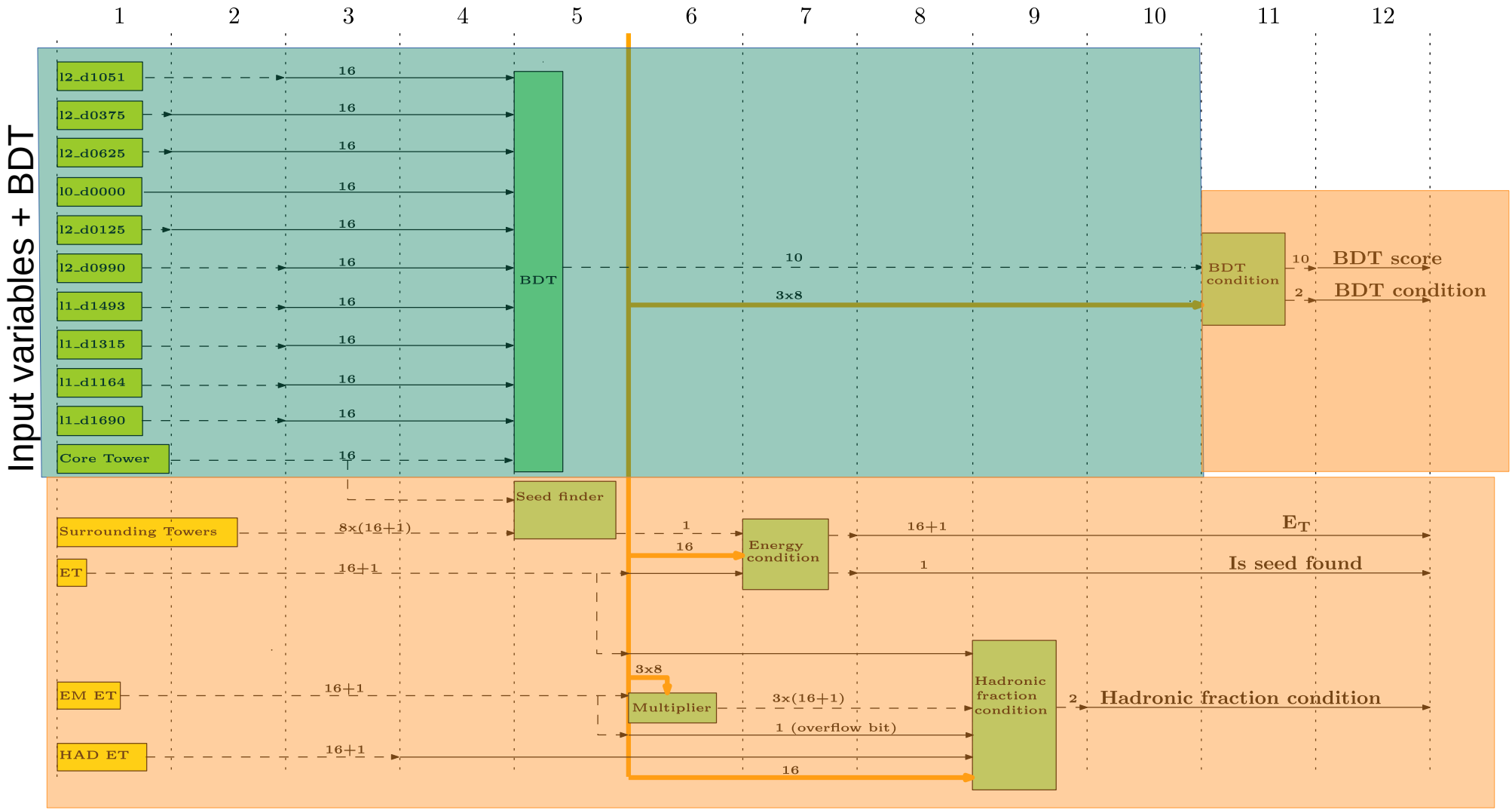
HLS

Conifer

Xilinx backend



Existing RTL



Other things

* Diagram not up to date

Requirements

- Overall latency constraint: 12 cycles @ 200 Mhz
- Fully pipelined – produce result every cycle
- High Flexibility for R&D and future changes
- Existing interface

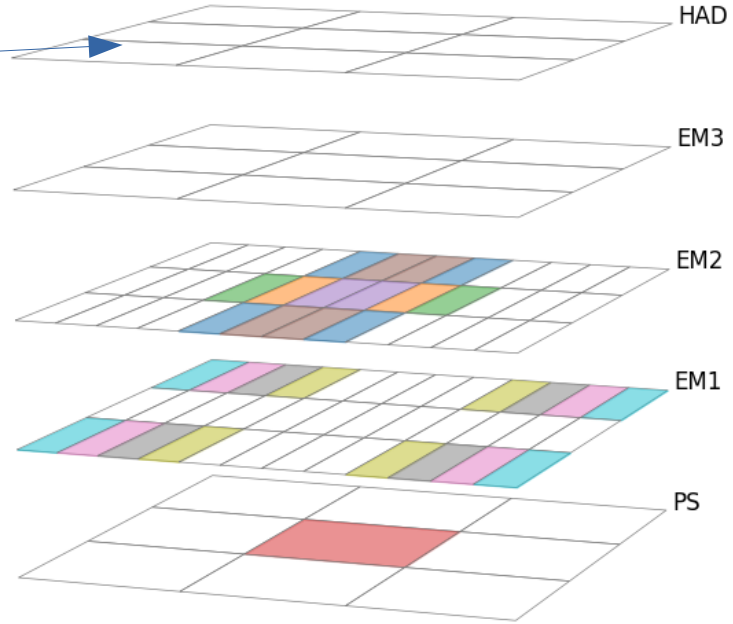
Flexibility

- Re-training very likely
  - Will change the BDT design’s latency
- BDT input variables might change
  - Flexibility in variable computation
- Don’t neglect the simulation
  - Must always reflect the model

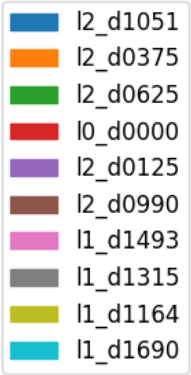
BDT Variables

Raw input:
99 cells

Each cell:
16-bit logic vector



11 variables:



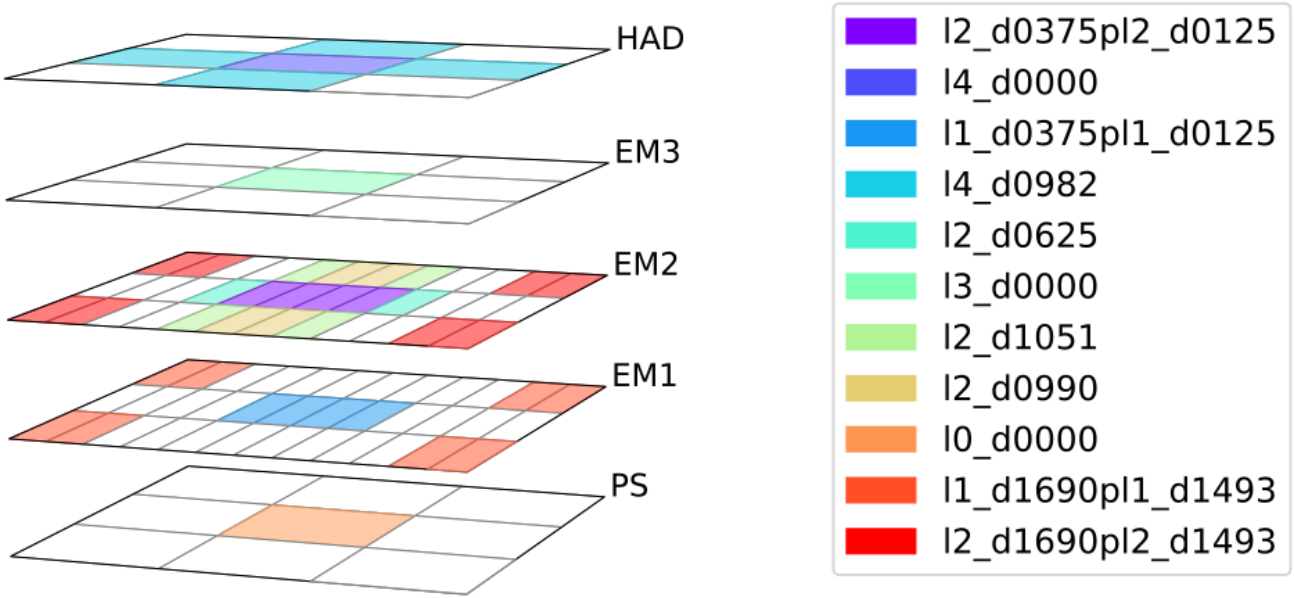
Each variable:
Sum over same-colored cells

Not visible:
sum over all cells in central “tower”

Sum of at most 4 cells = 2-stage adders = 2 cycles

But what if we want to sum over 8 cells? Different cells?

Whoops, these perform better *



* Not really, the actual ones are on the previous slide

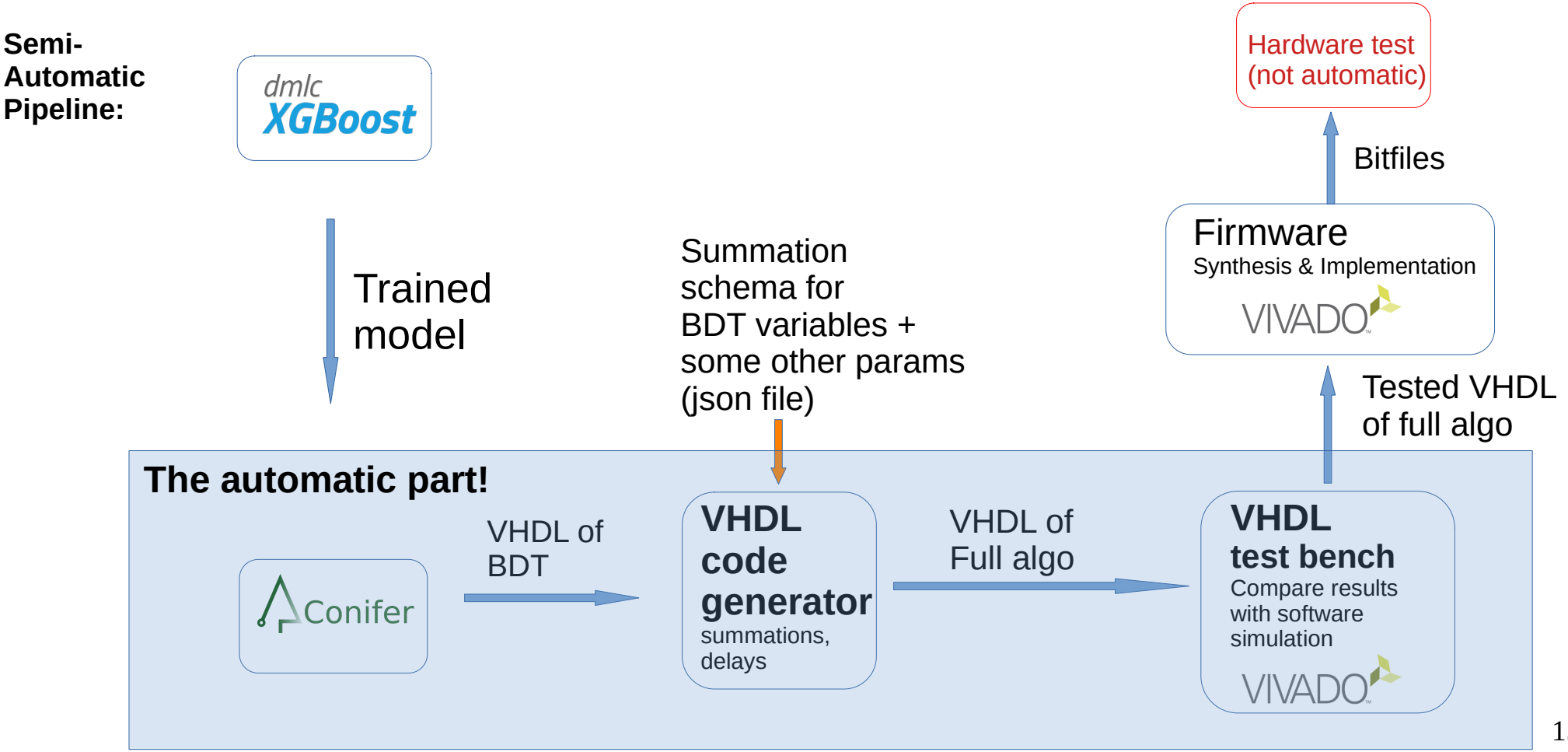
BDT Variables

- Raw input: 99 16-bit numbers
- Of which we pick N subsets of varying size (all configurable)
- All of them must be ready at same cycle
  - Those that are not must be delayed
- Expected to change many times

BDT RTL

- BDT structure parameters affect BDT latency
  - They are expected to change
- BDT score is used downstream
  - Other signals must be delayed accordingly

python™ to the Rescue



Inside the Code Generator- ”AdderTree”

Input

Compute these sums:

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

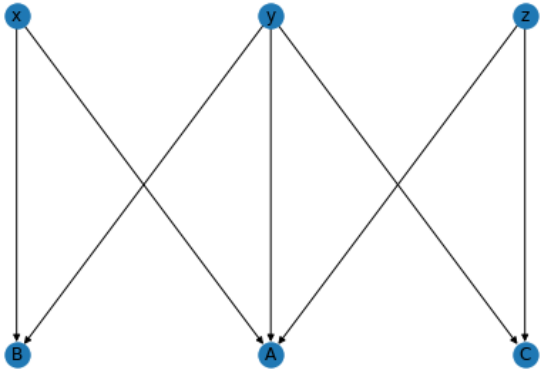
Output cycle requirements:

Sum	Ready at cycle #
A	unspecified
B	8
C	7

Cycles at which inputs are available:

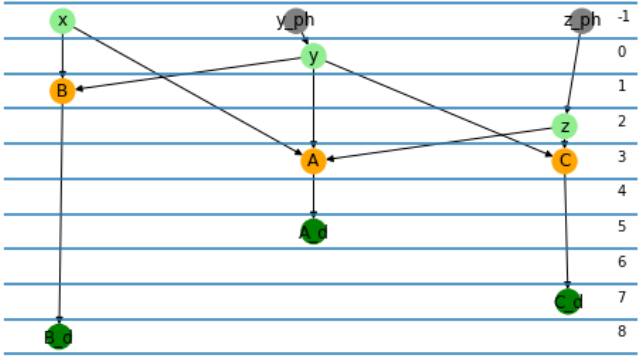
Input	Available At cycle #
x	0
y	1
z	3

Represent sum as graph



NetworkX
Network Analysis in Python

Compute latencies




```
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;
architecture Behavioral of AdderTree is
  signal A_out : DataWordWithCarry := (others => '0');
  signal B_OUT_Word_TO_DelayWC_B_B_d_IN_Word : DataWord
  signal C_OUT_Word_TO_DelayWC_C_C_d_IN_Word : DataWord
  signal DelayWC_y_C_OUT_Word_TO_C_IN_Words : DataWordW
  signal B_out : DataWordWithCarry := (others => '0');
  signal x : DataWordWithCarry := (others => '0');
  signal DelayWC_x_B_OUT_Word_TO_B_IN_Words : DataWordW
  signal DelayWC_x_A_OUT_Word_TO_A_IN_Words : DataWordW
  signal y : DataWordWithCarry := (others => '0');
  signal DelayWC_y_A_OUT_Word_TO_A_IN_Words : DataWordW
  signal C_out : DataWordWithCarry := (others => '0');
  signal z : DataWordWithCarry := (others => '0');
begin
  x(x'high - 1 downto 0) <= IN_Words(0);
  y(y'high - 1 downto 0) <= IN_Words(1);
  z(z'high - 1 downto 0) <= IN_Words(2);
  A: entity work.MultiAdderWithCarry
  generic map (
    stage => 2,
    delay => 0
  )
  port map (
    CLK => CLK,
    IN_Words(0) => DelayWC_x_A_OUT_Word_TO_A_IN_Words,
    IN_Words(1) => DelayWC_y_A_OUT_Word_TO_A_IN_Words,
    IN_Words(2) => z,
    IN_Words(3) => ZERO_DATA_WORD_WITH_CARRY,
    OUT_Word => A_out
  );
  B: entity work.MultiAdderWithCarry
  generic map (
    stage => 1,
    delay => 0
```

Generate fully pipelined VHDL!

AdderTree

Input

Compute these sums:

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

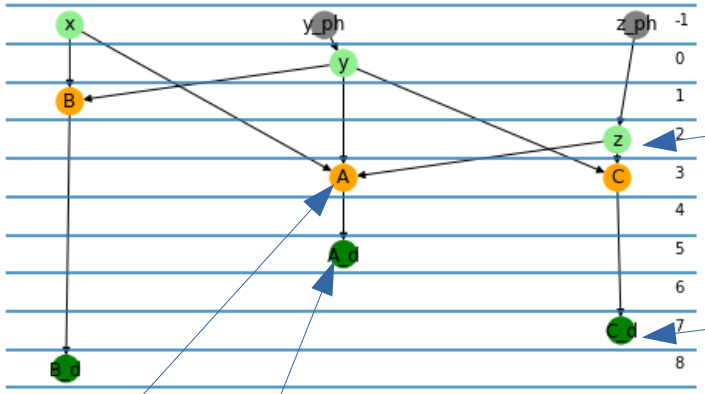
Output cycle requirements:

Sum	Ready at cycle #
A	unspecified
B	8
C	7

Cycles at which inputs are available:

Input	Available At cycle #
x	0
y	1
z	3

Each orange node accepts N inputs
And produces sum after $\lceil \log_2 N \rceil$ cycles



Input “z” is available only at cycle 3

Sum “C” ready, after delay

A=x+y+z has 3 inputs, ready after 2 cycles

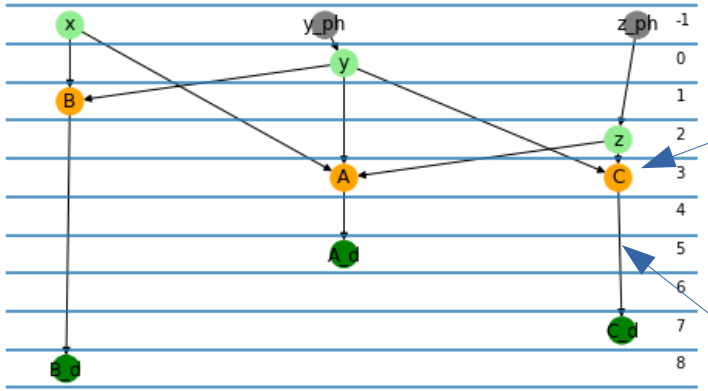
AdderTree VHDL output

```
--! !!!!!!!!!!!!!!!!! AUTO-GENERATED FILE. PLEASE DO NOT MODIFY !!!!!!!!!!!!!!!!!
--! Generated by graph_vhdl (https://gitlab.cern.ch/taul1ml/graph_vhdl), authored by David Reikher
--!
--! INPUTS:
--!   IN_Words(0)          <-------             x
--!   IN_Words(1)          <-------             y
--!   IN_Words(2)          <-------             z
--! OUTPUTS:
--!   OUT_Words(0)         ------->             A
--!   OUT_Words(1)         ------->             B
--!   OUT_Words(2)         ------->             C
--!
--! This adder tree implements the following sums:
--!   A = x + y + z
--!   B = x + y
--!   C = y + z
--!
--!
--! Outputs are ready at clock cycles marked by 'X':
--!   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
--! A |   |   |   |   |   | X |   |   |   |
--! B |   |   |   |   |   |   |   |   | X |
--! C |   |   |   |   |   |   |   | X |   |
```

AdderTree VHDL Output

```
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;
architecture Behavioral of AdderTree is
  signal A_out : DataWordWithCarry := (others => '0');
  signal B_OUT_Word_TO_DelayWC_B_B_d_IN_Word : DataWordWithCarry := (others => '0');
  signal C_OUT_Word_TO_DelayWC_C_C_d_IN_Word : DataWordWithCarry := (others => '0');
  signal DelayWC_y_C_OUT_Word_TO_C_IN_Words : DataWordWithCarry := (others => '0');
  signal B_out : DataWordWithCarry := (others => '0');
  signal x : DataWordWithCarry := (others => '0');
  signal DelayWC_x_B_OUT_Word_TO_B_IN_Words : DataWordWithCarry := (others => '0');
  signal DelayWC_x_A_OUT_Word_TO_A_IN_Words : DataWordWithCarry := (others => '0');
  signal y : DataWordWithCarry := (others => '0');
  signal DelayWC_y_A_OUT_Word_TO_A_IN_Words : DataWordWithCarry := (others => '0');
  signal C_out : DataWordWithCarry := (others => '0');
  signal z : DataWordWithCarry := (others => '0');
```

AdderTree VHDL Output




```
C: entity work.MultiAdderWithCarry
generic map (
  stage => 1,
  delay => 0
)
port map (
  CLK => CLK,
  IN_Words(0) => DelayWC_y_C_OUT_Word_TO_C_IN_Words,
  IN_Words(1) => z,
  OUT_Word => C_OUT_Word_TO_DelayWC_C_C_d_IN_Word
);
```

```
DelayWC_C_C_d: entity work.DelayWithCarry
generic map (
  delay => 3
)
port map (
  CLK => CLK,
  IN_Word => C_OUT_Word_TO_DelayWC_C_C_d_IN_Word,
  OUT_Word => C_out
);
```

```
OUT_Words(2) <= C_out(C_out'high - 1 downto 0);
```

```
OUT_Overflows(2) <= C_out(C_out'high);
```

MultiAdderWithCarry,
DelayWithCarry
Slightly modified entities originally written
By **Francesco Gonnella**

Usage

```python
from graph_vhdl.adder_graph import AdderGraph
from graph_vhdl.comments import AdderGraphCommentGenerator

g = AdderGraph.from_sums({"A":["x","y", "z"], "B":["x","y"], "C": ["y", "z"]},
                         input_order=["x", "y", "z"],
                         output_order=["A", "B", "C"],
                         input_ready_cycles={'y':1, 'z':3},
                         required_latencies={'B': 8, 'C':7})
g.draw() # Draw summation graph
g.draw_latency() # Draw summation graph with latencies computed

# Dump VHDL
adder_graph_comment_gen = AdderGraphCommentGenerator()
design = g.to_VHDLGraph(entity_name="AdderTree", comment_generator=adder_graph_comment_gen)
design.compile()
design.to_vhd_file("AdderTree.vhd")
```

More Pipeline Functionality

- Compute the delays based on generated BDT RTL latency
- Generate configuration file for the software simulation containing BDT model and input variable schema
- Run full algorithm in Vivado XSim test bench – compare with simulation
  - On real ATLAS sample
  - On randomly generated data (“fuzzing”)

Summary

- ML models are desirable in complex environments
- A BDT model is operational in the ATLAS L1 trigger in 2024
- Many challenges going from HLS model to production
  - Must be very flexible
  - Automation is crucial
- Code (currently only accessible to CERN account holders)
  - **AdderTree** - Readable auto-generated code for configurable fully pipelined summations and delays
    - https://gitlab.cern.ch/taul1ml/graph_vhdl
  - **Full pipeline** code available here:
    - https://gitlab.cern.ch/taul1ml/vhdl_bdt_testbench

References

XGBoost https://arxiv.org/abs/1603.02754

TensorFlow https://github.com/tensorflow/tensorflow

PyTorch https://github.com/pytorch/pytorch

hls4ml https://arxiv.org/abs/2103.05579

Conifer https://iopscience.iop.org/article/10.1088/1748-0221/15/05/P05026

FINN https://dl.acm.org/doi/10.1145/3020078.3021744

FWXMachina https://iopscience.iop.org/article/10.1088/1748-0221/17/09/P09039

NetworkX Network Analysis in Python https://github.com/networkx/networkx

erc

Grant # 945878
https://cordis.europa.eu/project/id/945878

Thank you!

A Word on Simulation

- Final algorithm must be simulated, often in separate places
  - Hardware simulation
  - Performance studies + further R&D
  - Full system simulation (e.g. for monitoring of production system)
- Implement simulation “core” and “interface”
  - Core remains fixed and has all core functionality
  - Interface changes from one environment to another