CERN openlab

Fast Decision Forest Inference on AMD AI Engines

CERN openlab Summer Student Programme Lightning talks

Ayoub Agouzoul

Supervisors: Sioni Summers, Ioannis Xiotidis

August 2026

40 million collisions/s, only microseconds to decide

The L1T/Global Trigger keeps roughly 1 event in 400

LHC collisions
40 MHz
every 25 ns a new event

Level-1 Trigger
~ microseconds
FPGAs decide: keep or discard

Kept events
~100 kHz
read out, stored, processed further

Decision forests are accurate, robust, and small enough to fit the latency budget.

Conifer

CERN-developed open-source library that turns a trained decision forest (e.g., a BDT) into ultra-low-latency firmware.

Already used in the ATLAS trigger during Run 3 (tau leptons); extensive use planned by ATLAS and CMS in Phase 2.

Supports: HLS / VHDL for FPGA fabric, plus CPU backends.

Goal of this project

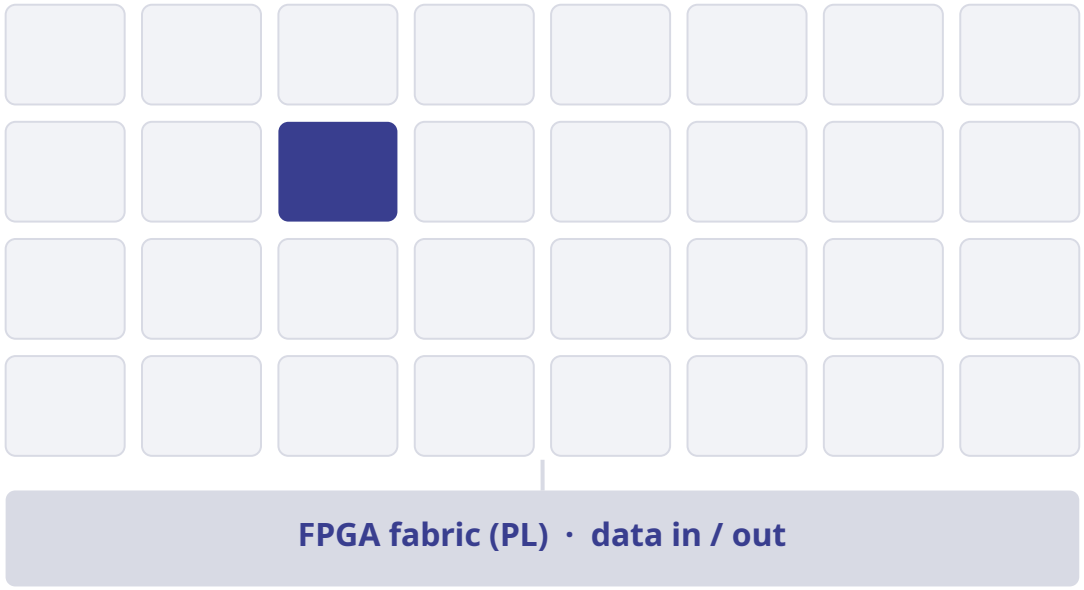
Add a new conifer backend targeting AMD AI Engines, the vector-processor array inside recent Versal SoCs.

Goal: the lowest possible latency per event, so decision forests keep fitting in the trigger budget.

AI Engines: a grid of vector processors next to an FPGA

Offering a second kind of compute — hundreds of small SIMD cores you program

AI Engine array



One tile = VLIW + SIMD vector unit at 1.25 GHz, its own local memory, connected to its neighbours by streams.

Add SIMD meaning

Why it is attractive

Hundreds of independent tiles, each doing many operations per cycle.

Very high compute density for the power — designed for ML inference.

Sits next to the fabric already used by the trigger.

Why it is hard

Fixed vector instruction set: you no longer design the datapath, you fit the algorithm to the machine.

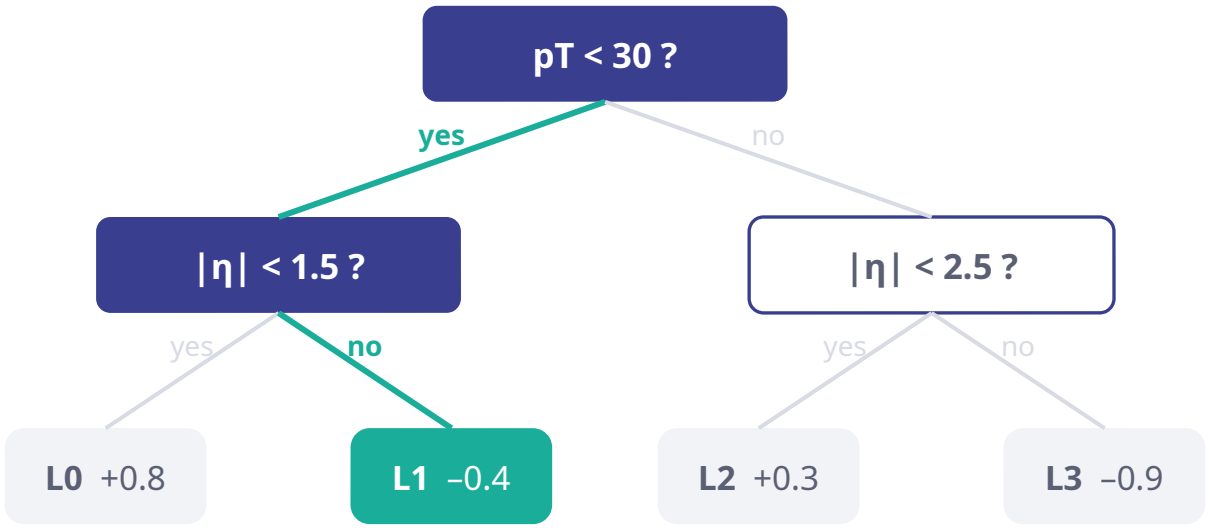
Vector lanes hate branches — and a decision tree is nothing but branches.

Feeding the tile fast enough is its own problem.

Typical decision tree inference: tree traversal

Toy example: one tree, two features, four possible outcomes

This event: **pT = 20 GeV** , **|η| = 3.0**



Path taken: **pT < 30 (yes) → |η| < 1.5 (no) → leaf L1, score –0.4 assigned.**

Why this is slow on a vector machine

Each comparison decides which node to read next, the memory access depends on the result of the previous one.

That makes a serial chain as deep as the tree, per tree, per event.

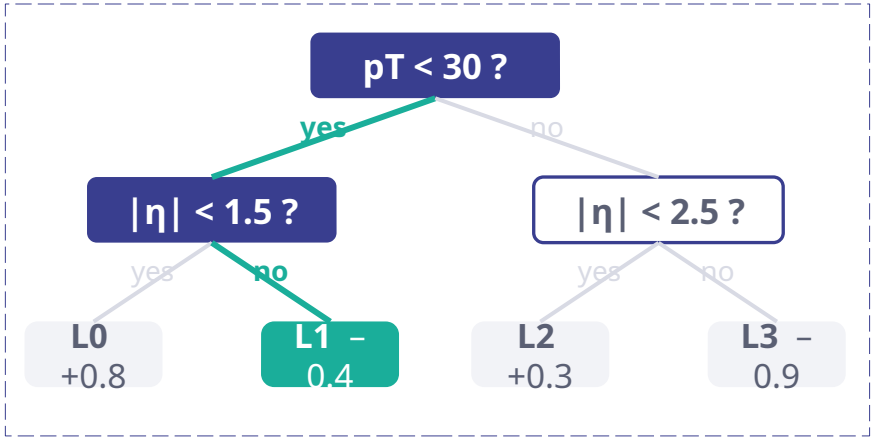
The branch is data-dependent: two events in two vector lanes want to visit different nodes, so the lanes cannot share work.

A forest has hundreds of trees, and all of them do this.

We need the same answer without following the path.

QuickScorer*: cross out the leaves

Every comparison is now independent



event: **pT = 20 GeV , |η| = 3.0**

	L0	L1	L2	L3	
Give every leaf a bit:	L0	L1	L2	L3	
pT < 30 ? TRUE	*skipped*				*test passes → nothing is excluded*
|η| < 1.5 ? FALSE	0	1	1	1	*fails → its left branch is unreachable*
|η| < 2.5 ? FALSE	1	1	0	1	*fails → its left branch is unreachable*
AND them together	0	1	0	1	
First surviving leaf		L1			**score –0.4 — same answer as walking the tree**

Why this suits a vector machine

- No branches and no data-dependent addresses: one instruction tests the same node for 32 events at once, and the forest becomes compares and bitwise ANDs.

Making it fast on the AI Engine

The biggest win was choosing how to spread the work

1 · Vectorise across events

One lane = one event, so every lane tests the same node: compile-time constants and contiguous loads.

Spreading trees across lanes instead needs a different feature per lane: inefficient memory load. Cost of one extra feature: +78 cycles that way, +0.6 this way.

Same ensemble, same tile: 5 595 cycles per event scoring one event at a time, 167 scoring 32 at once.

2 · Keep the core fed

Stalls ran to 19% of cycles, dominated by memory: the kernel was reading its tables one scalar at a time. Removing those loads bought 1.38×.

Streaming events into the core instead of buffering a group removes the fill from the critical path — the buffered path waits about a cycle per feature word before it can start, the streamed one does not.

A 64-bit input port then removed what stream stall was left.

3 · Get the answer back out

The exit leaf is the lowest set bit of a mask — and this machine has no vector bit-scan at all. Building it out of mask tests instead: −22% to −31% per tree at depths 3 to 6.

Storing each tree's nodes together, rather than grouping nodes by feature, cut the scalar loads from 1431 to 480 per pass — 1.5× on an unchanged algorithm.

Challenges

Every point is a cycle-accurate simulation, hours each — so the workflow had to become measure-first, guess-never.

The first 5–11× was a mapping difference, not the algorithm: against an equally-mapped baseline it is 1.46×.

Depth is the limit — QuickScorer's cost grows 2.07× per level against naive's 2.00×, so the win is a band, not a trend.

Results

On a single AI Engine-ML tile, optimized kernel

134 ns — per event @ 32 trees, depth 4, 32 events per pass

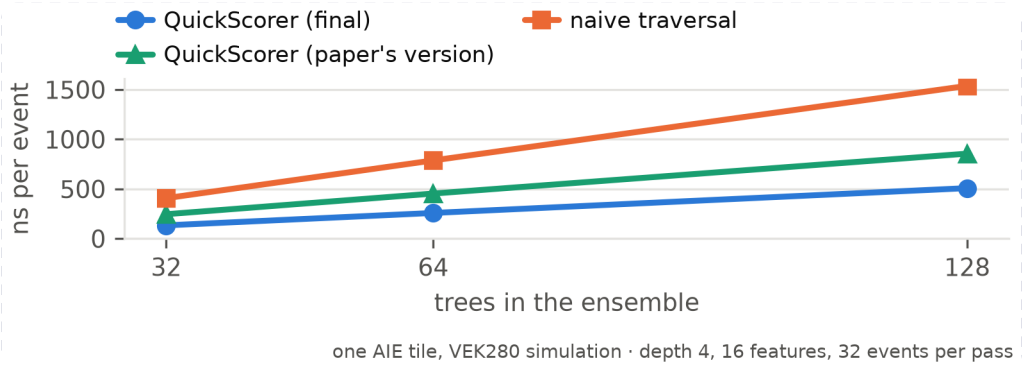
4.2 ns — per tree per event: 5.2 cycles @ 1.25 GHz

1.46x — faster than an equally optimized naive traversal

Also measured

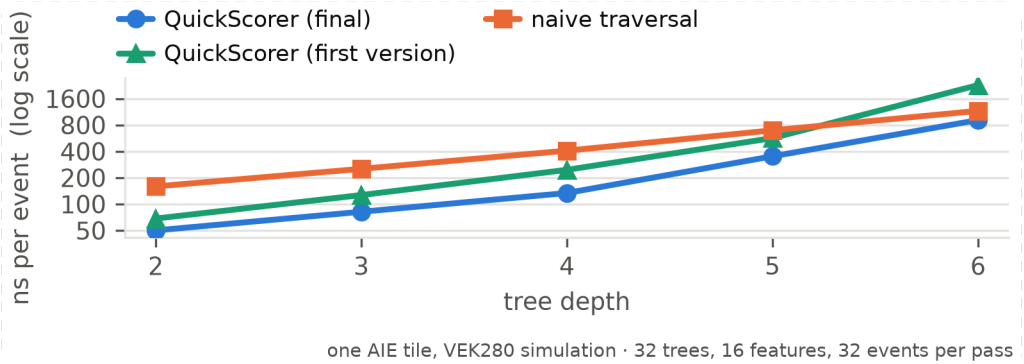
- linear in ensemble size and ~2.0x per extra tree level.
- every simulation is bit-exact against the QS conifer CPU backend.

Cost per event vs. ensemble size



one AIE tile, VEK280 simulation · depth 4, 16 features, 32 events per pass

Cost per event vs. tree depth



one AIE tile, VEK280 simulation · 32 trees, 16 features, 32 events per pass

Conclusion

And next steps…

Run on real silicon

Everything so far is cycle-accurate simulation. First job: measure the same kernels on the board.

Scale across tiles

One tile gives the latency. Replicating over many tiles will give higher throughput.

Oblique splits

Let a node cut on a combination of features instead of one. More representative power for the same depth.

conifer backend

Wire current code up as a proper AI Engine backend. QuickScorer already exists in conifer as Python/C++ CPU backends.

Takeaway

Decision forests look like the worst possible fit for a branch-free vector machine. Rewriting the inference as compares and bitwise ANDs makes them fit, and the AI Engine array becomes a credible target for the next generation of triggers.

Thank you!

With thanks to Sioni Summers and Ioannis Xiotidis, to Dimitrios Danopoulos and Christopher Brown,
and to CERN, CERN openlab and the Next Generation Triggers project.