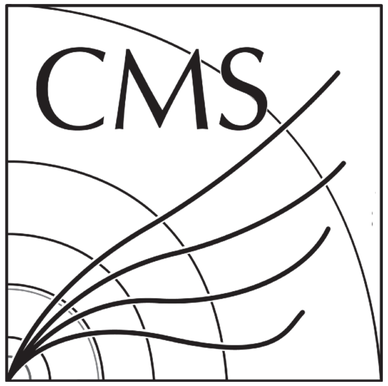
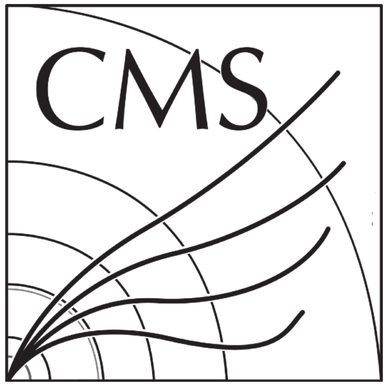
FPGAs, HLS, and Boosted Decision Trees with

Sioni Summers
sioni@cern.ch
sioni.web.cern.ch



FPGAs, HLS, and Boosted Decision Trees with

Section 1



Introduction

- *Boosted Decision Trees* (BDTs) or *Decision Forests* are a Machine Learning method to make predictions from data
- Conifer is a Python library for converting BDTs to FPGAs for fast inference
  - Different implementations for different use cases
- In this session we will:
  - Learn how BDTs work for training and inference
  - Learn three ways how BDT inference is implemented for FPGAs in conifer
  - Learn how to use conifer to deploy BDTs on FPGAs
- The aim is to both learn how to use conifer, and use it to study more about HLS and FPGA implementations
- Links, references:
  - Conifer GitHub repository: https://github.com/thesps/conifer
  - Conifer website (docs and downloads): https://ssummers.web.cern.ch/conifer/
  - Paper: *Fast Inference of Boosted Decision Trees in FPGAs for particle physics*

About me

- PhD in HEP from Imperial College London
  - PhD Thesis: “Applications of FPGAs to triggering in particle physics”
- Recently Senior Fellow, now Applied Physicist at CERN working on Level 1 Trigger Upgrade for CMS
  - Where we want to do complicated processing very fast on FPGAs and use HLS extensively
- I’ve mostly worked on designing and implementing detector reconstruction algorithms for Level 1 Trigger
  - Track & vertex reconstruction, particle flow, jets
- Also using Machine Learning in the triggers on FPGAs with low latency
  - **hls4ml** and **conifer** both as a developer and user

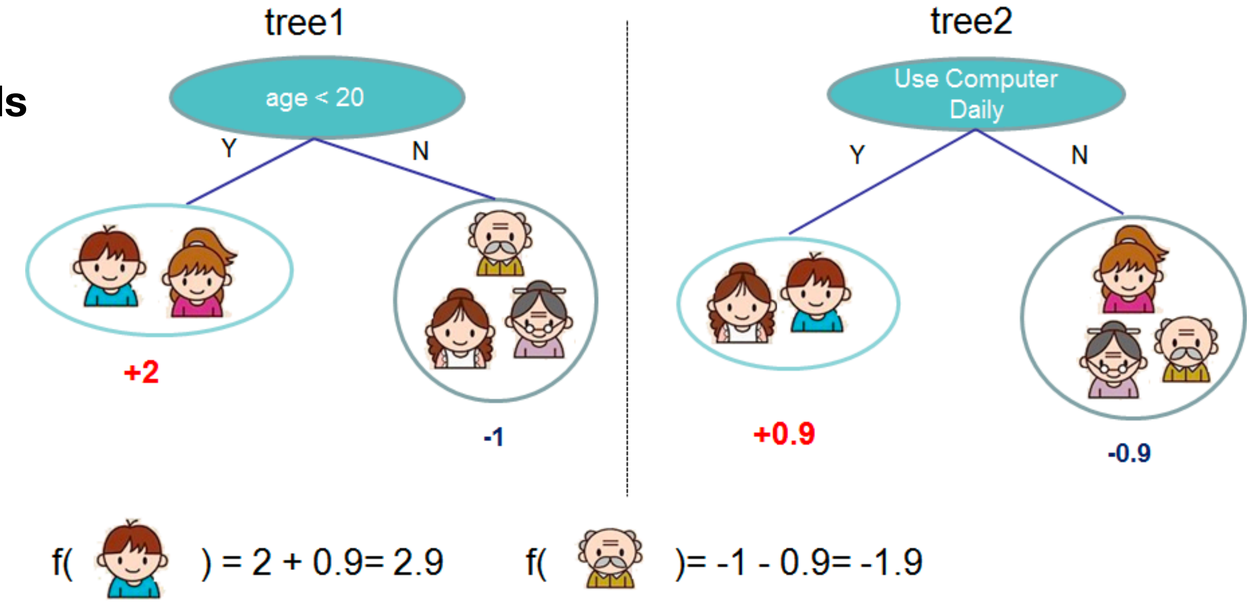
Quick ML Introduction

- Using XGBoost's Elements of Supervised Learning Introduction
- Train a **model** on training data to predict target variable y from features x
  - $y = f(\Theta, x)$ model parameters Θ
- **Train** to find best parameters according to an **objective function**
  - $obj(\Theta) = L(\Theta) + \Omega(\Theta)$ Loss function L, Regularization Ω
- Supervised learning trains on labelled data so we can evaluate some metric of prediction quality
  - e.g. mean squared error $L(\Theta) = \Sigma(y_i - \hat{y}_i)^2$ where y_i are our truth labels and $\hat{y}_i$ are the model predictions

Quick ML Introduction

- Using XGBoost's Elements of Supervised Learning Introduction
- Train a **model** on training data to predict target variable y from features x
- A Boosted Decision Tree model is an ensemble of Decision Trees
- The splits of each Decision Tree are chosen based on the training objective function
- In an ensemble each learner (tree) is relatively weak, but the aggregation is a stronger prediction

e.g. predict whether individuals will like a computer game



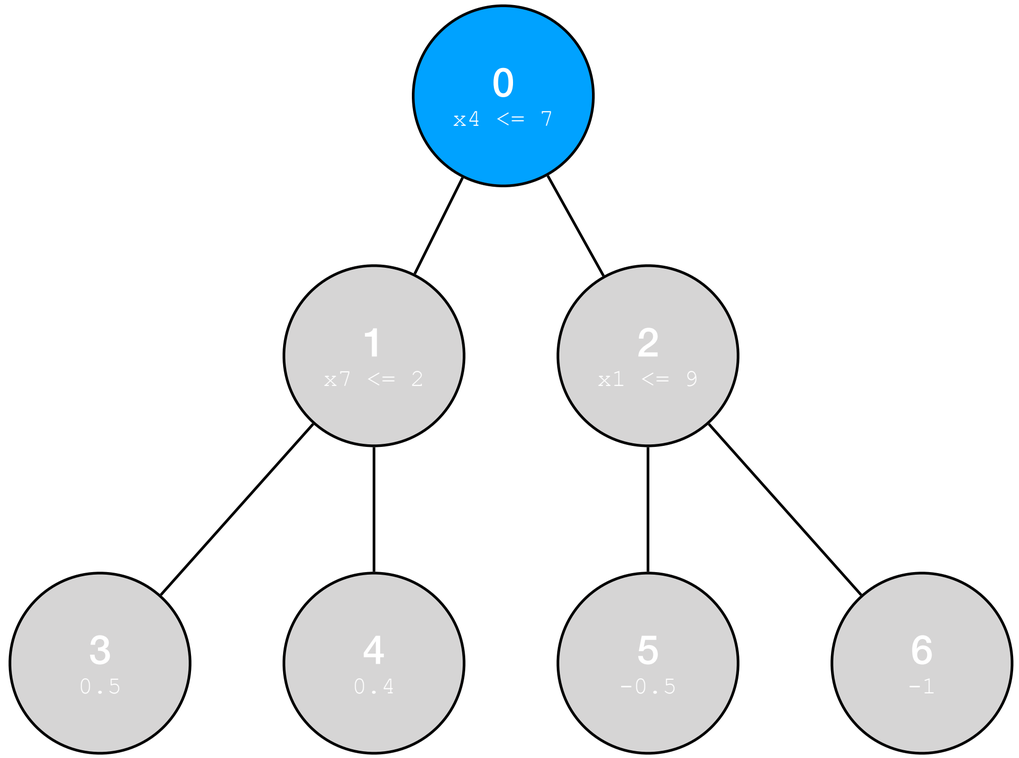
BDTs in HEP

- BDTs have been used for a long time in HEP
  - You may be familiar with ROOT's TMVA
- They have fallen in popularity compared to Neural Networks that can be extremely powerful on lower level data
- Still some papers coming out in 2023 about their use in HEP!
- They remain useful and popular for some specific reasons:
  - High level / tabular data
  - Easy to get started
  - Easy(ish) to interpret
  - Robust (against overfitting, against irrelevant features)
  - Relatively inexpensive to train and then make predictions

BDT Inference

- Start at the root node - compare the selected feature with the threshold, go left or right depending on result

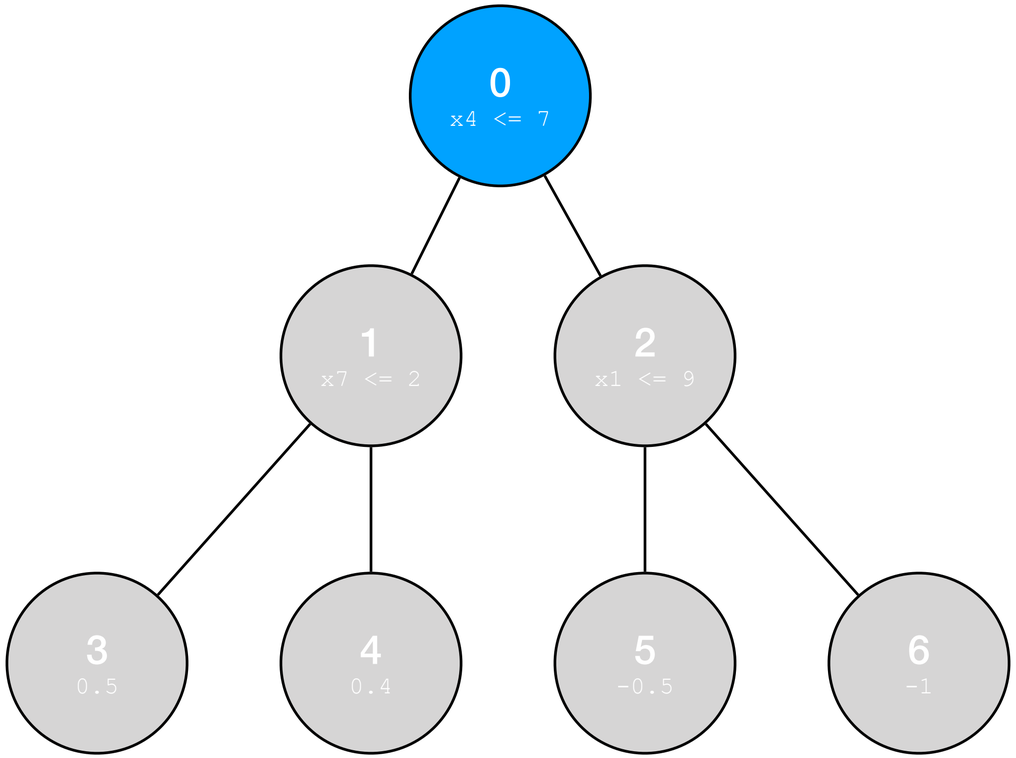
$$x = [\ x_0,\ x_1,\ x_2,\ x_3,\ x_4,\ x_5,\ x_6,\ x_7\ ]$$



BDT Inference

- Start at the root node - compare the selected feature with the threshold, go left or right depending on result

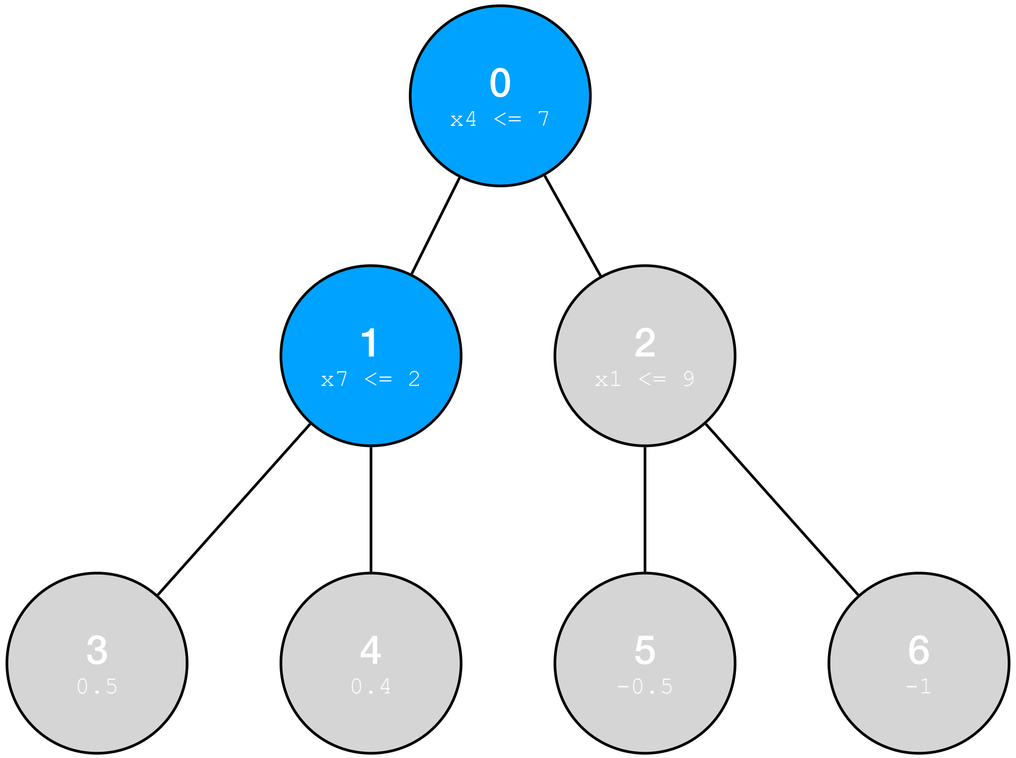
```
x = [ –, 12, –, –, 3, –, –, 5 ]
```



BDT Inference

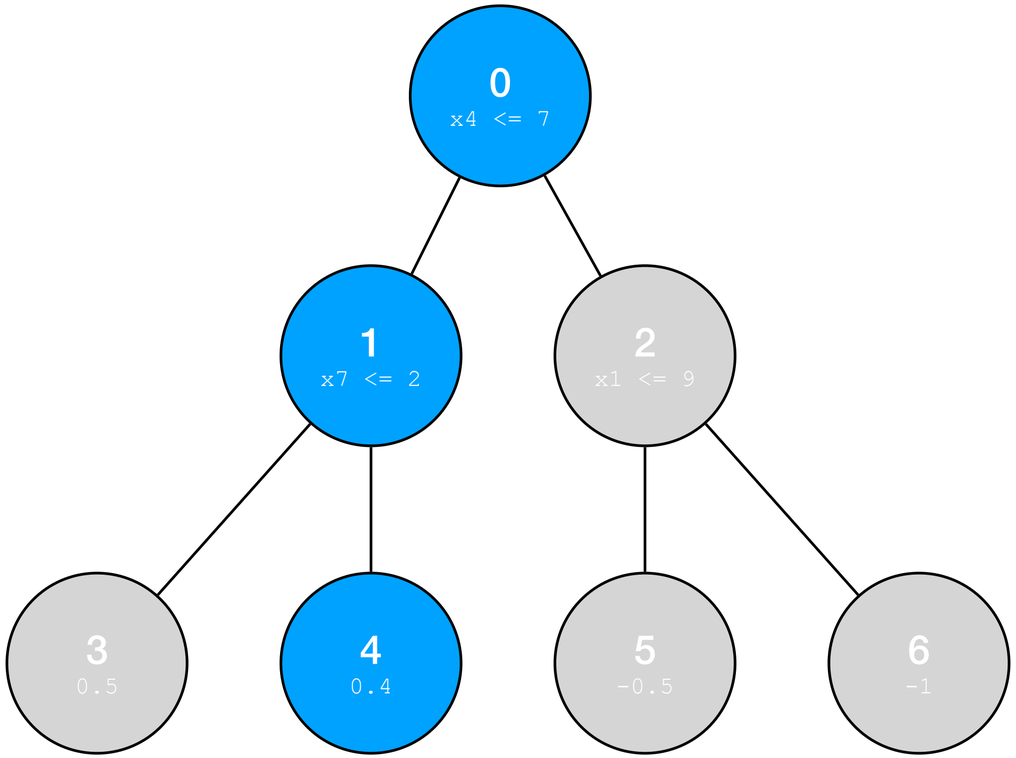
- Start at the root node - compare the selected feature with the threshold, go left or right depending on result
- Continue until reaching leaf - compare the selected feature with the threshold, go left or right depending on result

```
x = [ –, 12, –, –, 3, –, –, 5 ]
```



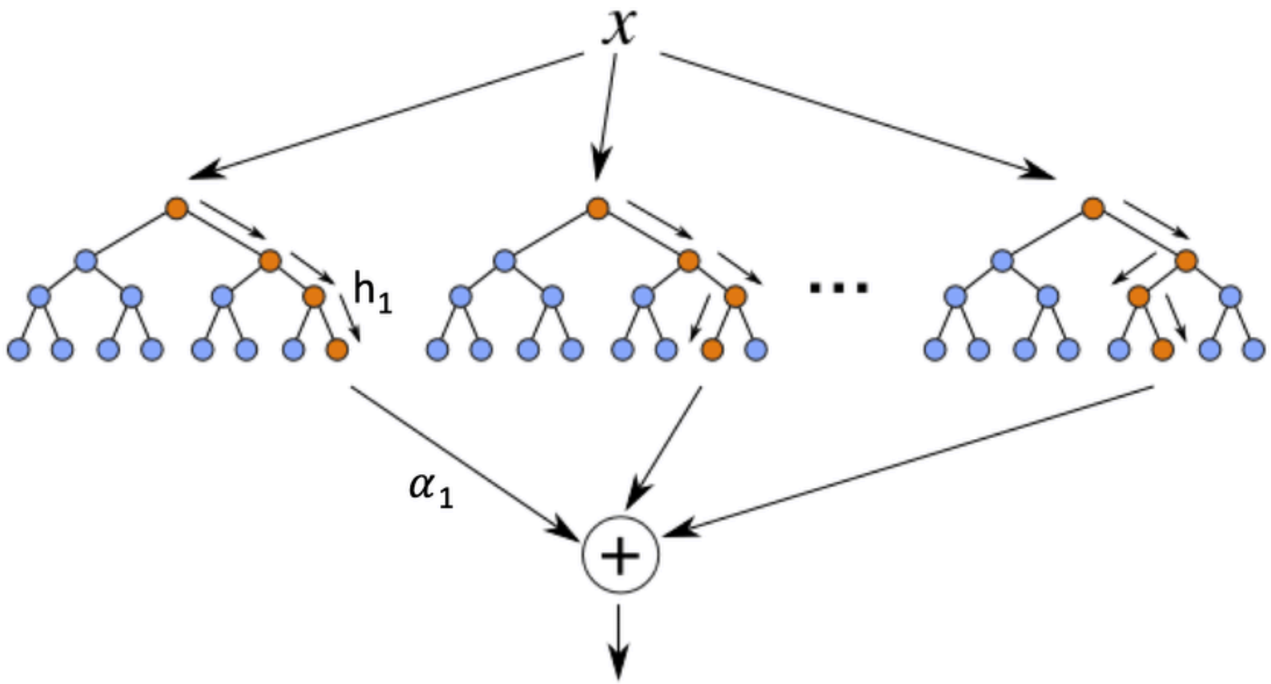
BDT Inference

- Start at the root node - compare the selected feature with the threshold, go left or right depending on result
- Continue until reaching leaf - compare the selected feature with the threshold, go left or right depending on result
- The value of the terminal leaf is the tree prediction



BDT Inference

- Repeat the same procedure for every tree in the ensemble, sum up the tree scores for the BDT prediction
- Apply the inverse of the training loss function to obtain class probabilities

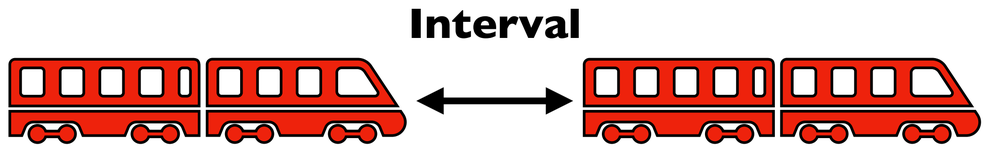


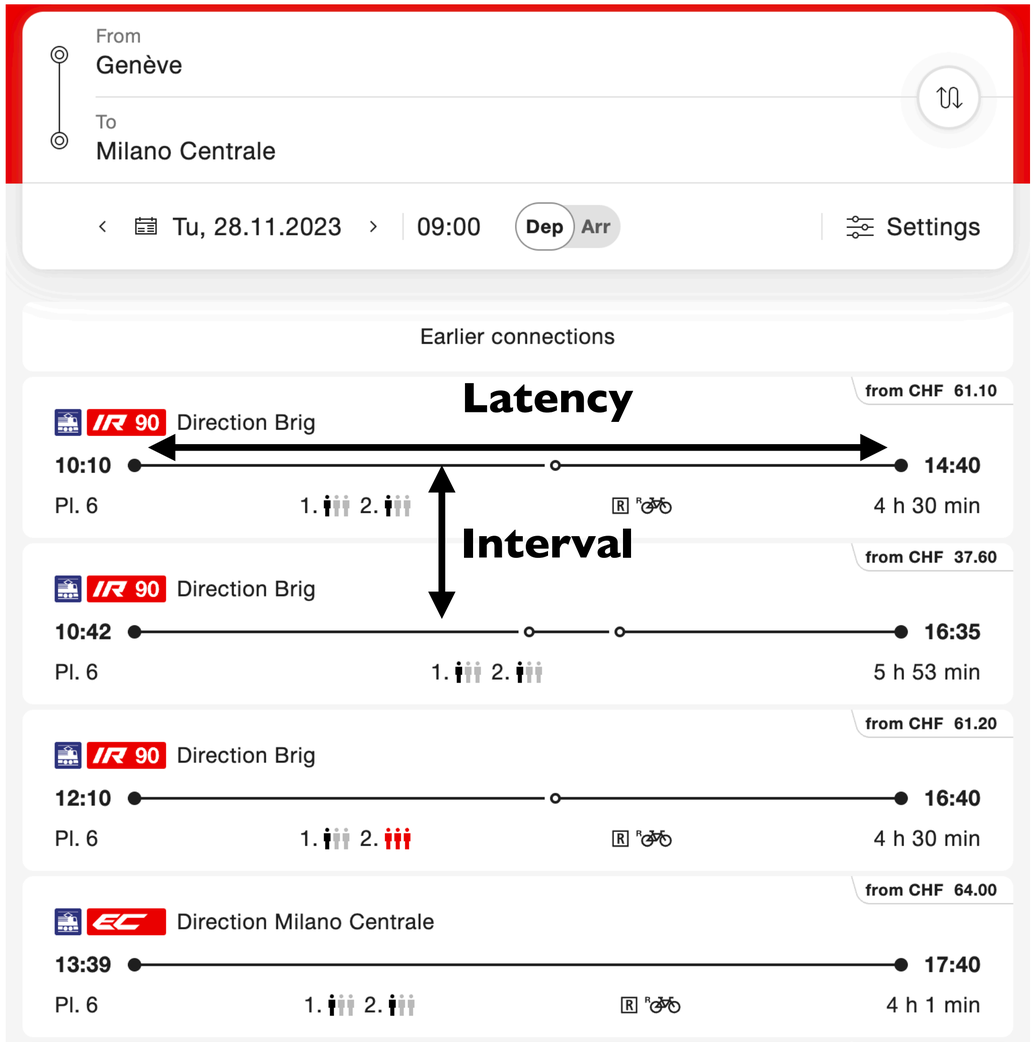
FPGAs

- Some characteristics of FPGAs to keep in mind…
- These were taught also by Giovanni, but it's okay to hear things twice
- Two types of parallelism: resource and pipeline
  - Resource parallelism enables us to do different tasks simultaneously to reach low latency
  - Pipeline parallelism enables us to do the same task on different data at high throughput
- In the automotive factory the many robots are resource parallelism and the conveyor belt is pipelining
- High performance requires use of both types

Loop Analysis

- With FPGAs we can take advantage of pipeline processing
- We need to work to keep the pipeline filled with data
- Depends on the loops of our algorithm and their inter-dependencies
- First some terminology:
  - ‘Interval’ or ‘Initiation Interval’ : gap between *start* of subsequent executions of a process
    - How often to trains depart the station?
  - ‘Latency’ : delay between start of execution of a process, and output of results
    - How long does it take to get from A to B?





Loop Analysis

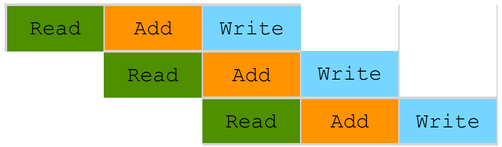
- Loops can have dependencies that impacts scheduling, unrolling, and interval
- Consider this loop executed sequentially

```
for(i = 0; i < 3; i++)
    a[i] = a[i] + 1;
```

  - The loop has Latency 3 cycles, Interval 3 cycles

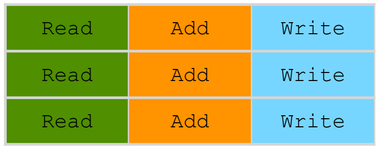
- This loop has no iteration dependence (iteration `i` does not depend on any other iteration)
  - It can be pipelined: loop has Latency 3 cycles, Interval 1 cycle

```
for(i = 0; i < 3; i++)
    a[i] = a[i] + 1;
```



- If all of `a[i]` can be read simultaneously (e.g. it's in FPGA registers not BRAMs), the loop can be *unrolled*

```
for(i = 0; i < 3; i++)
    a[i] = a[i] + 1;
```



Loop Analysis

- Some loops have dependencies (loop-carried dependence)

```
for(i = n; i > 0; i--)
    a[i] = a[i-1];
```

- We can't pipeline or unroll this loop since the `read` of iteration `i` depends on the `write` of iteration `i-1`

- For best performance with parallel architectures, we need to understand and optimise our loops
  - Defines how we can distribute loop iterations across different processing units
  - Merge loops where possible
  - Break dependencies by reordering loops

fixed-point arithmetic

- Introduced by Giovanni on Monday
- Reminder: floating-point is like scientific notation

123456

integer

1.23456×10^5

scientific notation

fixed-point arithmetic

- Introduced by Giovanni on Monday
- Reminder: floating-point is like scientific notation

0000123456

10 digit integer

$1.2345600 \times 10^{005}$

scientific notation
8 digit mantissa
3 digit exponent

0011010111

10 bit integer

$1.00110011 \times 2^{1011}$

floating point
8 bit mantissa
4 bit exponent

fixed-point arithmetic

- Introduced by Giovanni on Monday
- Reminder: floating-point is like scientific notation
- With floating point the **bitwidth** of the mantissa and exponent are *fixed*. The **value** of the mantissa and exponent can *change*
  - Have constant relative precision
- With fixed point the **bitwidth** is *fixed*. The **value** of the exponent is *fixed* (and implicit). The **value** of the mantissa can *change*
  - Have constant absolute precision

$1.00110011 \times 2^{1011}$

floating point
8 bit mantissa
4 bit exponent

01101.0011

fixed point
9 bit width
5 bit integer
(4 bit fraction)

fixed-point arithmetic

- Introduced by Giovanni on Monday
- Reminder: floating-point is like scientific notation
- With floating point the **bitwidth** of the mantissa and exponent are *fixed*. The **value** of the mantissa and exponent can *change*
- With fixed point the **bitwidth** is *fixed*. The **value** of the exponent is *fixed* (and implicit). The **value** of the mantissa can *change*

Cheap & fast arithmetic in hardware

High dynamic range
Expensive & slow in hardware

0011010111

10 bit integer

01101.0011

fixed point
9 bit width
5 bit integer
(4 bit fraction)

$1.00110011 \times 2^{1011}$

floating point
8 bit mantissa
4 bit exponent

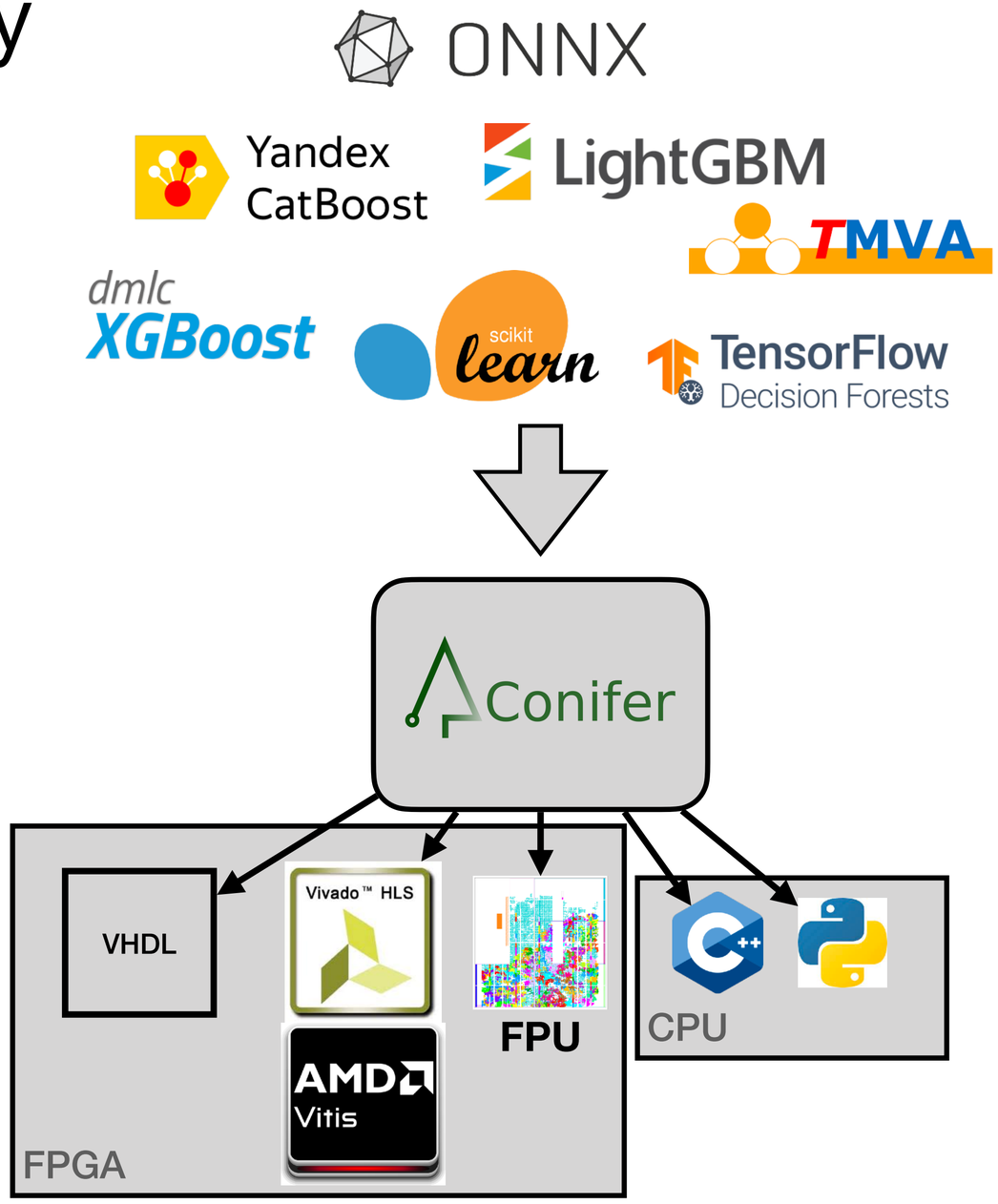
Expressiveness / Interpretability

fixed-point for BDTs

- When we train a BDT our data, thresholds, and scores have some real numerical values
- We need to choose what data type to use for our model - in FPGAs we have freedome
- We could use floating point and not waste any brain cycles
  - But it will always be more expensive than using fixed point
- Perform a numerical analysis of the model and see which range/precision is required
- The first interactive exercise exposes how to control these in conifer
- **Note**: this not only applies to BDTs!
  - Any algorithm that performs many arithmetic operations can benefit from a numerical analysis and choice of fixed-point types!

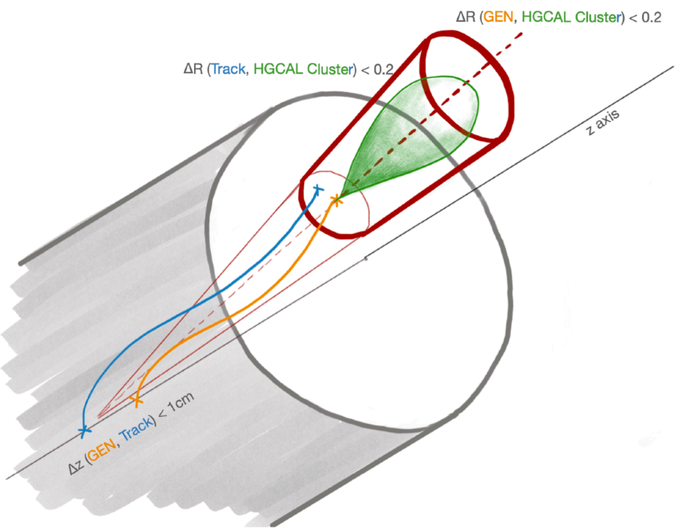
Conifer library

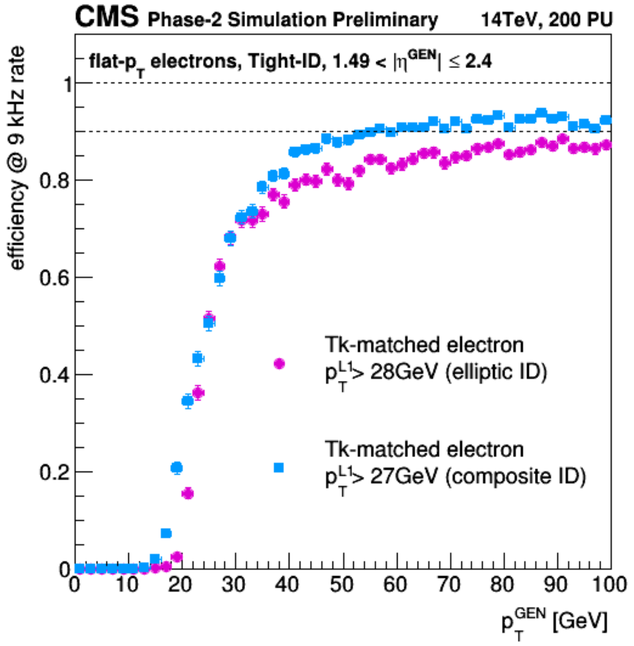
- conifer is a Python package, published to PyPI
  - `pip install conifer`
- It has a structure like a compiler
  - Converters / frontends for different BDT training libraries
  - Internal Representation
  - Backends for different compute targets
    - Three FPGA targets that we'll go through today
    - CPU targets for reference / emulation rather than high performance



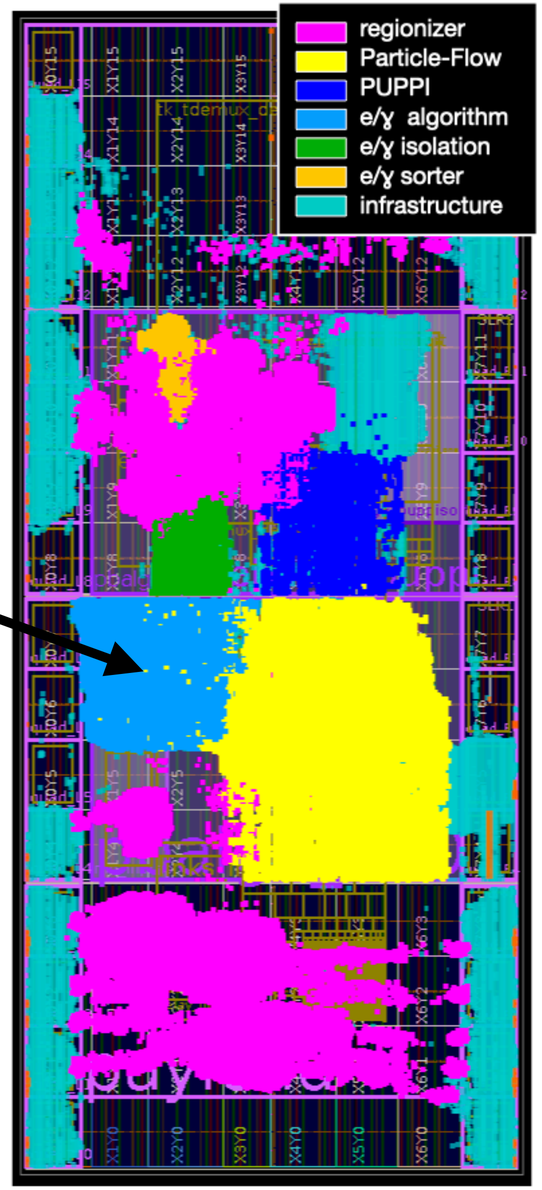
Example use case

- Improving electron reconstruction in Correlator Layer 1 of CMS Phase 2 L1T
- Predict whether a pair of {track, calorimeter cluster} are consistent with both originating from an electron
- Full e/γ algorithm has 10 BDT copies, plus other logic (e.g. dR)
  - Total consumes 3.1% VU13P LUTs, 18 cycles latency @ 180 MHz (100 ns)
- CMS-DP-2023-047



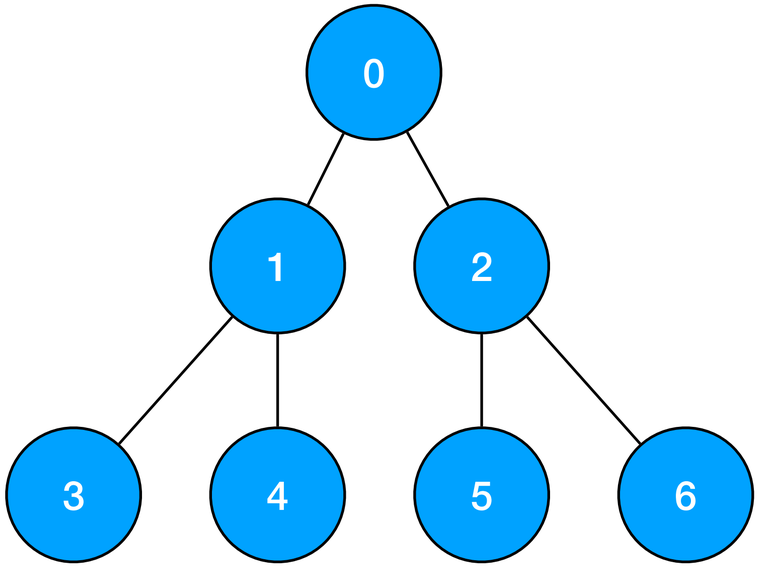


BDT gain



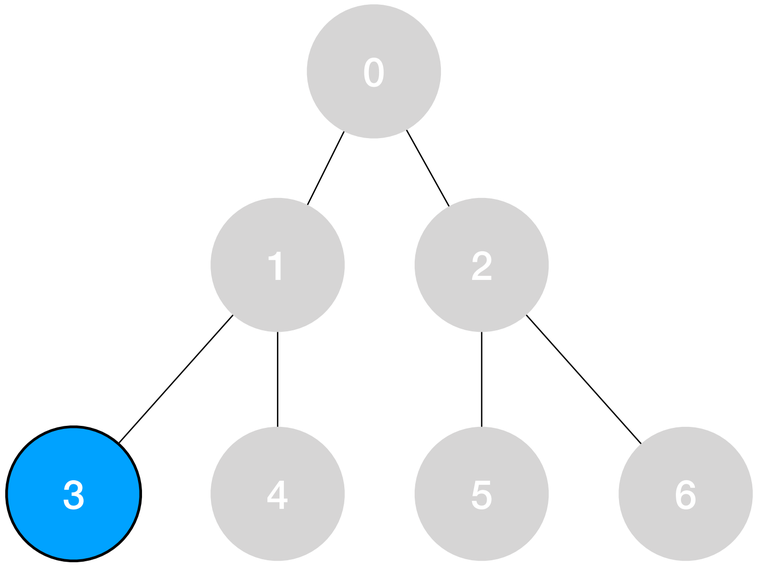
Conifer Implementation

- For a tree: find which leaf is reached given a data sample x
- ‘Invert’ the problem: for each node ask “does the decision path reach this node?” starting at the leaves



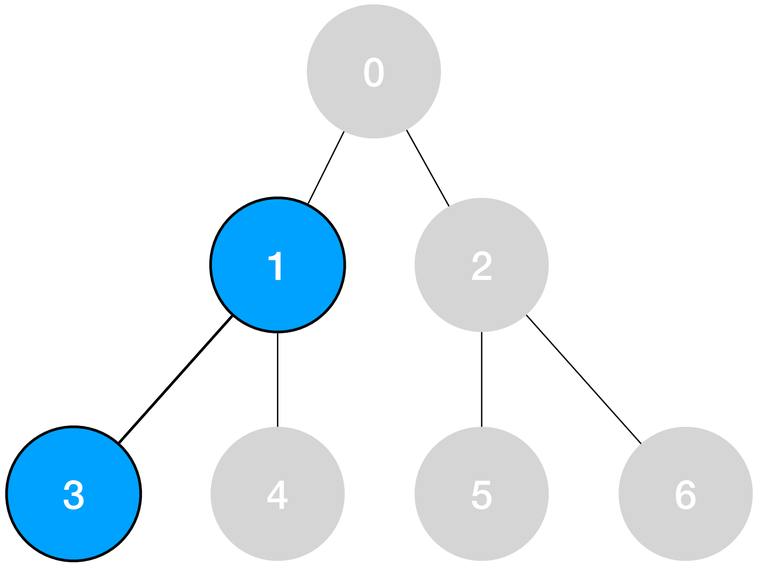
Conifer Implementation

- For a tree: find which leaf is reached given a data sample x
- ‘Invert’ the problem: for each node ask “does the decision path reach this node?” starting at the leaves
- For leaf node ‘3’:
  - The decision path reaches ‘3’ if: the decision path reached ‘1’ AND the comparison at ‘1’ goes ‘left’



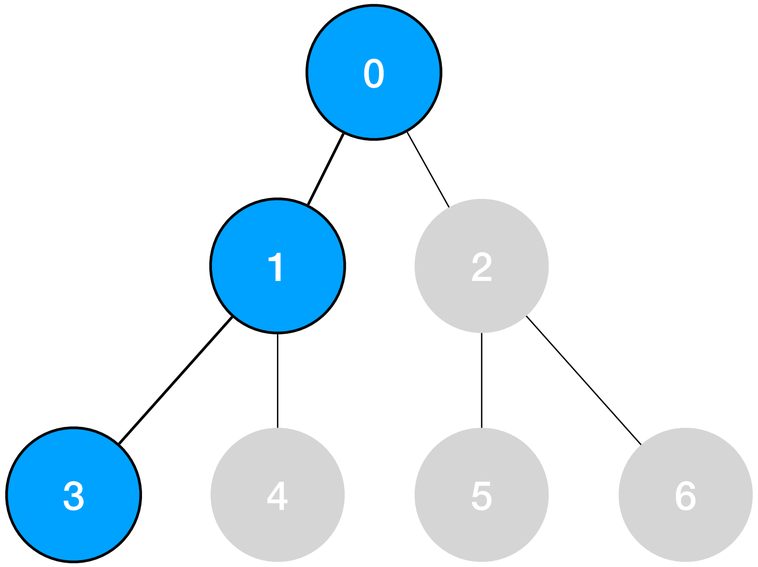
Conifer Implementation

- For a tree: find which leaf is reached given a data sample x
- ‘Invert’ the problem: for each node ask “does the decision path reach this node?” starting at the leaves
- For leaf node ‘3’:
  - The decision path reaches ‘3’ if: the decision path reached ‘1’ AND the comparison at ‘1’ goes ‘left’
- For node ‘1’:
  - The decision path reaches ‘1’ if: the decision path reached ‘0’ AND the comparison at ‘0’ goes ‘left’



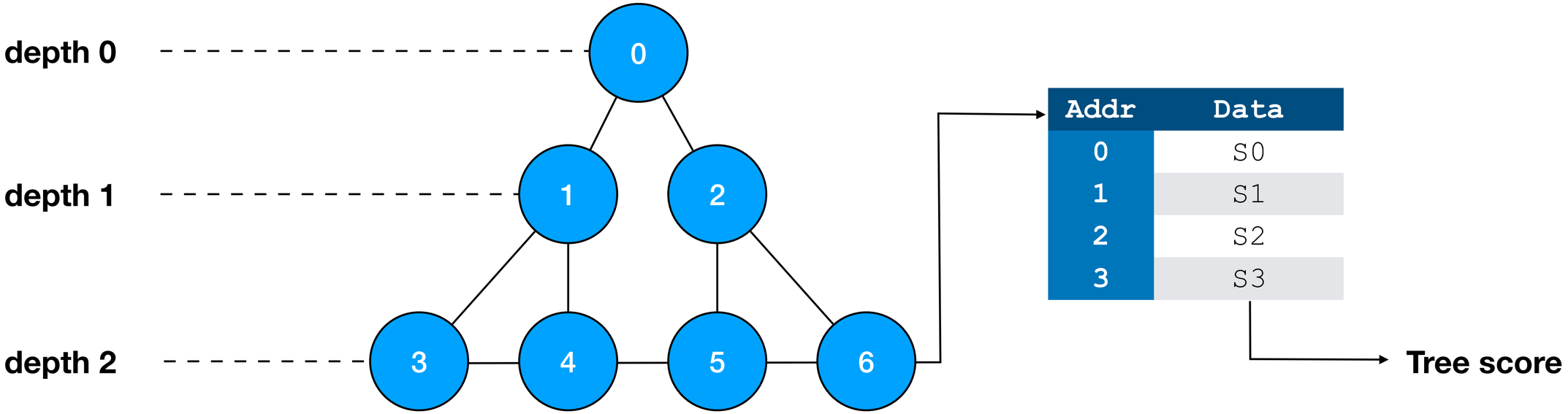
Conifer Implementation

- For a tree: find which leaf is reached given a data sample x
- ‘Invert’ the problem: for each node ask “does the decision path reach this node?” starting at the leaves
- For leaf node ‘3’:
  - The decision path reaches ‘3’ if: the decision path reached ‘1’ AND the comparison at ‘1’ goes ‘left’
- For node ‘1’:
  - The decision path reaches ‘1’ if: the decision path reached ‘0’ AND the comparison at ‘0’ goes ‘left’
- For node ‘0’:
  - The decision path always passes through the root node



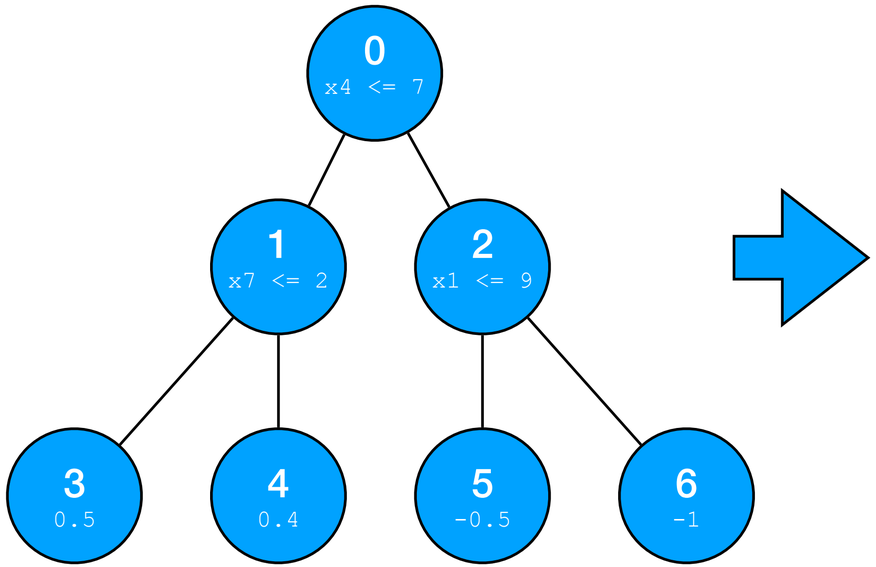
Conifer Implementation

- For a tree: find which leaf is reached given a data sample x
- ‘Invert’ the problem: for each node ask “does the decision path reach this node?” starting at the leaves
- We can **parallelise** this over paths by brute force: evaluate all nodes at the same depth simultaneously
- We can **pipeline** this over different data: each node can do a comparison on new data with II=1
- For each leaf node we have a boolean: TRUE if the decision path reaches leaf, otherwise FALSE
- Concatenate the boolean for each leaf node → select the value corresponding to the leaf



Tree Representation

- Before looking at the code, a note about data representation
- We represent trees “scikit-learn”-style ie flat
  - No representation of individual nodes
  - Each node variable (threshold, feature, value) is a tree-level array
  - A left/right child index array points to the children for each node
- Some special values: ‘-2’ typically means leaf (e.g. child index -2, feature -2)




```
Tree:
        index          : [ 0,  1,  2,   3,   4,    5,  6]
        children_left  : [ 1,  3,  5,  -2,  -2,   -2, -2]
        children_right : [ 2,  4,  6,  -2,  -2,   -2, -2]
        parent         : [ -1, 0,  0,   1,   1,    2,  2]
        feature        : [ 4,  7,  1,  -2,  -2,   -2, -2]
        threshold      : [ 7,  2,  9,  -2,  -2,   -2, -2]
        value          : [-1, -1, -1, 0.5, 0.4, -0.5, -1]
```

HLS Code 1

- Perform all the comparisons simultaneously: `unroll` the loop
- Store `boolean` results in a fully-partitioned array "`comparison`"

```
// Execute all comparisons
Compare: for(int i = 0; i < n_nodes; i++){
  #pragma HLS unroll
  // Only non-leaf nodes do comparisons
  // negative values mean is a leaf (sklearn: -2)
  if(feature[i] >= 0){
    comparison[i] = x[feature[i]] <= threshold[i];
  }else{
    comparison[i] = true;
  }
}
```

HLS Code 2

- Compute the node `activation` (`true` if decision path traverses node, otherwise `false`)

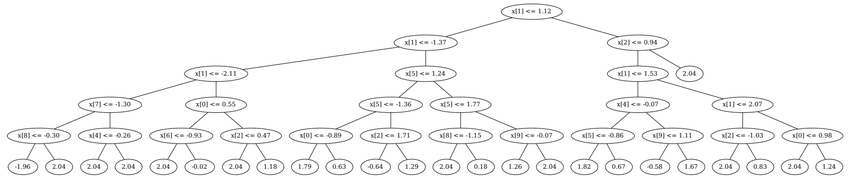
```
// Determine node activity for all nodes
int iLeaf = 0;
Activate: for(int i = 0; i < n_nodes; i++){
  #pragma HLS unroll
  // Root node is always active
  if(i == 0){
    activation[i] = true;
  }else{
    // If this node is the left child of its parent
    if(i == children_left[parent[i]]){
      activation[i] = comparison[parent[i]] && activation[parent[i]];
    }else{ // Else it is the right child
      activation[i] = !comparison[parent[i]] && activation[parent[i]];
    }
  }
  // Skim off the leaves
  if(children_left[i] == -1){ // is a leaf
    activation_leaf[iLeaf] = activation[i];
    value_leaf[iLeaf] = value[i];
    iLeaf++;
  }
}
```

HLS Code

- Compute the node `activation` (`true` if decision path traverses node, otherwise `false`)

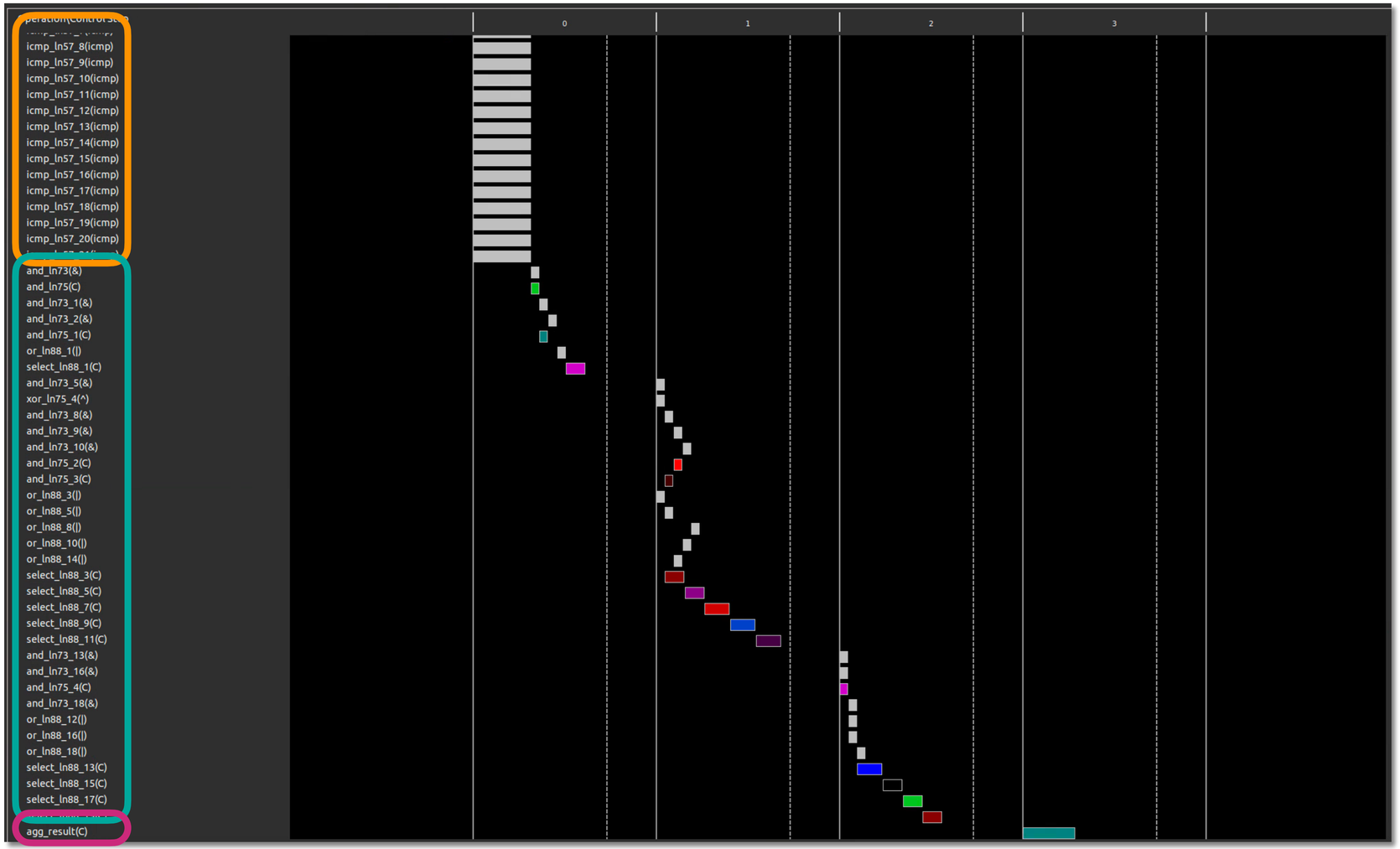
```
for(int i = 0; i < n_leaves; i++){
   if(activation_leaf[i]){
     return value_leaf[i];
   }
 }
```

Scheduling - Tree



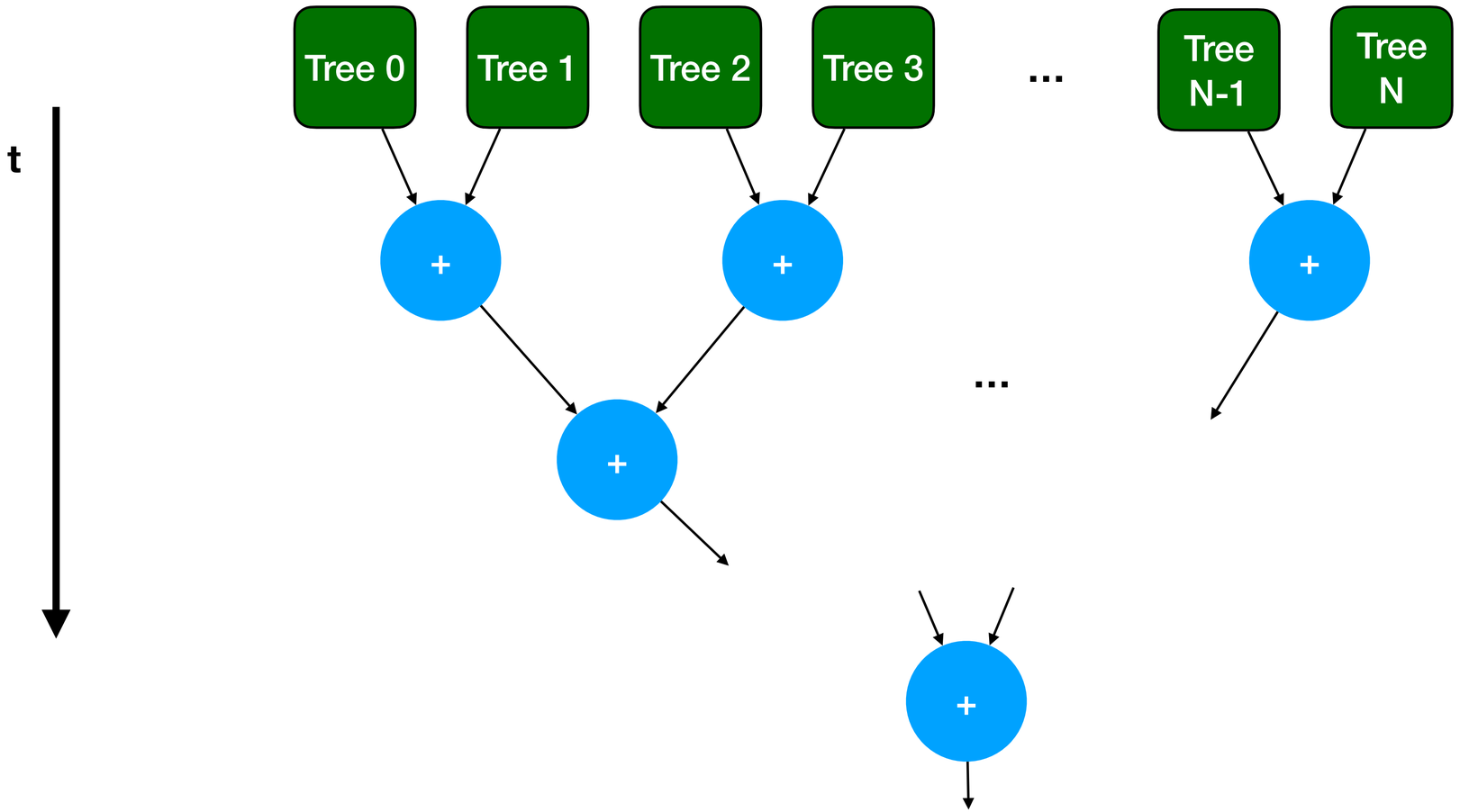
- Did we achieve what we described?
- Vitis HLS Schedule Viewer in GUI
  - Tree depth = 5, some sparsity
- All **comparisons** in parallel at the start
- Cascade of **boolean operations**
  - AND, OR, XOR, NOT
- '**Aggregate**' at end

t (clock cycles) →



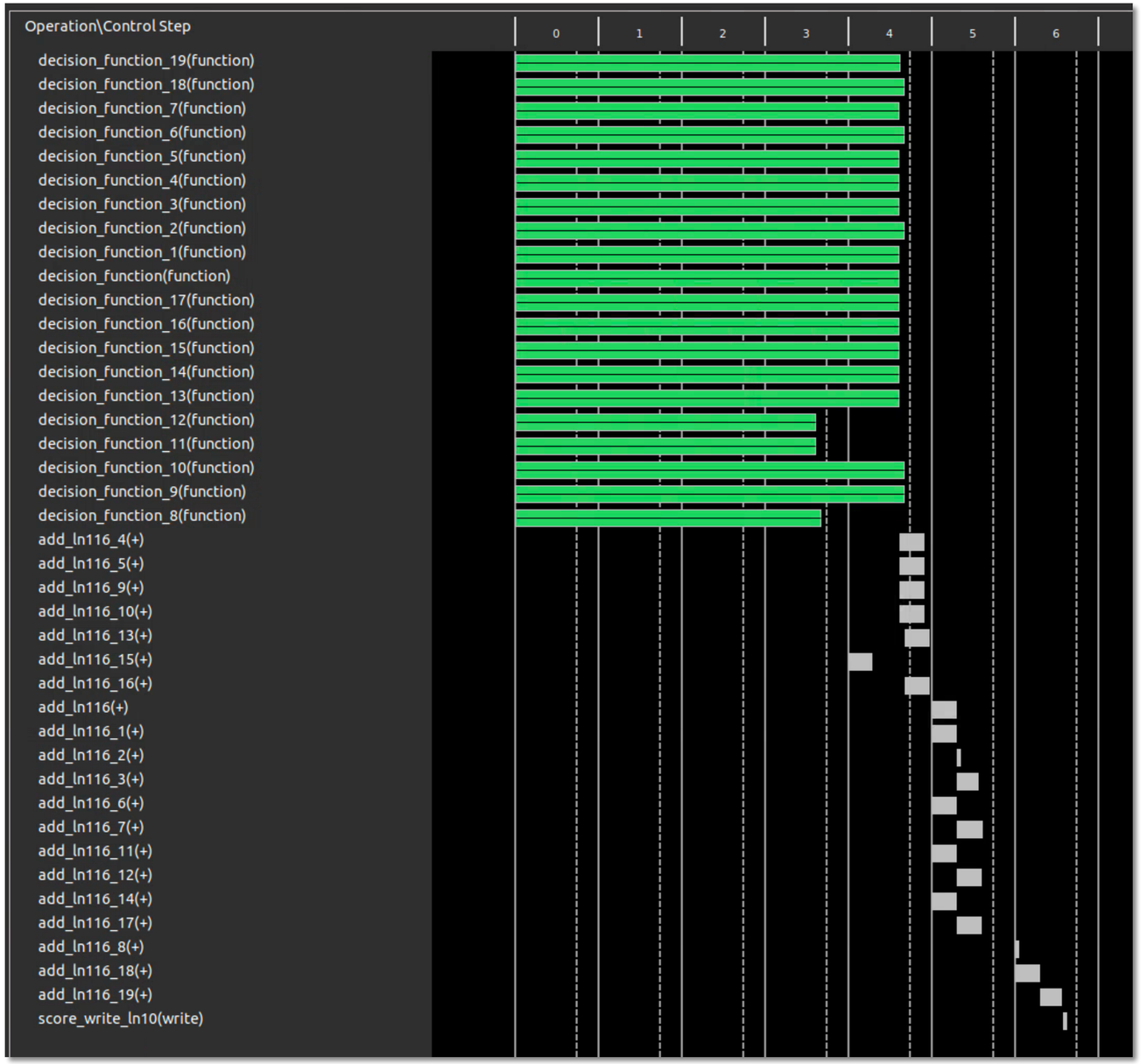
Conifer Implementation

- For a forest: aggregate over all trees - normally summation, but can be other e.g. some quantile
- A parallel addition also uses a kind of tree: adder tree (like “pairwise reduce”)



Scheduling - Forest

- Did we achieve what we described?
- Vitis HLS Schedule Viewer in GUI
  - Number of trees = 20
  - Tree from previous slides is one of them
- All tree inferences performed in parallel
- Tree scores summed in pairs
- Total latency: 7 clock cycles



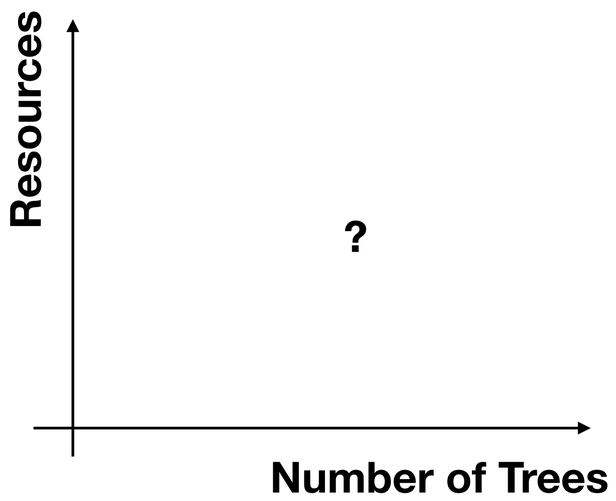
Resource Usage - Trees

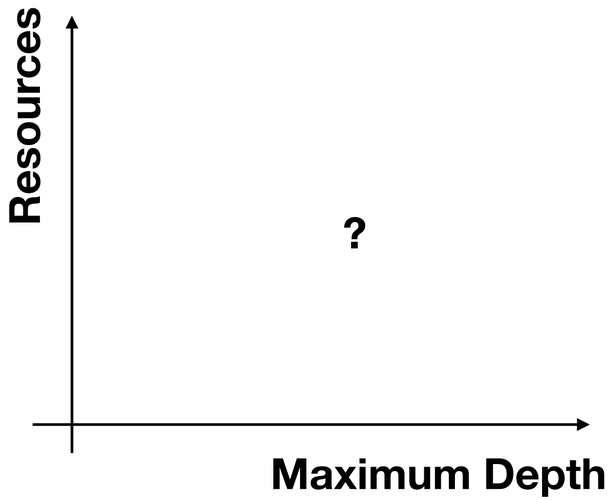
- Each tree uses independent logic
- Resource usage depends on structure of the tree
- Since thresholds are ‘baked in’ to comparisons, resource usage of each ‘>=‘ can depend on the threshold value as well
- Can see up to factor 2 difference in LUT usage of different trees
- Normally FFs would also be used but this model must be too trivial…

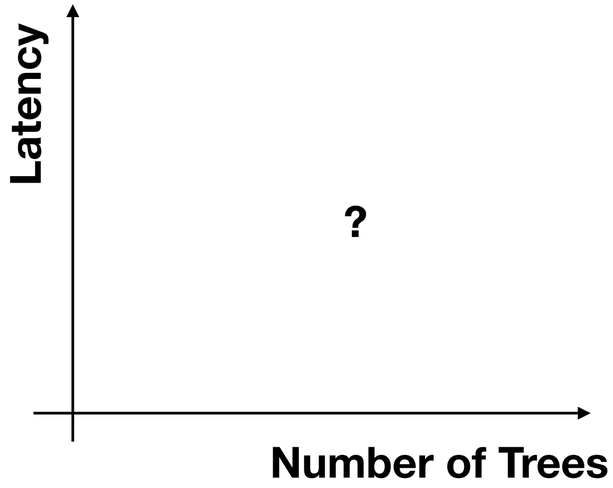
```
+ Detail:
    * Instance:
    +---------------------------------------+----------------------+---------+----+---+-----+-----+
    |                Instance               |        Module        | BRAM_18K| DSP| FF| LUT | URAM|
    +---------------------------------------+----------------------+---------+----+---+-----+-----+
    |scores_9_decision_function_fu_161      |decision_function     |        0|   0|  0|  206|    0|
    |scores_8_decision_function_1_fu_153    |decision_function_1   |        0|   0|  0|  246|    0|
    |scores_16_decision_function_10_fu_223  |decision_function_10  |        0|   0|  0|  278|    0|
    |scores_15_decision_function_11_fu_215  |decision_function_11  |        0|   0|  0|  268|    0|
    |scores_14_decision_function_12_fu_209  |decision_function_12  |        0|   0|  0|  167|    0|
    |scores_13_decision_function_13_fu_201  |decision_function_13  |        0|   0|  0|  268|    0|
    |scores_12_decision_function_14_fu_193  |decision_function_14  |        0|   0|  0|  247|    0|
    |scores_11_decision_function_15_fu_185  |decision_function_15  |        0|   0|  0|  282|    0|
    |scores_10_decision_function_16_fu_177  |decision_function_16  |        0|   0|  0|  208|    0|
    |scores_19_decision_function_17_fu_169  |decision_function_17  |        0|   0|  0|  153|    0|
    |scores_1_decision_function_18_fu_97    |decision_function_18  |        0|   0|  0|  232|    0|
    |scores_decision_function_19_fu_89      |decision_function_19  |        0|   0|  0|  234|    0|
    |scores_7_decision_function_2_fu_145    |decision_function_2   |        0|   0|  0|  248|    0|
    |scores_6_decision_function_3_fu_137    |decision_function_3   |        0|   0|  0|  278|    0|
    |scores_5_decision_function_4_fu_129    |decision_function_4   |        0|   0|  0|  190|    0|
    |scores_4_decision_function_5_fu_121    |decision_function_5   |        0|   0|  0|  243|    0|
    |scores_3_decision_function_6_fu_113    |decision_function_6   |        0|   0|  0|  232|    0|
    |scores_2_decision_function_7_fu_105    |decision_function_7   |        0|   0|  0|  230|    0|
    |scores_18_decision_function_8_fu_235   |decision_function_8   |        0|   0|  0|  208|    0|
    |scores_17_decision_function_9_fu_229   |decision_function_9   |        0|   0|  0|  248|    0|
    +---------------------------------------+----------------------+---------+----+---+-----+-----+
    |Total                                  |                      |        0|   0|  0| 4666|    0|
    +---------------------------------------+----------------------+---------+----+---+-----+-----+
```

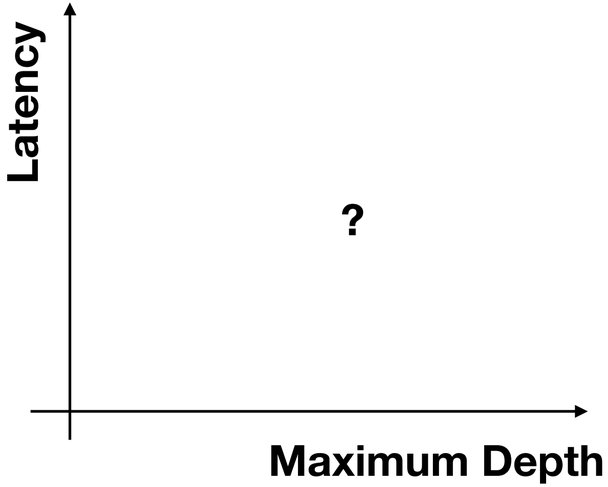
Exercise 1 - mini quiz

- Given what we now know about the implementation, how do you expect the resources and latency to vary with number of trees and depth?





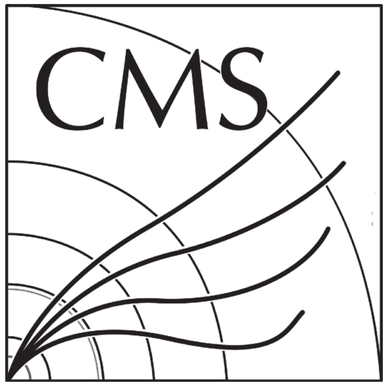




Section 1 summary

- We've discussed Boosted Decision Trees (BDTs) and how they work algorithmically
- We've discussed FPGAs and their features that make them suitable for high performance computation
- We've discussed the conifer library for BDTs in FPGAs
  - How the inference algorithm is designed for low latency and high throughput - 'inverting the problem'
  - We looked at how that's written in HLS
  - We looked at how that HLS synthesizes to the intended design
- Some useful guiding principles:
  - Think about how the problem should map onto parallel and pipelined logic *before* writing code
    - That said sometimes with HLS it's easier to just write the code and see what happens
  - Think 'branchless': the logic is always doing something, but sometimes you don't use its result
- Next we will get a first hands on with conifer

Break

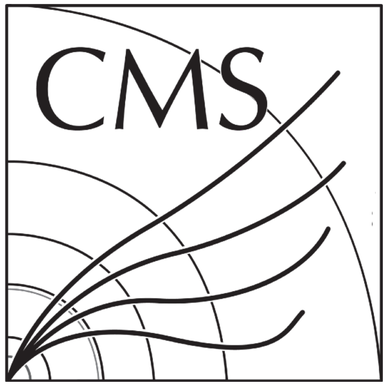


Exercise 1

- Conifer conversion and HLS walkthrough
- Clone the GitHub repository and work through notebook part 1
  - `git clone https://github.com/thesps/conifer-tutorial`
    - If you go through it fast, try changing things like training a model with a different size (number of trees, maximum depth)
- Return for a summary…

FPGAs, HLS, and Boosted Decision Trees with

Section 2



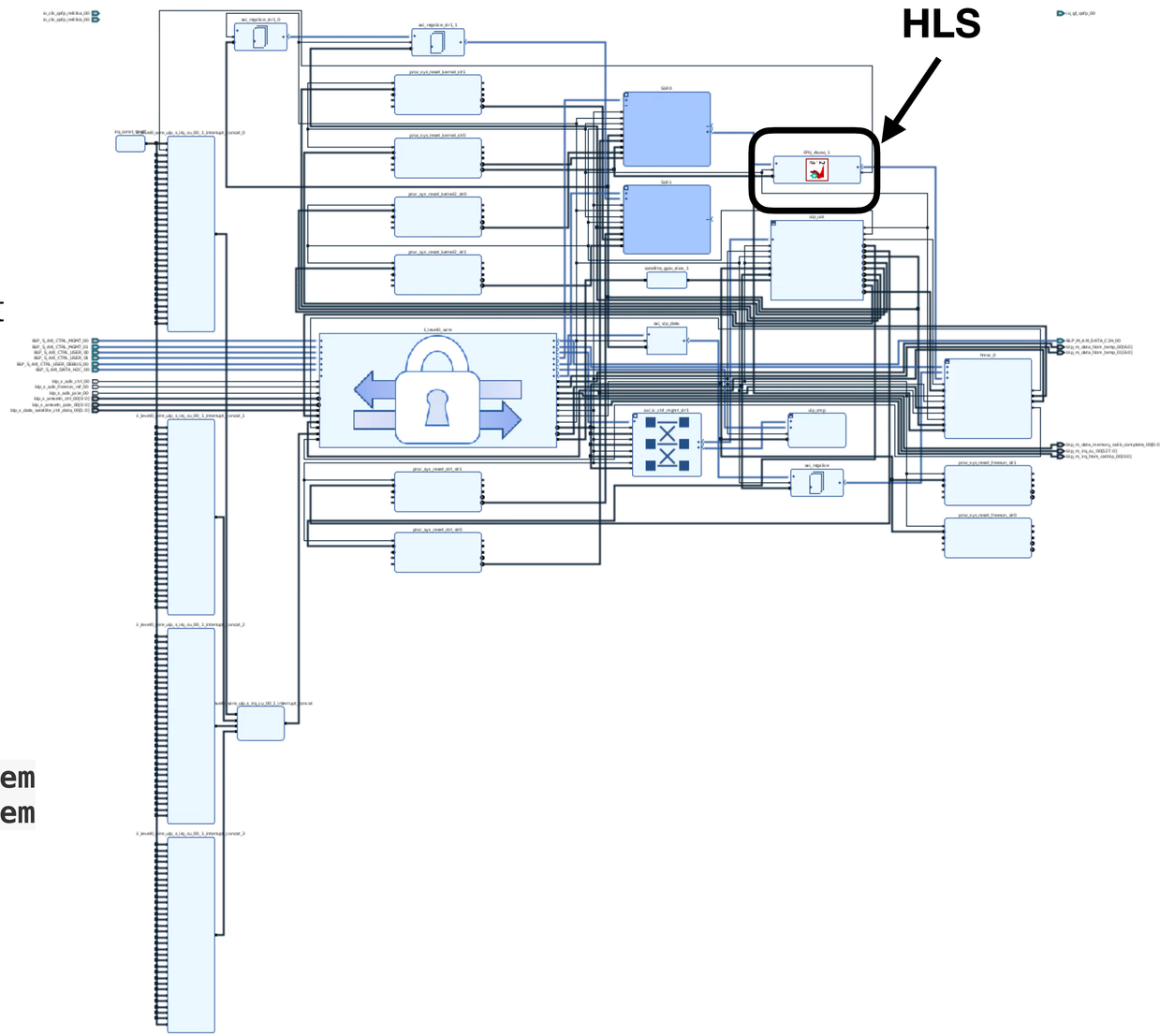
VHDL

- conifer also has a hand-written VHDL implementation
- We won't use it extensively today but it can be interesting to compare side-by-side the HLS with the VHDL
- Notebook part 1b will walk you through it
- To the right is the VHDL version of the tree traversal that we previously saw in HLS
- The main difference is that we have to do the scheduling of operations to clock cycles ourselves in VHDL
  - Each 'if rising_edge(clk) then' registers a signal
  - It can be very unintuitive - the latency of this section of code depends on the maximum depth of the tree

```
activation(0) <= true; -- the root node is always active
GenAct:
for i in 1 to nNodes-1 generate
  LeftChild:
  if i = iChildLeft(iParent(i)) generate
    process(clk)
    begin
      if rising_edge(clk) then
        activation(i) <= comparisonPipe(depth(i))(iParent(i))
                         and activation(iParent(i));
      end if;
    end process;
  end generate LeftChild;
  RightChild:
  if i = iChildRight(iParent(i)) generate
    process(clk)
    begin
      if rising_edge(clk) then
        activation(i) <= (not comparisonPipe(depth(i))(iParent(i)))
                         and activation(iParent(i));
      end if;
    end process;
  end generate RightChild;
end generate GenAct;
```

Building accelerators

- We can target Xilinx FPGAs as accelerators using Vitis
- We need to add interfaces for the data I/O
- Preferred way to do this is with AXI
  - Then we can transfer data via Direct Memory Access (DMA) host → card → host
- After synthesizing our block with Vitis HLS, we run Vitis by invoking **v++** to 'link' our design
- Under-the-hood it runs Vivado for full Place and Route

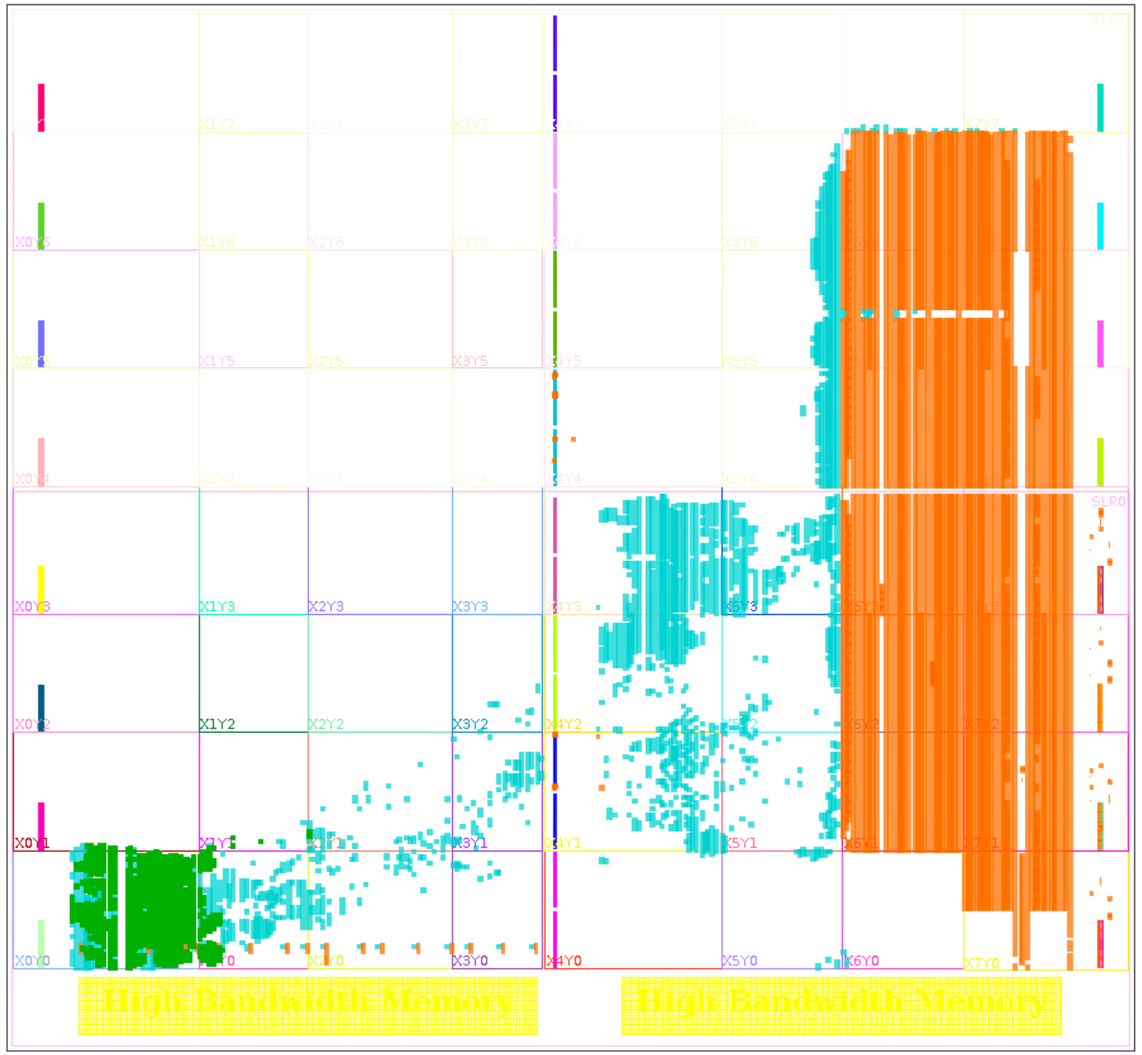



```
void times_2(int N, int* x, int* y){
  #pragma hls interface mode=m_axi port=x offset=slave bundle=gmem
  #pragma hls interface mode=m_axi port=y offset=slave bundle=gmem

  for(int i = 0; i < N; i++){
    #pragma hls pipeline
    y[i] = x[i] * 2;
  }
}
```

Building accelerators

- We can target Xilinx FPGAs as accelerators using Vitis
- We need to add interfaces for the data I/O
- Preferred way to do this is with AXI
  - Then we can transfer data via Direct Memory Access (DMA) host → card → host
- After synthesizing our block with Vitis HLS, we run Vitis by invoking **v++** to 'link' our design
- Under-the-hood it runs Vivado for full Place and Route




```
void times_2(int N, int* x, int* y){
  #pragma hls interface mode=m_axi port=x offset=slave bundle=gmem
  #pragma hls interface mode=m_axi port=y offset=slave bundle=gmem

  for(int i = 0; i < N; i++){
    #pragma hls pipeline
    y[i] = x[i] * 2;
  }
}
```

Building accelerators

- In conifer we add a section to the configuration to specify the target FPGA and some settings
  - An important one is the data type of the data on the bus
  - The default is `float`, cast to `ap_fixed` in FPGA
- conifer adds some HLS dressing to read/write data and execute inference in a variable bound loop (like Alveo example)

```
void copy_input(int n, accelerator_input_t* x_in, input_arr_t x_int){
  for(int i = 0; i < n_features; i++){
    x_int[i] = x_in[n_features*n + i];
  }
}

void copy_output(int n, score_arr_t score_int, accelerator_output_t* score_ou
  for(int i = 0; i < BDT::fn_classes(n_classes); i++){
    score_out[BDT::fn_classes(n_classes)*n + i] = score_int[i];
  }
}

void myproject_accelerator(int N, int& n_f, int& n_c, accelerator_input_t* x,
  #pragma HLS interface mode=m_axi port=x offset=slave bundle=gmem0
  #pragma HLS interface mode=m_axi port=score offset=slave bundle=gmem0
  #pragma HLS interface mode=s_axilite port=N
  #pragma HLS interface mode=s_axilite port=n_f
  #pragma HLS interface mode=s_axilite port=n_c
  n_f = n_features;
  n_c = BDT::fn_classes(n_classes);
  for(int n = 0; n < N; n++){
    #pragma HLS pipeline
    input_arr_t x_int;
    score_arr_t score_int;
    copy_input(n, x, x_int);
    bdt.decision_function(x_int, score_int);
    copy_output(n, score_int, score);
  }
}
```

Forest Processing Unit

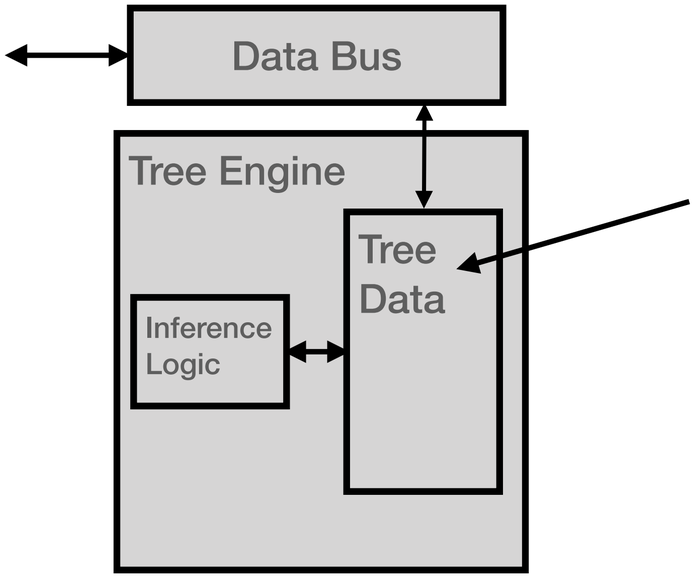
- So far we looked at ‘static’ BDT evaluation
  - One trained model → one HLS function → one IP → one bitfile
  - So if the model changes at all, we need to redo everything → takes hours!
- In next section we will look at a more dynamic & reconfigurable implementation called “Forest Processing Unit” (FPU)
- It’s still implemented with HLS, so will be a first look at going away from fixed-latency, fixed-function types of designs using HLS

FPU Design

- We would like a base design that can perform inference of ~any BDT model afterwards (within some limits)
- And we would like to take advantage of the FPGA to get good performance (fast inference)
- **Idea 1**: represent the BDT as data, operate inference on that data, and load new data for a new model
- **Idea 2**: parallelise over trees by having independent ‘Tree Engines’, aggregate their output for the model

FPU Design

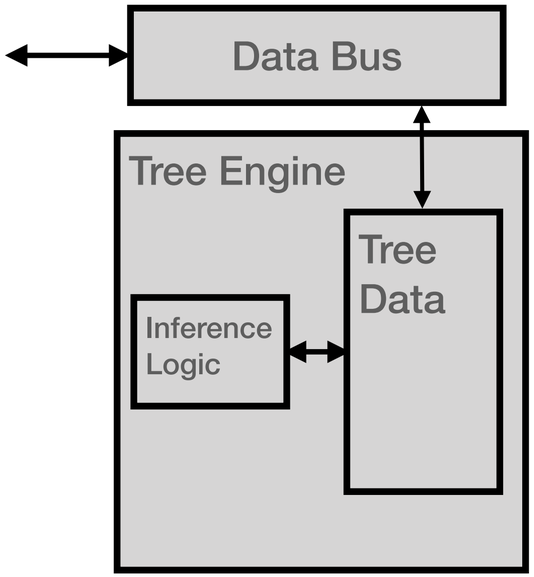
- **Idea 1**: represent the BDT as data, operate inference on that data, and load new data for a new model
- Use a data representation like we already used, and map to BRAMs
  - Many independent small memories
- Store one node at one address, child indices are pointers to other addresses




```
Tree:
        index          : [ 0,  1,  2,   3,   4,    5,  6]
        children_left  : [ 1,  3,  5,  -2,  -2,   -2, -2]
        children_right : [ 2,  4,  6,  -2,  -2,   -2, -2]
        parent         : [ -1, 0,  0,   1,   1,    2,  2]
        feature        : [ 4,  7,  1,  -2,  -2,   -2, -2]
        threshold      : [ 7,  2,  9,  -2,  -2,   -2, -2]
        value          : [-1, -1, -1, 0.5, 0.4, -0.5, -1]
```

FPU Design

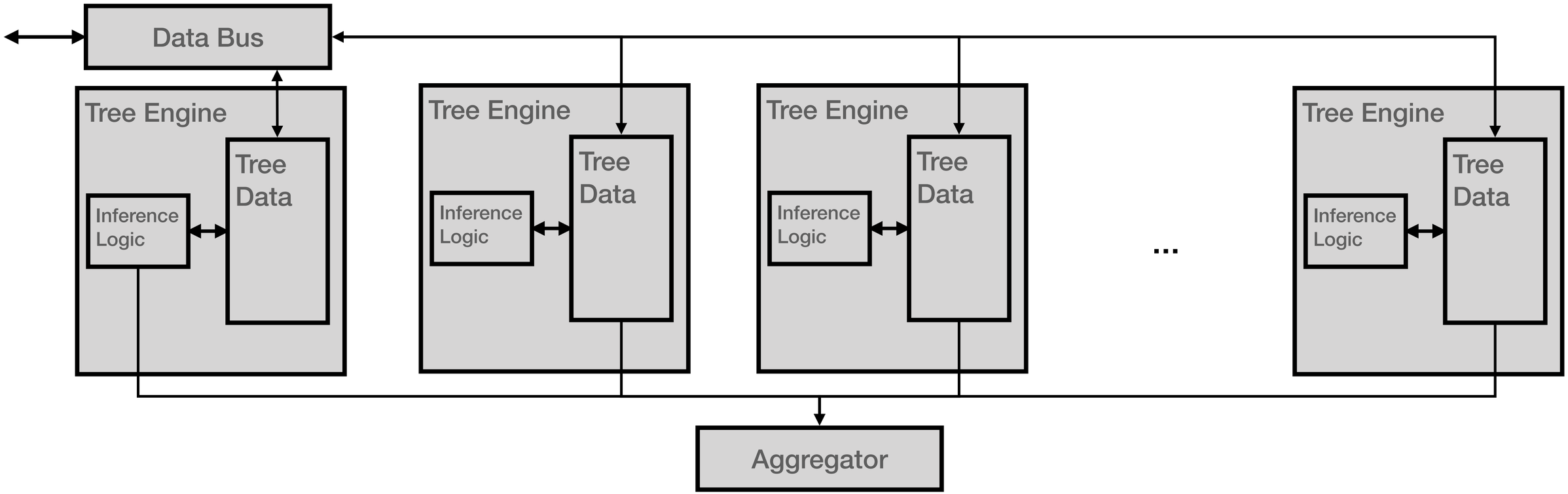
- **Idea 1**: represent the BDT as data, operate inference on that data, and load new data for a new model
- To perform inference of a model on some data we need to:
  - Read the next node
  - Compare the appropriate feature with the threshold
  - Get the pointer to the next node
- Question: what would be the pipeline initiation interval of this loop?




```
void TreeEngine(T X[NVARS], DecisionNode nodes[NNODES], U& y){
  #pragma HLS pipeline
  ap_int<ADDRBITS> i = 0;
  auto node = nodes[i];
  node_loop : while(!node.is_leaf){
    #pragma HLS pipeline
    i = X[node.feature] <= node.threshold ?
                                node.child_left : node.child_right;
    node = nodes[i];
  }
  y = node.score;
}
```

FPU Design

- **Idea 2**: parallelise over trees by having independent ‘Tree Engines’, aggregate their output for the model
- Put as many Tree Engines as will fit in the FPGA
- Number of Tree Engines will constrain the model size that fits



FPU Design

- **Idea 2**: parallelise over trees by having independent ‘Tree Engines’, aggregate their output for the model
- Put as many Tree Engines as will fit in the FPGA
- Number of Tree Engines will constrain the model size that fits

```
U y_acc = 0;
for(int i = 0; i < NTE; i++){
 #pragma HLS unroll
  U y_i = 0;
  TreeEngine(X, nodes[i], y_i);
  y_acc += y_i;
}
```

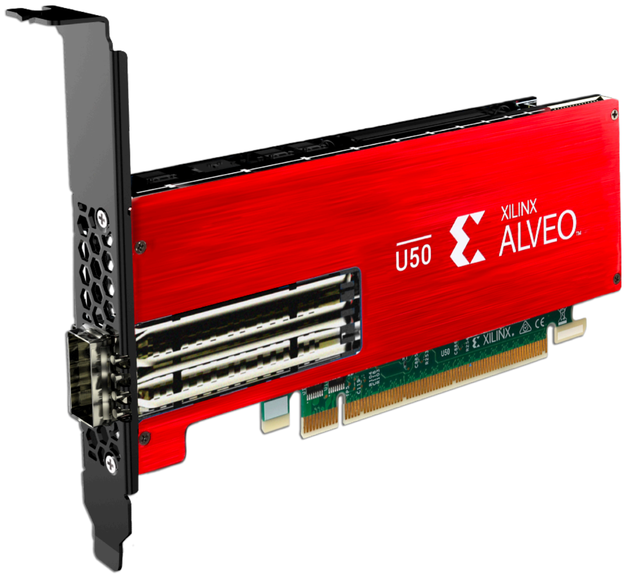
FPU System Design

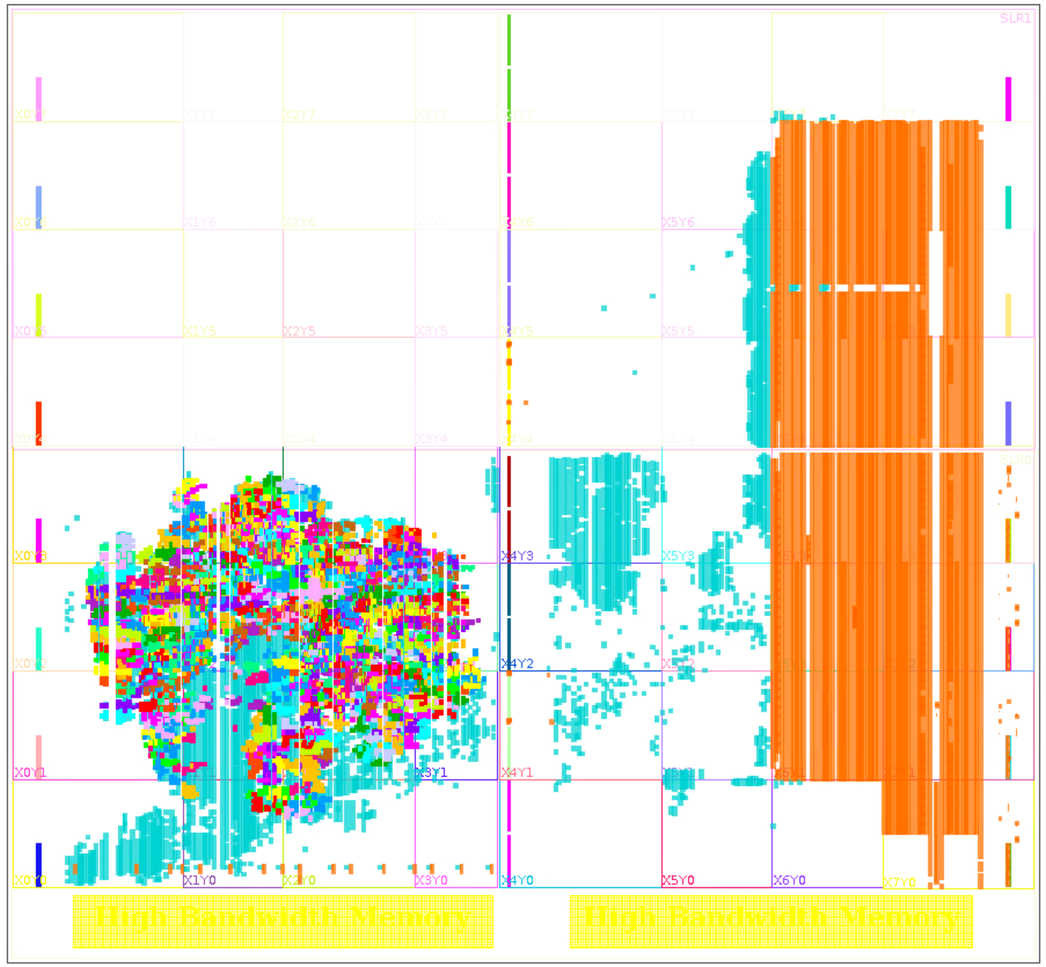
- Putting it together
  - One function that has arguments for both BDT-data and inference-data, and an ‘instruction’ parameter for what to do
- Define the node memories as static to keep the data in between function calls
  - Load nodes once, perform inference later whenever (multiple times)
  - Later load new nodes for a different model..
- This code is a simplified view of that:

```
void fpu_top_level(int* X, int* y, int instruction, DecisionNode* nodes){
  #pragma interface …
  static DecisionNode nodes_internal[NTE][NNODES];
  #pragma HLS array_partition variable=nodes_int dim=1
  if(instruction == 0){
    load_nodes(nodes, nodes_internal);
  }
  if(instruction == 1){
    decision_function(X, y);
  }
}
```

FPU Floorplan

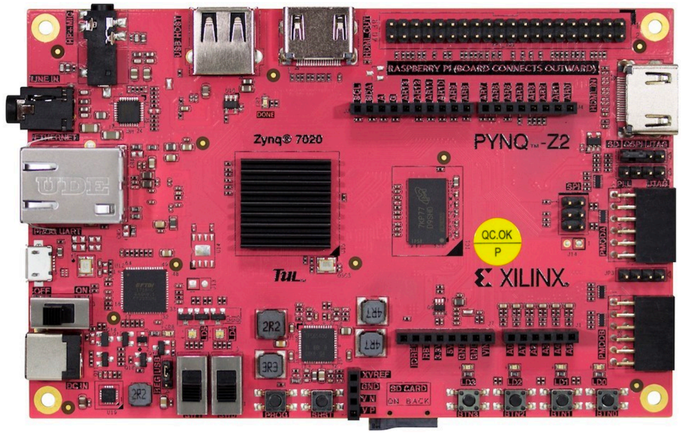
- FPU with 200 Tree Engines in Alveo U50
  - Each TE is highlighted in colour (with a repeating cycle)
- BRAMs for nodes are in columns
- Logic near BRAMs is TE inference logic

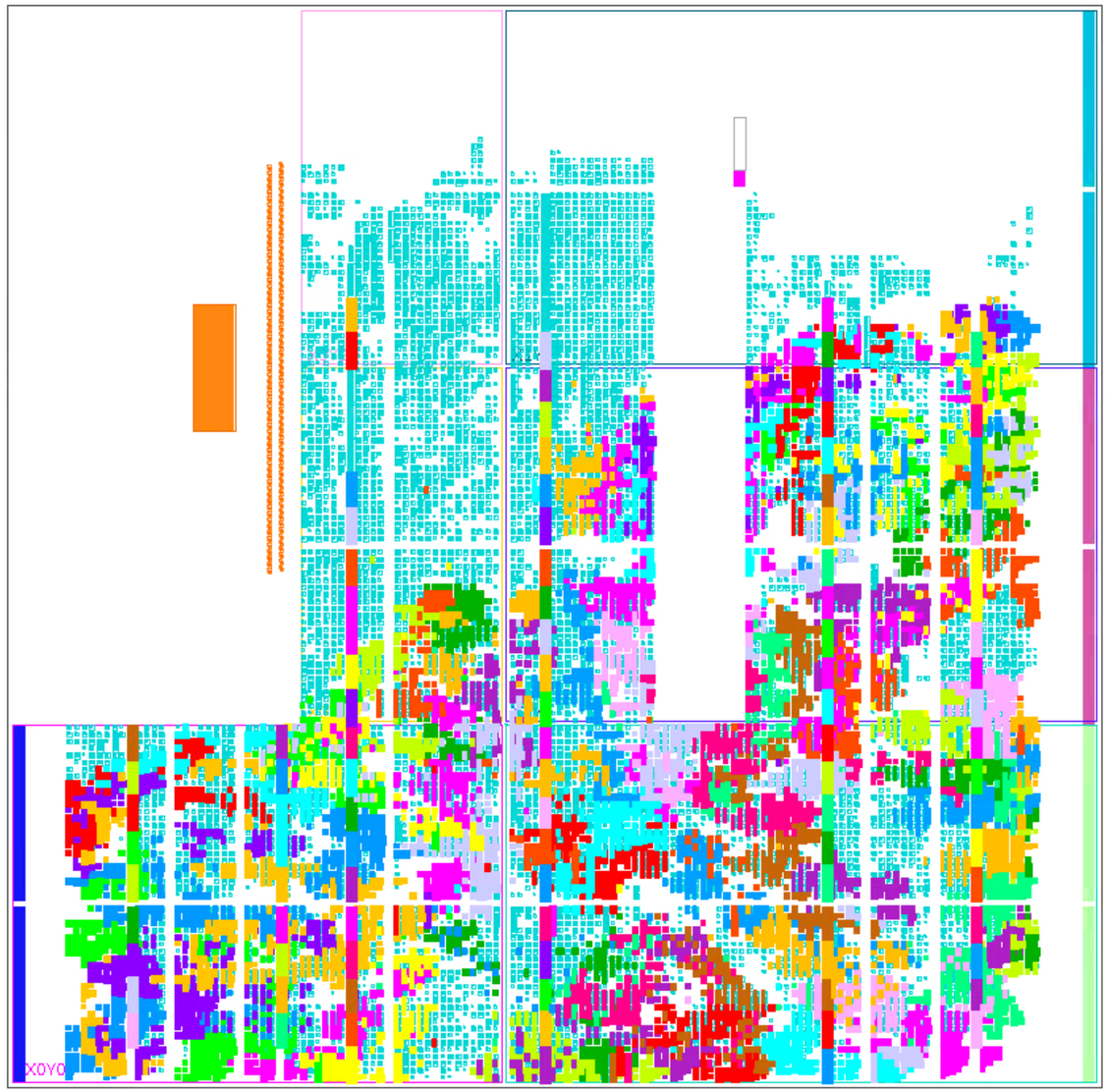




FPU Floorplan

- FPU with 100 Tree Engines in pynq-z2
  - Each TE is highlighted in colour (with a repeating cycle)
- BRAMs for nodes are in columns
- Logic near BRAMs is TE inference logic

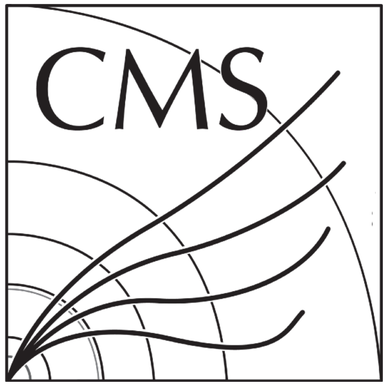




Section 2 summary

- We've looked at some steps beyond the first model-specific HLS
- Hand-written VHDL implementation of the same code
- Building accelerators from the model-specific HLS
- Designing reconfigurable architectures with HLS - the Forest Processing Unit
  - A design where the specific BDT model is unknown at build time, and loaded later as data
- Next we will try these three things
- Note: in the exercises we will run some 'accelerators'
  - Probably in practise they will actually be 'decelerators'
  - There are some examples where they give a real speed up, but it typically requires a large model and lots of data (batch size)

Break



Exercise 2

- Part 2a : building a static accelerator (Model-specific HLS → bitfile → runtime)
  - Will take around 1 hour of build time
- Part 2b: FPU hands on (straight to inference after downloading the bitfile)
  - Need to share access to the FPGA cards so don’t all try this at once
- Part 1b: if waiting for a synthesis or access to an Alveo try this VHDL notebook

Summary

- We have looked in detail at the conifer package for BDT inference on FPGAs
- We've gone through the different implementations and learned some more about HLS and FPGA programming
- This afternoon we will look at hls4ml for NNs on FPGAs