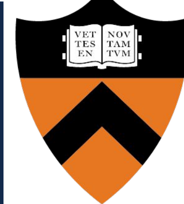


GNNs for Particle-Tracking on FPGAs

Connecting the Dots, May 31, 2022

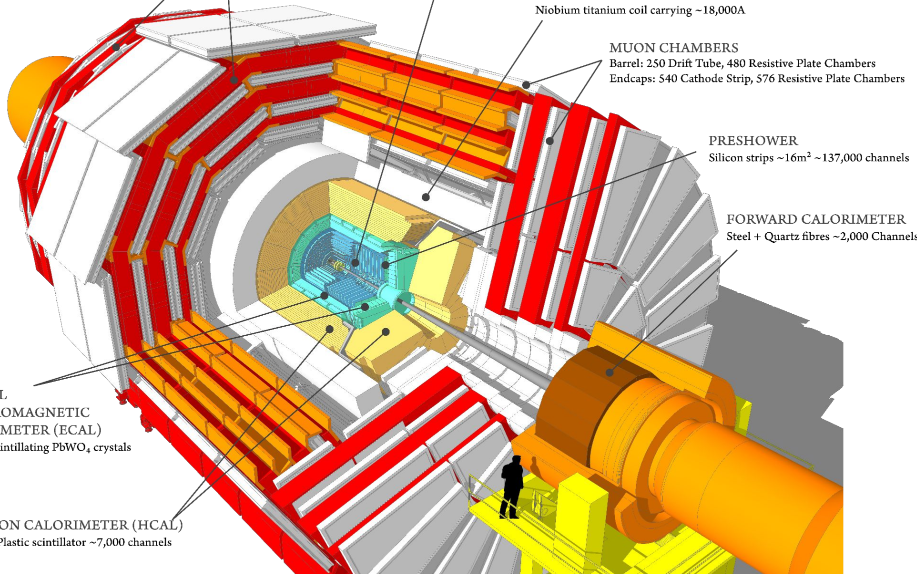
Abdelrahman Elabd, Vesal Razavimaleki, Shi-Yu Huang, Javier Duarte, Markus Atkinson, Gage DeZoort, Peter Elmer, Scott Hauck, Jin-Xuan Hu, Shih-Chieh Hsu, Bo-Cheng Lai, Mark Neubauer, Isobel Ojalvo, Savannah Thais, Matthew Trahms

Front. Big Data 5:828666 (2022)

Outline

1. The overarching problem
    a. **The machine-learning problem:** How do we perform particle-tracking at the Large Hadron Collider (LHC)?
    b. **The hardware problem:** What are the hardware limitations (for particle-tracking) set by the High-Luminosity LHC (HL-LHC)?
2. The machine-learning problem
    a. What are graphs and graph neural networks (GNNs)?
    b. How can GNNs be used for particle-tracking at the LHC?
3. The hardware problem
    a. What are FPGAs, and why do we need them at the LHC?
    b. What is High-Level Synthesis (HLS), and why do we need it in order to use FPGAs?
    c. How do we use FPGAs and HLS to meet the hardware limitations set by the HL-LHC?

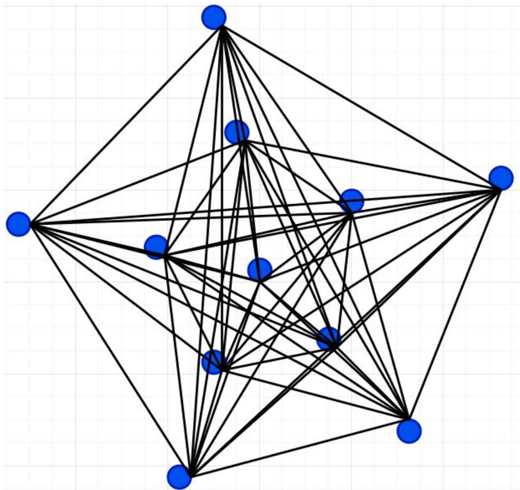
The overarching problem

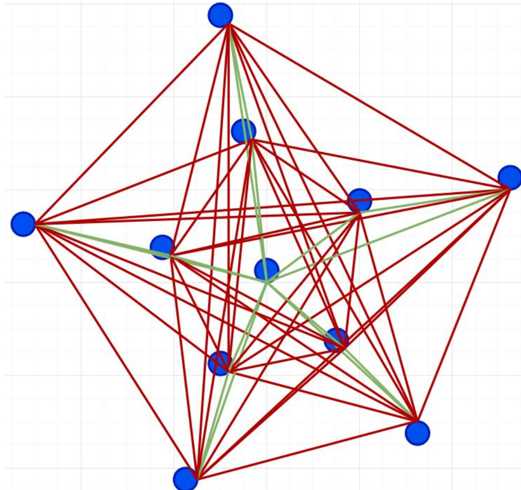


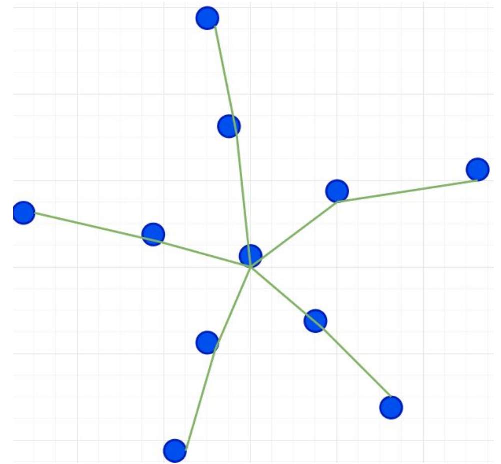
Example: CMS

- L1 Trigger: 4 microsecond latency limit
- High-Level Trigger (HLT): 1 khz output rate
- The High-Luminosity LHC will increase the number of proton-proton collisions in any collision event by about 5x-7x
- Current tracking algorithms scale **worse than quadratically** with the number of detector-hits

Graphs and Graph Neural Networks





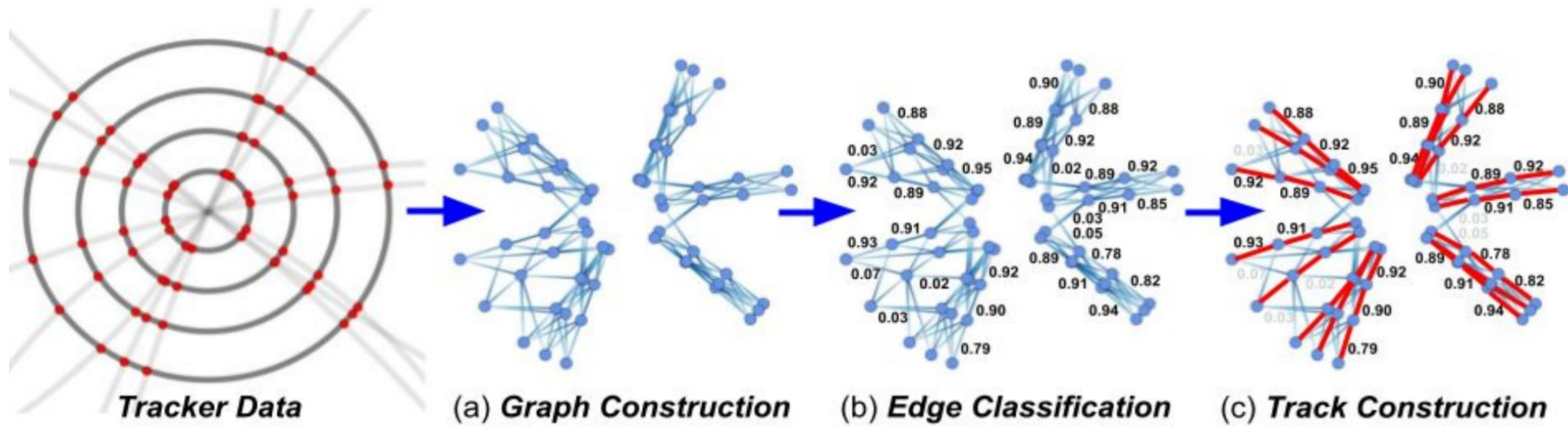


Track-segment candidates

Classified track-segments

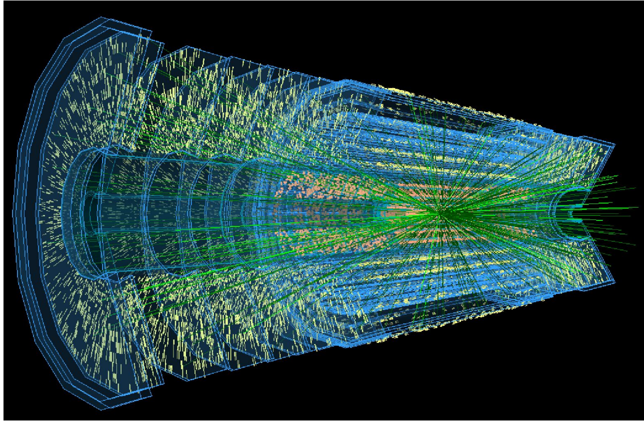
Tracks

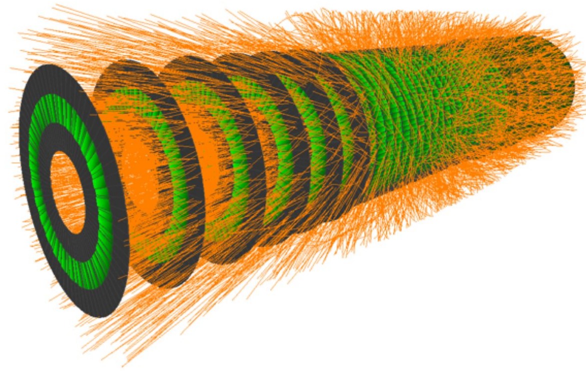
The machine-learning problem: Workflow



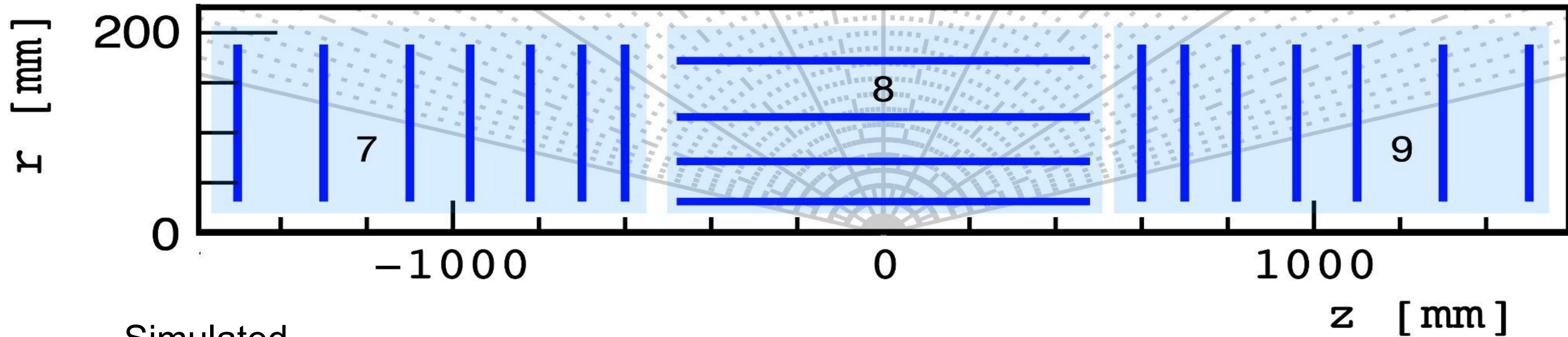
- **Graph construction**
- **Edge classification ↔ Track-segment construction**
- Full-track construction

Dataset



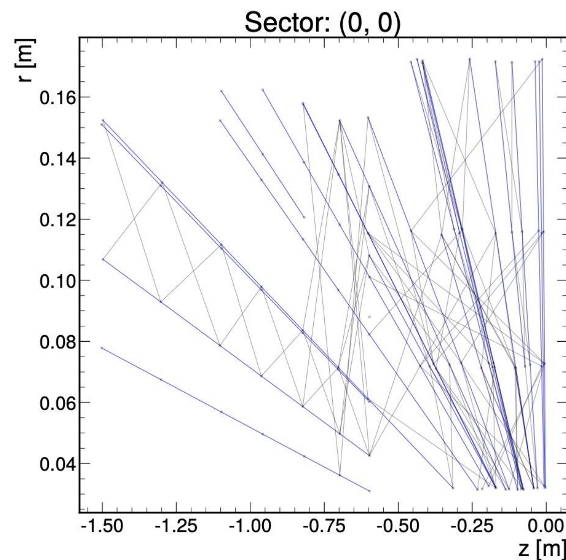


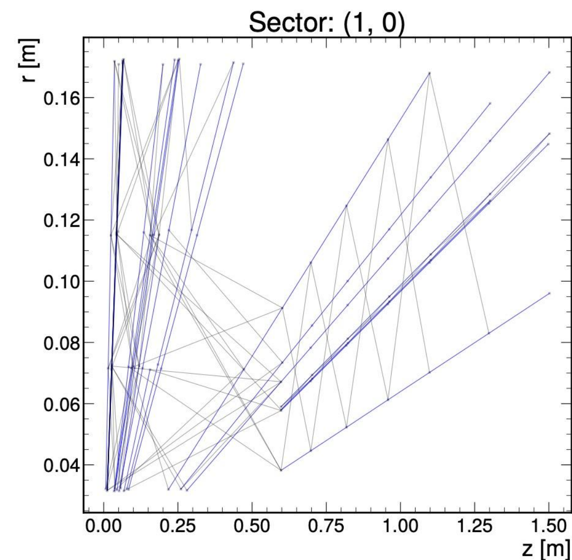
TrackML Dataset: kaggle.com/c/trackml-particle-identification/data [arXiv:1904.06778]



- Simulated
- LHC-like detector
- High-Luminosity pileup

Graph construction

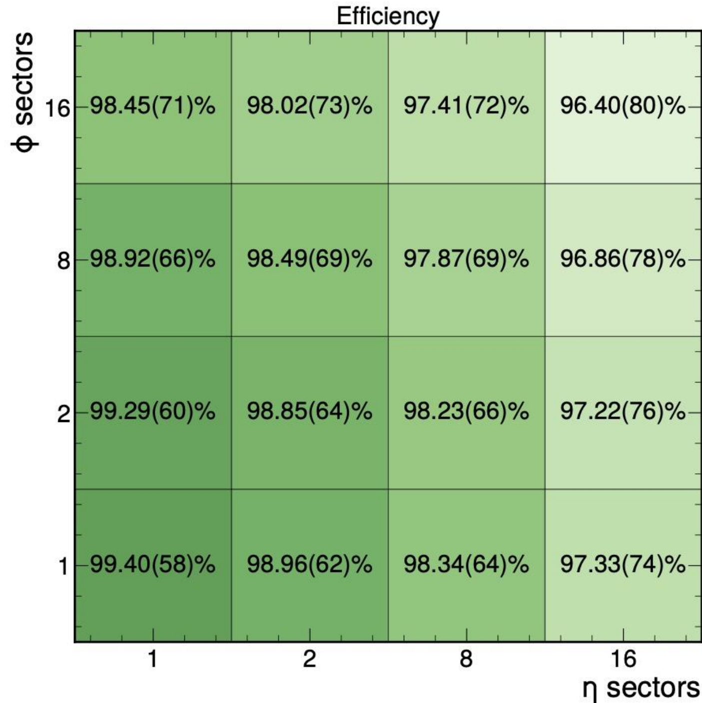


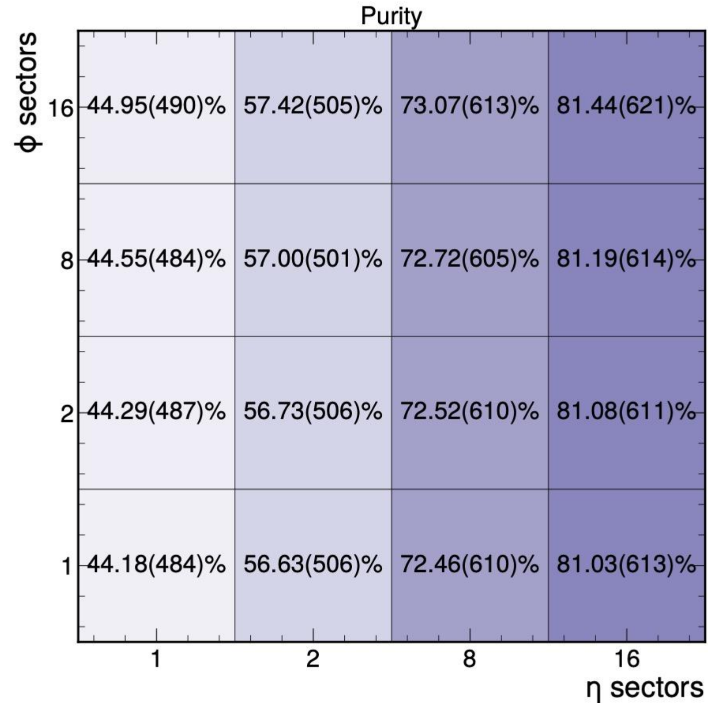


- Geometric cut:
    - **phi_slope_max:** 0.0006; $\Delta\phi / \Delta r = \frac{\phi_j - \phi_i}{r_j - r_i}$
    - **z0_max:** 15000 mm; $z_0 = z_i - r_i \frac{z_j - z_i}{r_j - r_i}$
- **pt cut:** 2 GeV
- Segmentation
    - **n_phi_sections:** 8, **n_eta_sections:** 2
- n_nodes, n_edges @ 95th percentile: (113, 196)
- Graph construction efficiency: 98% and purity: 57%

Purity/efficiency

- 98% efficiency and 57% purity for 2 eta sectors, 8 phi sectors

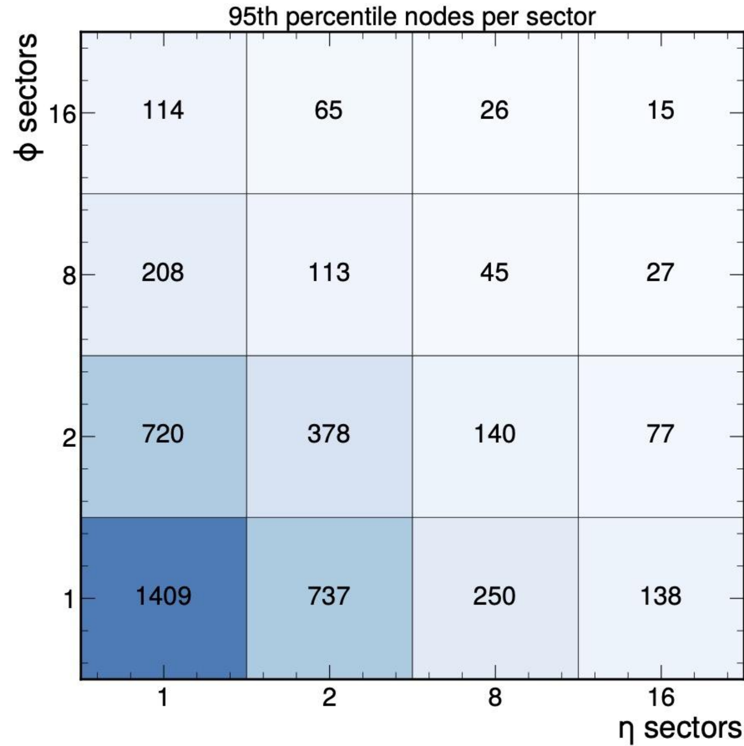


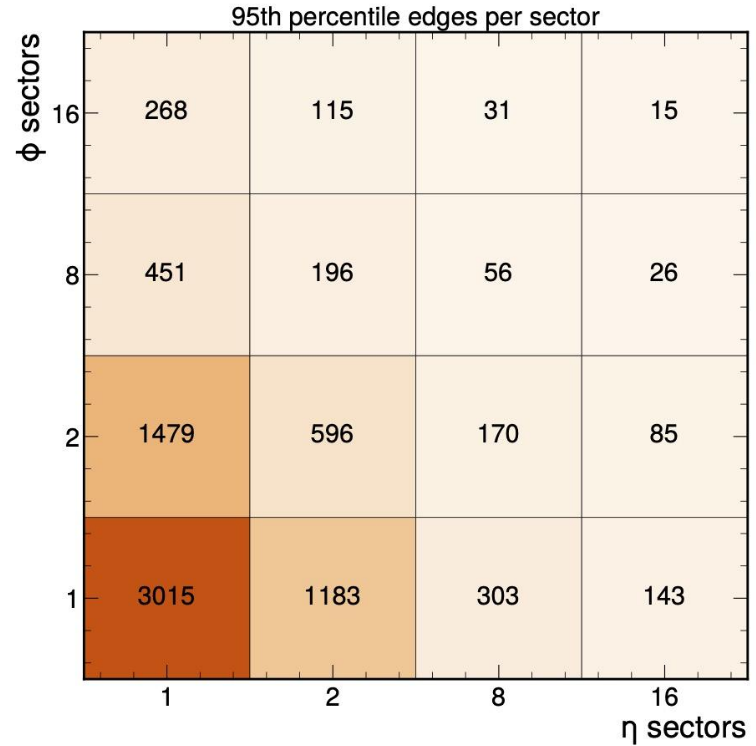


Efficiency = **(**# of true segments after cut**)/(**# of true segments before cut**)**
Purity = **(**# of true segments after cut**)/(**# of total segments after cut**)**

95th percentile nodes/edges

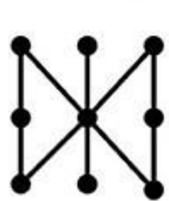
- 113 nodes and 196 edges for 2 eta sectors, 8 phi sectors

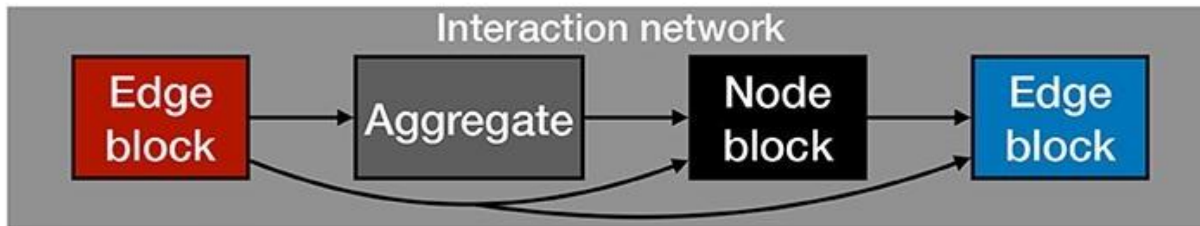


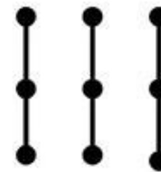


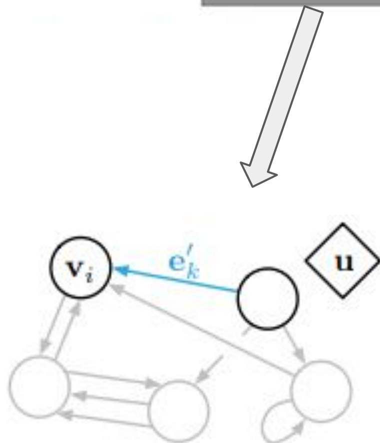
Edge-classification model: Interaction Network

- Dezoort et al., 2021: arXiv:2103.16701

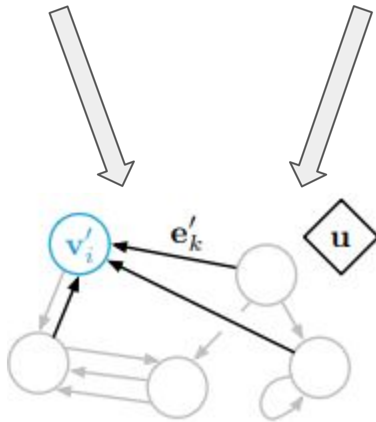






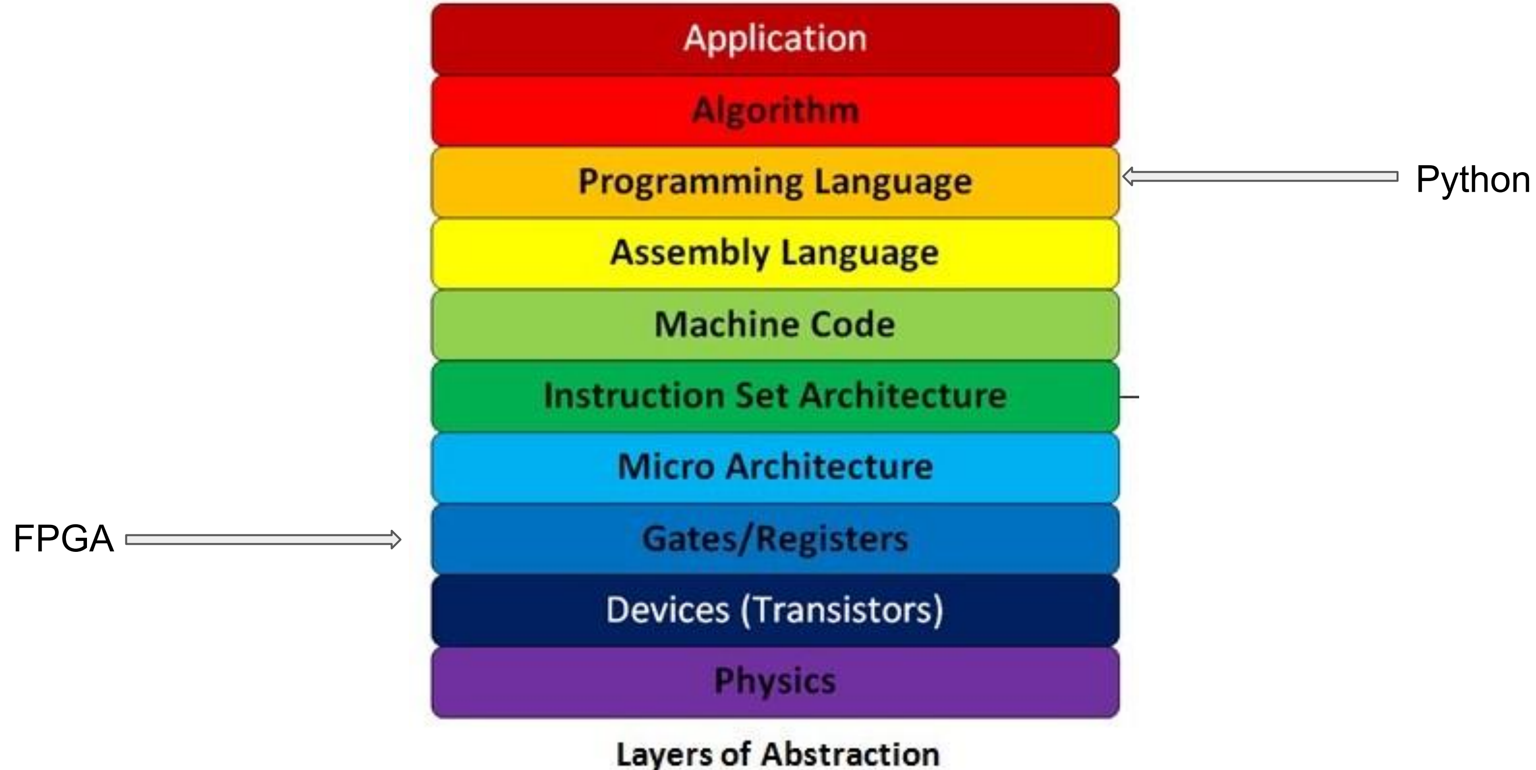


+

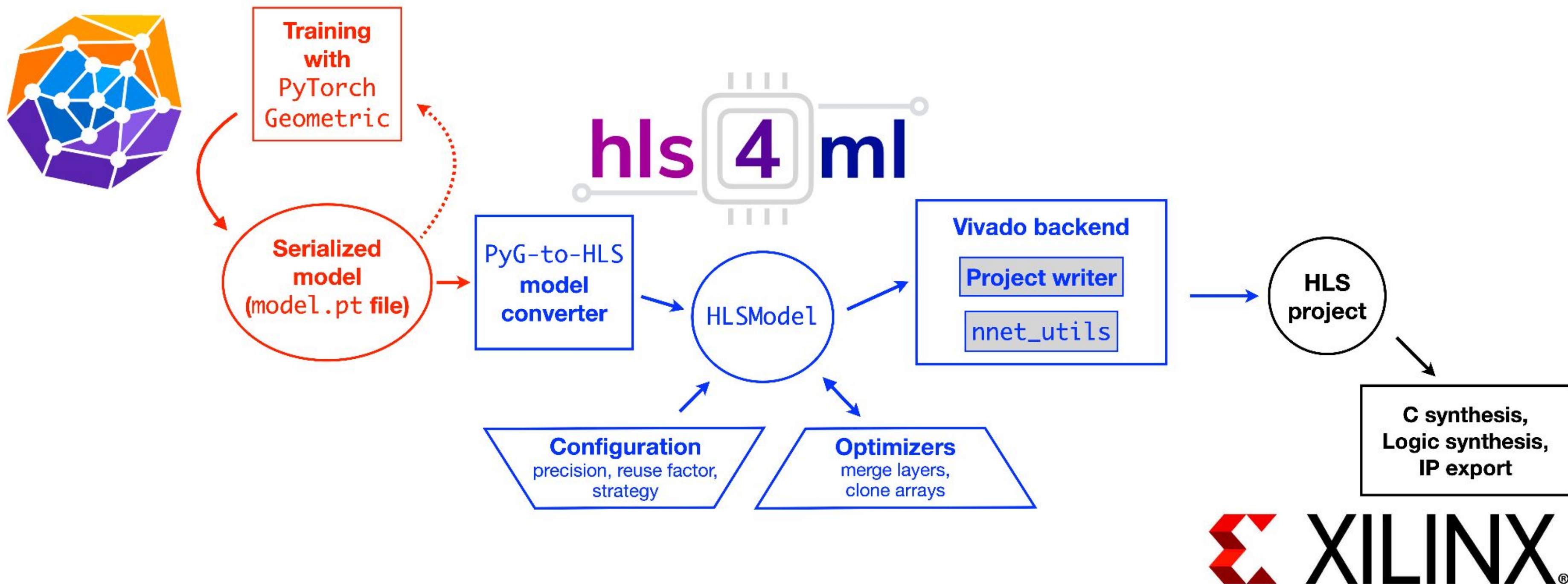


= Message Passing

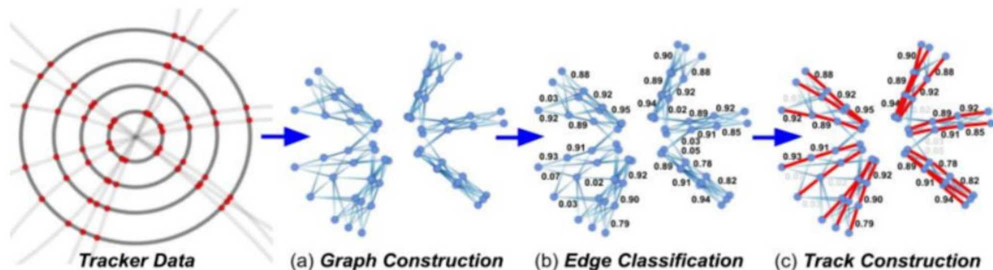
The hardware problem: CPUs vs FPGAs

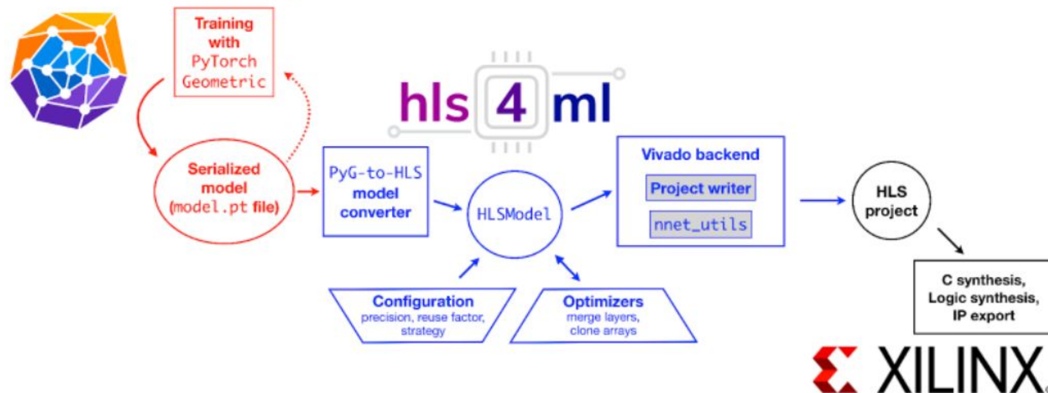


The hardware problem: Workflow

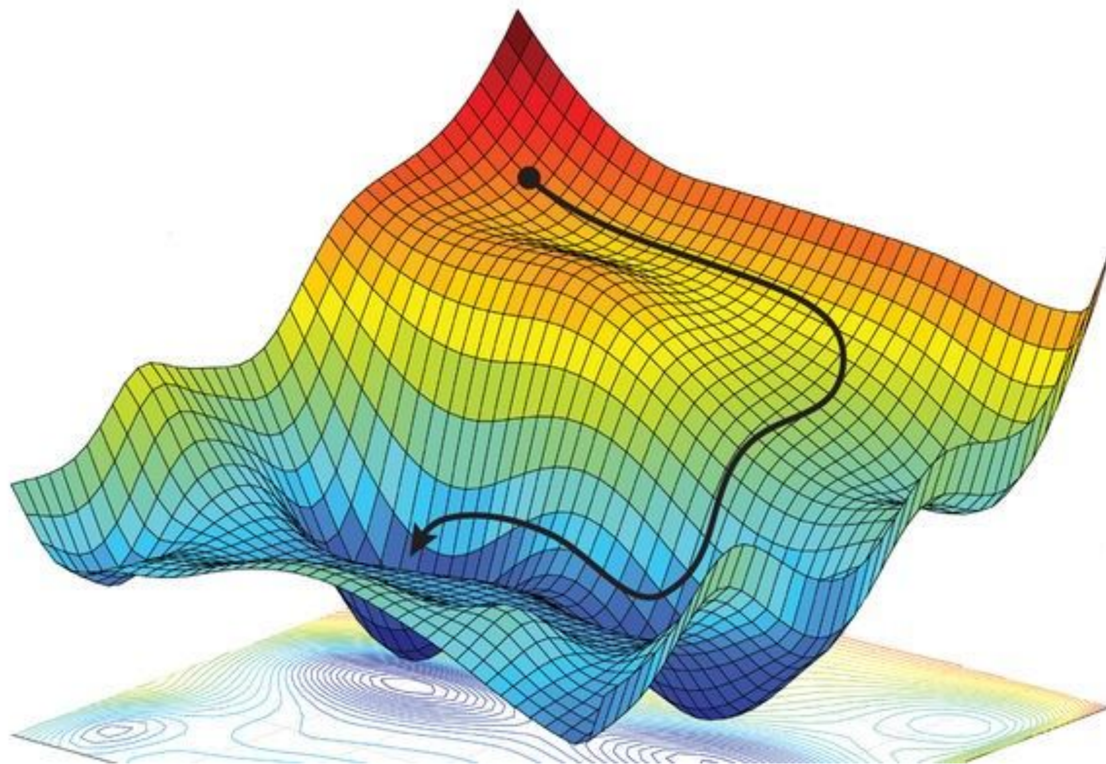


The big picture: GNNs for Particle-Tracking on FPGAS

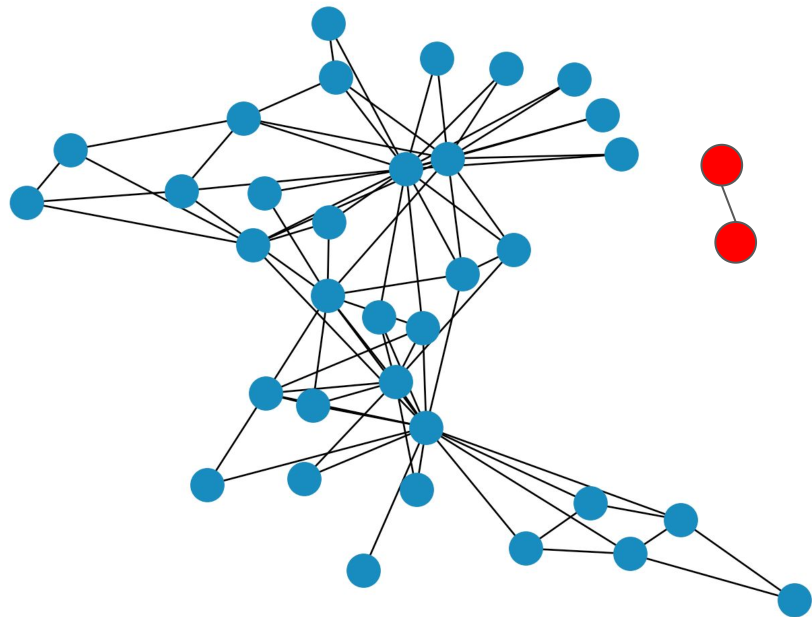


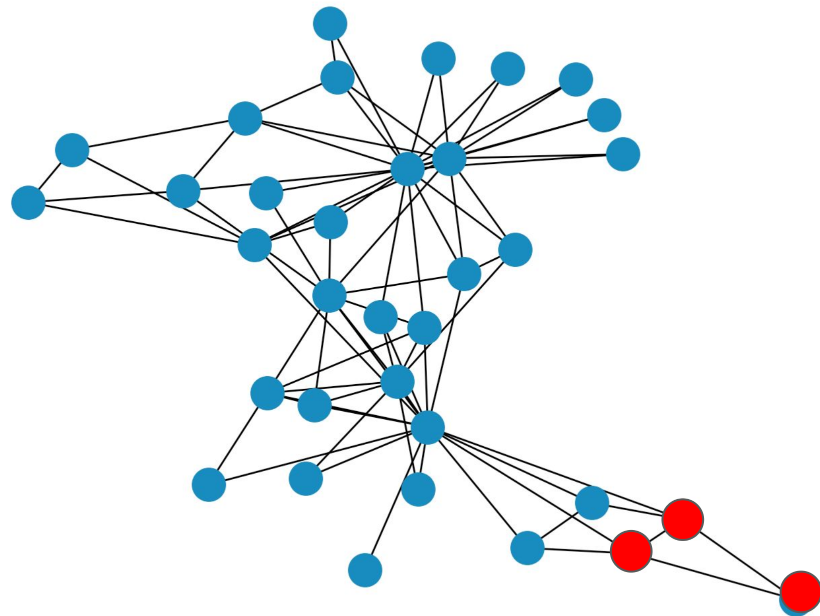


Post Training Quantization vs. Quantization Aware Training



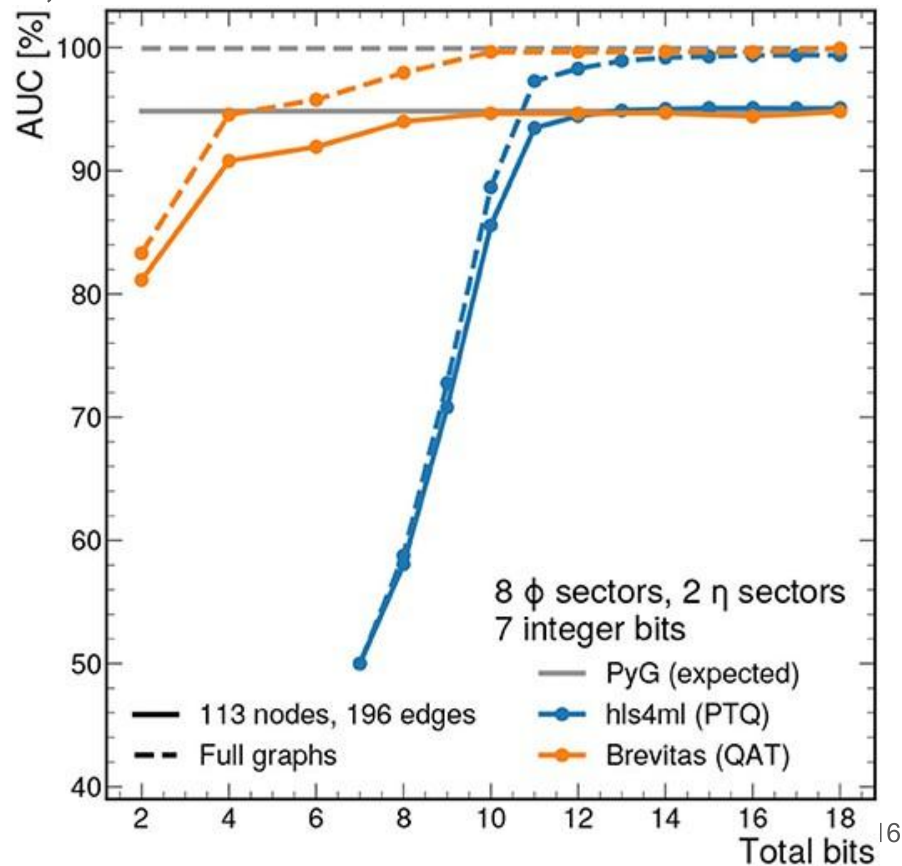
Padding and Truncation





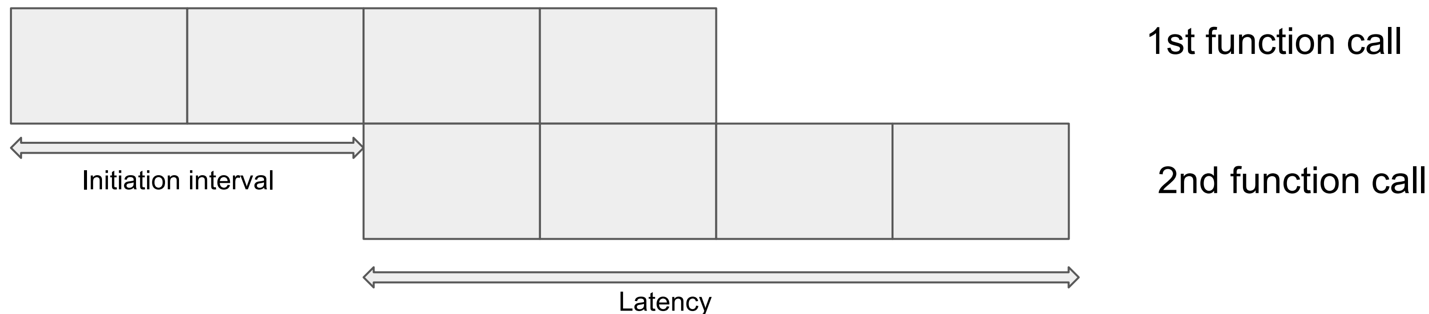
Performance vs Precision

- AUC vs. total bit width X for ap_fixed<X,X/2>
- PTQ = Post Training Quantization
- QAT = Quantization Aware Training



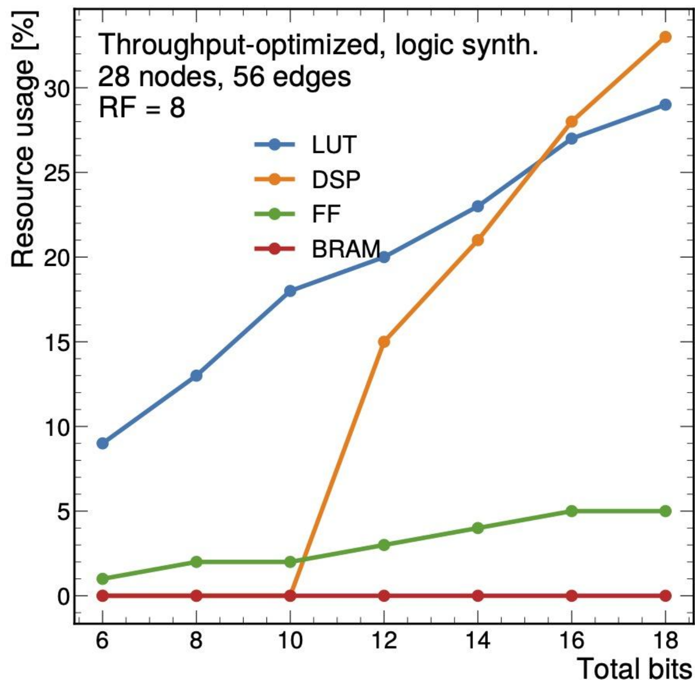
FPGA HLS Implementations

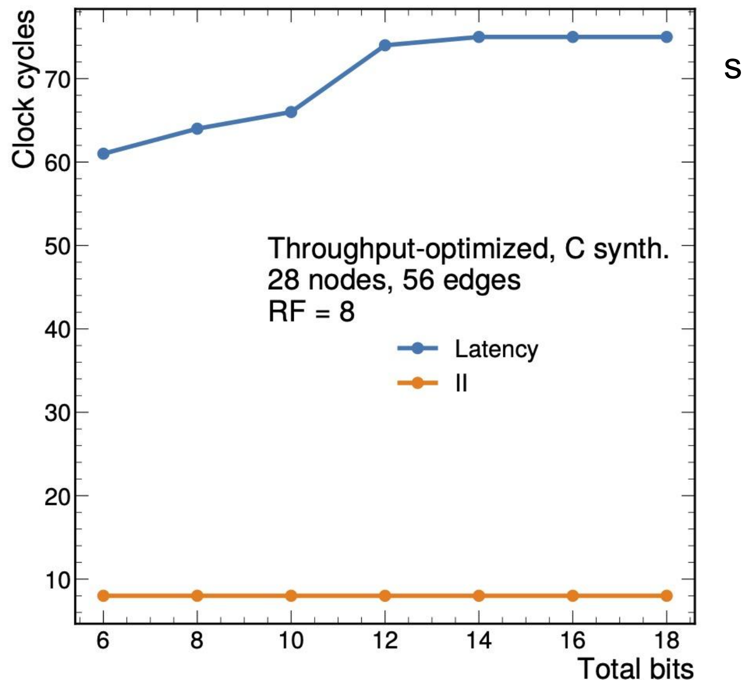
- Two implementations:
  - Throughput-optimized
    - Lower latency/initiation-interval
    - Greater resource-usage
    - Smaller graphs
  - Resource-optimized
    - Greater latency/initiation-interval
    - Lower resource-usage
    - Larger graphs



Precision scan, throughput-optimized

- FPGA: Xilinx Virtex UltraScale+ VU9P
- L1 latency limit = 4 microseconds = 800 clock cycles
- ~30% of maximum resource usage, <10% of maximum latency

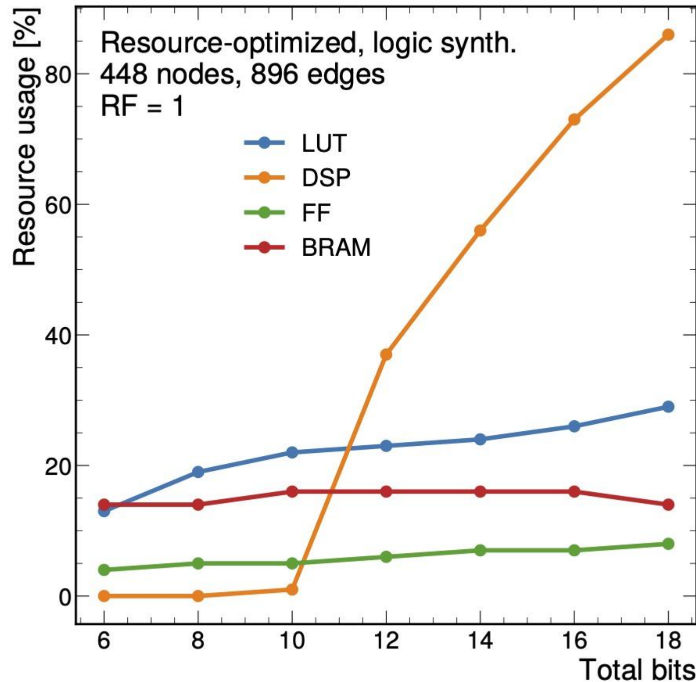


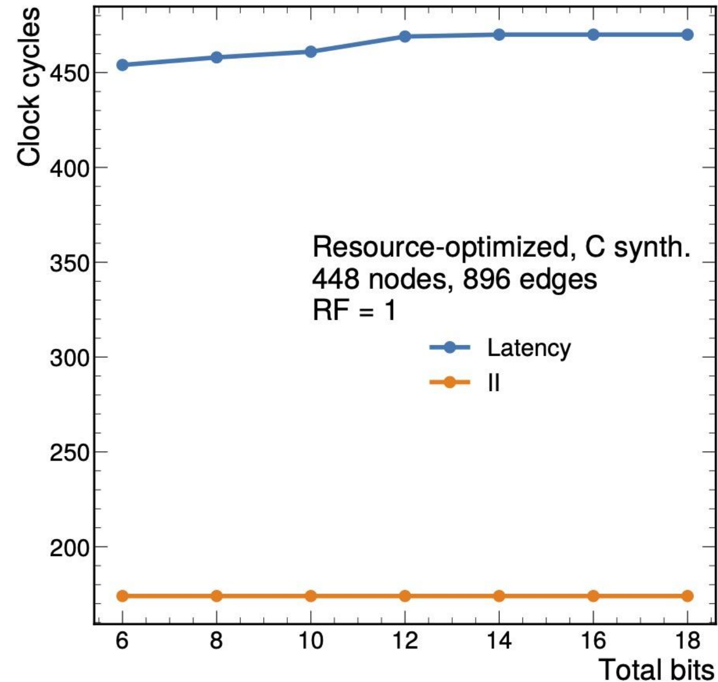


s

Precision scan, resource-optimized

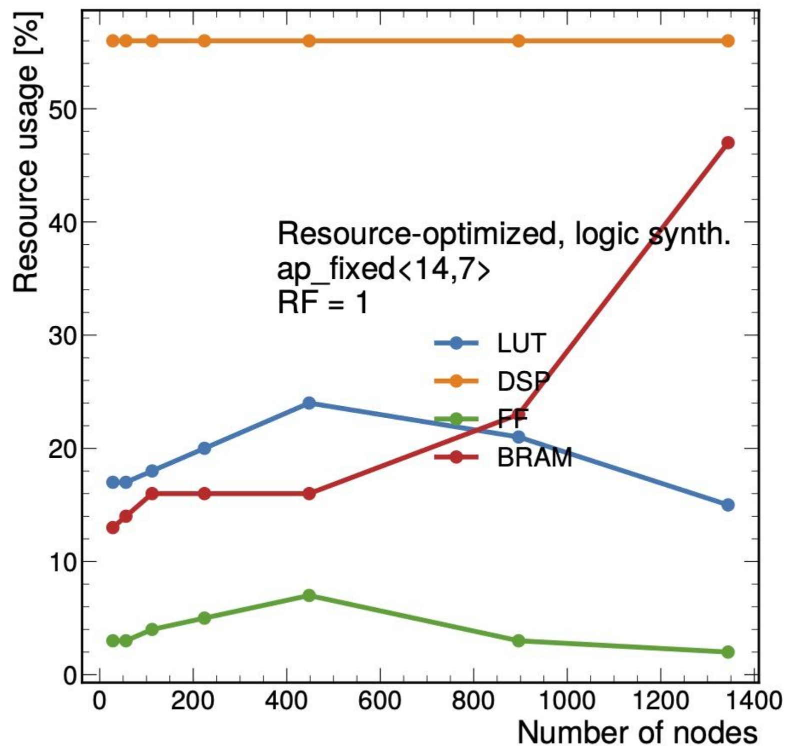
- DSPs: used in matrix multiplication
- Latency = 450 clock cycles = 2.25 microseconds

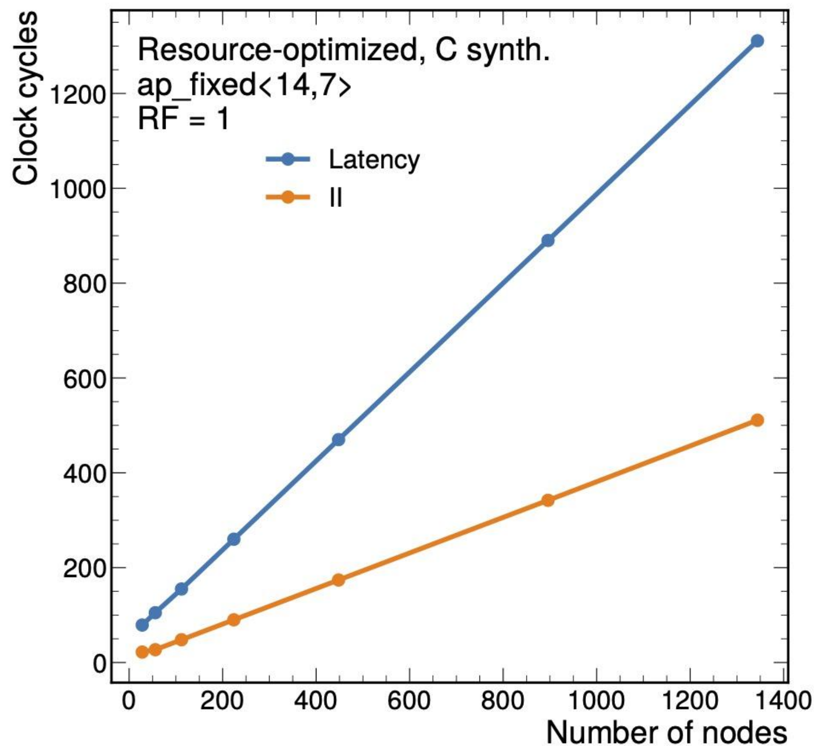




Size scan, resource-optimized

- Precision fixed at 14 bits
- Bigger graphs: High-Level Trigger coprocessor





Summary and outlook

- Applications
  - Throughput-optimized
    - Small graphs (28 nodes with 56 edges, or smaller)
    - <1 microsecond latency
      - Real-time L1 trigger
  - Resource-optimized
    - Larger graphs (1344 nodes with 2688 edges, or slightly larger)
    - 0.5-6 microseconds depending on graph-size
      - Real-time L1 trigger for smaller graphs
      - Offline L1 trigger
      - High-level trigger
      - CPU coprocessor application
- Future efforts
  - Throughput-optimized synthesizability for larger graphs
  - FPGA implementation: Graph construction, track construction
    - Serializing data flow between the different parts of the machine-learning workflow

iris
hep

Backup

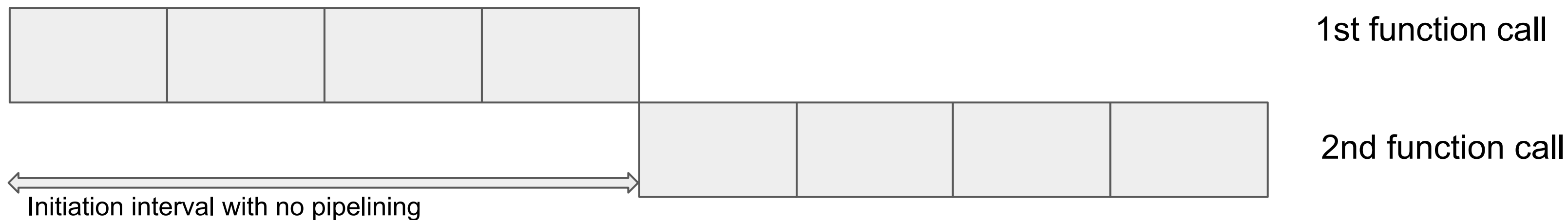
Where's the code?

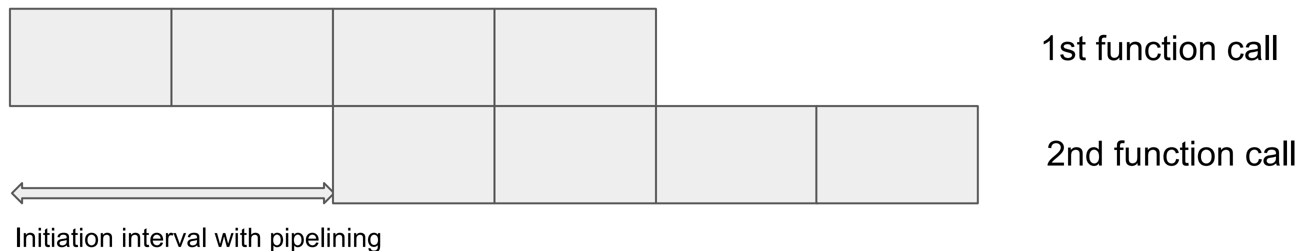
- With just the throughput-optimized backend
  - https://github.com/abdelabd/hls4ml/tree/pyg_to_hls_rebase
  - Pull-request in the works: https://github.com/fastmachinelearning/hls4ml/pull/379

- With both the throughput-optimized and resource-optimized backends
  - https://github.com/abdelabd/hls4ml/tree/pyg_to_hls_rebase_w_dataflow

HLS Preprocesser Directives

- **Array_partition:** Splits large arrays into smaller arrays for concurrent memory access
- **Unroll**: Unrolls for-loops in space, so that each iteration can happen concurrently
- **Pipeline**: Utilizes idle resources to minimize II

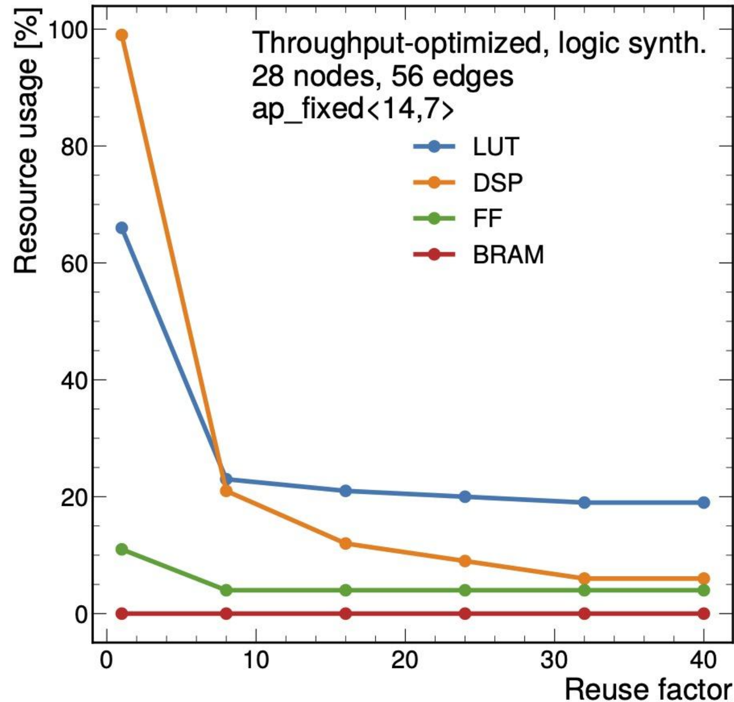


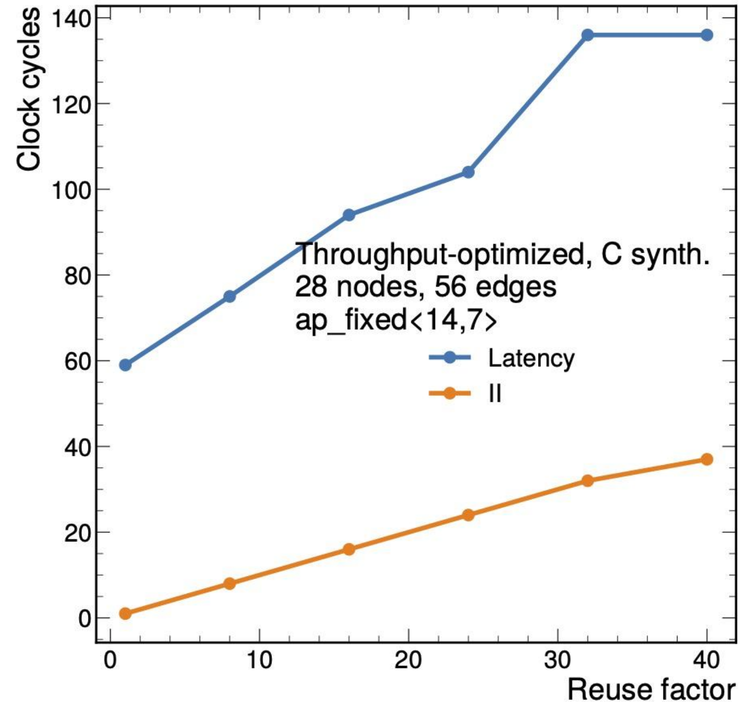


...

Reuse factor scan, throughput-optimized

- 28 nodes, 56 edges, ap_fixed<14,7>, VU9P
- RF can reduce resource usage





Throughput-optimized implementation

- Overall design is pipelined between each block
- Fully partitioned arrays
- Fully unrolled loops

Algorithm 1 Throughput-optimized edge block.

```
Input: node_attr[n_nodes][node_dim], edge_attr[n_edges][edge_dim], edge_index[n_edges][2]
Output: edge_update[n_edges][edge_dim]
  pipeline algorithm with factor RF                                   ▷ HLS pragma
  procedure PARTITION ARRAYS(node_attr, edge_attr, edge_update)
      completely partition node_attr                                  ▷ HLS pragma
      completely partition edge_attr                                  ▷ HLS pragma
      completely partition edge_update                                ▷ HLS pragma
  end procedure
  procedure CREATE NN INPUT(node_attr, edge_attr, edge_index)
      initialize phi_input[n_edges][edge_dim+2×node_dim]
      completely partition phi_input                                  ▷ HLS pragma
      for i ← 1, n_edges do
          completely unroll loop                                      ▷ HLS pragma
          receiver_index ← edge_index[i][0]
          sender_index ← edge_index[i][1]
          receiver ← node_attr[receiver_index]
          sender ← node_attr[sender_index]
          edge ← edge_attr[i]
          phi_input[i] ← ⟨receiver, sender, edge⟩
      end for
      return phi_input
  end procedure
  procedure COMPUTE EDGE UPDATE(phi_input)
      for i ← 1, n_edges do
          completely unroll loop                                      ▷ HLS pragma
          edge_update[i] ← NN(phi_input[i])
      end for
      return edge_update
  end procedure
```

Resource-optimized implementation

- Overall design is pipelined between each block
- Functions and loops within each block are also pipelined (1)
- Duplicate arrays for parallel, concurrent access (2)
- Loop-unrolling matches parallelization (3)
- Array-partitioning matches parallelization (4)

Algorithm 2 Resource-optimized edge block.

Input: node_attr[n_{nodes}][node_dim], edge_attr[n_{edges}][edge_dim], edge_index[n_{edges}][2]
Output: edge_update[n_{edges}][edge_dim]

procedure PARTITION ARRAYS(node_attr, edge_attr, edge_index, edge_update)
    cyclically partition node_attr with factor node_dim×PF ▷ HLS pragma
    cyclically partition edge_attr with factor edge_dim×PF ▷ HLS pragma
    cyclically partition edge_index with factor 2×PF ▷ HLS pragma
    cyclically partition edge_update with factor edge_dim×PF ▷ HLS pragma
    (4)
end procedure
procedure COPY NODE ATTRIBUTES(node_attr) (2)
    initialize node_attr_copies[PF][n_{nodes}][node_dim]
    for $i \leftarrow 1, n_{\text{nodes}}$ **do**
        unroll loop with factor PF (3) ▷ HLS pragma
        pipeline loop with factor RF (1) ▷ HLS pragma
        for $j \leftarrow 1$, PF **do**
            completely unroll loop ▷ HLS pragma
            node_attr_copies[j][i] ← node_attr[i]
        end for
    end for
    return node_attr_copies
end procedure
procedure COMPUTE EDGE UPDATE(edge_attr, node_attr_copies, edge_index)
    for $i \leftarrow 1, n_{\text{edges}}$ **do**
        unroll loop with factor PF (3) ▷ HLS pragma
        pipeline loop with factor RF (1) ▷ HLS pragma
        receiver_index ← edge_index[i][0]
        sender_index ← edge_index[i][1]
        receiver ← node_attr_copies[i%PF][receiver_index]
        sender ← node_attr_copies[i%PF][sender_index]
        edge ← edge_attr[i]
        phi_input ← ⟨receiver, sender, edge⟩
        edge_update[i] ← NN(phi_input)
    end for
    return edge_update
end procedure

Resource and latency scans

- Precision scan: reuse-factor = 8, over a range of fixed-point precision with total bits from 6 to 18
- Reuse factor scan: ap_fixed<14,7>, over a range of reuse factors from 1 to 40
- Graph-size scan: ap_fixed<14,7> and reuse-factor=8, over a range of graph-sizes from 7 nodes with 14 edges to 1388 nodes with 2776 edges.
- Resource-usage numbers are retrieved from Vivado (logic) synthesis, and latency numbers are retrieved from Vivado HLS (C) synthesis
- Two designs: throughput-optimized and resource-optimized
- Maximum graph sizes:
  - 28 nodes, 56 edges for throughput-optimized
  - 1,344 nodes, 2,688 edges for resource-optimized