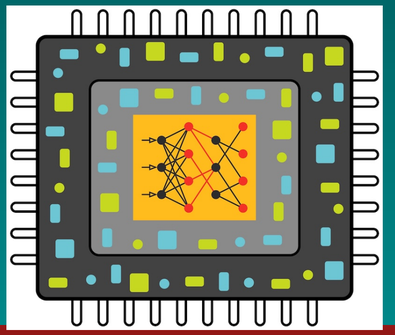
University of Twente

Dynamics-Based Maintenance (DBM)
Computer Architecture for Embedded Systems (CAES)

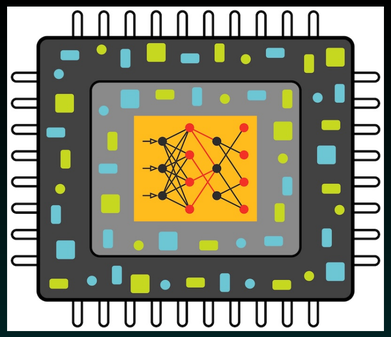
TrackCore-F: Tracking Transformer Synthesis for Low-Latency FPGA Deployment

Arjan Blankestijn, Uraz Odyurt, Amirreza Yousefzadeh

2025-06-17
EuCAIFCon Conference



Overview



- Can we achieve online or pseudo-online tracking?
  => Tracking at run-time using ML algorithms …

- Traditional tracking (**post-mortem**)
  => Iterative
  => Does not scale

- Using ML, specifically Transformers (**still post-mortem**)
  => One-shot tracking (per event)
  => Suitable for hardware acceleration

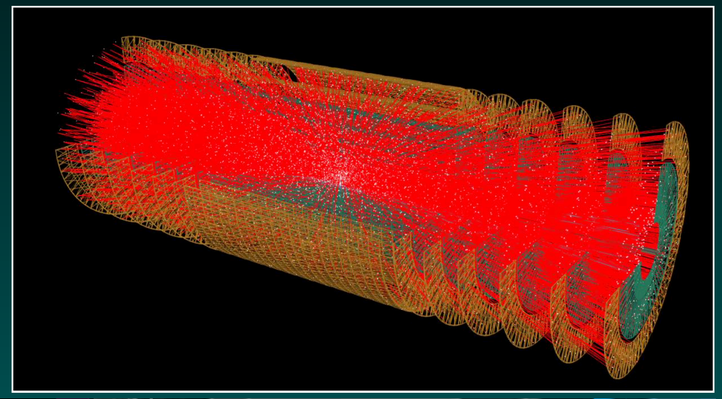
This is ongoing research …

Subatomic particle tracking

- Tracking: Reconstruct **particle trajectories** from recorded hits
  => Has to happen at every event

- Why we do this? Two main measurements:
  => Tracking and calorimetry -> **Momentum** and **energy**
  => Discover/study **particle behaviour**

- Present algorithms use **Kalman filters**
  => **Not linearly** scalable
  => Multiple steps, multiple passes
  => Cannot support HL-LHC era [2029-] (pile-up)
  => Numerous detector HW upgrades
  => SW and algorithm upgrades



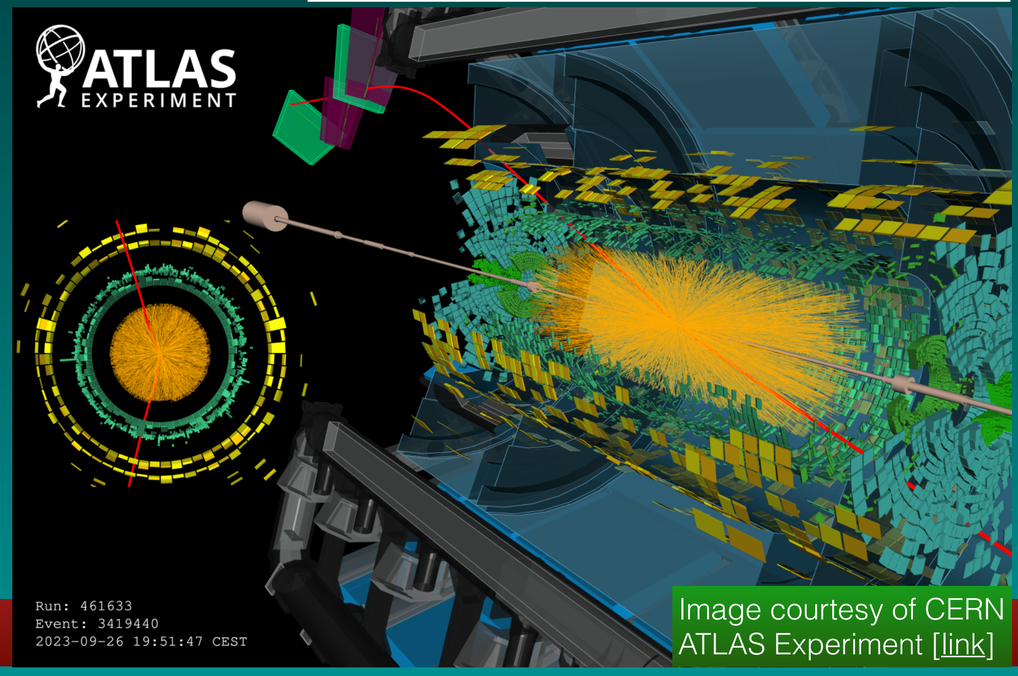


Image courtesy of CERN
ATLAS Experiment [link]

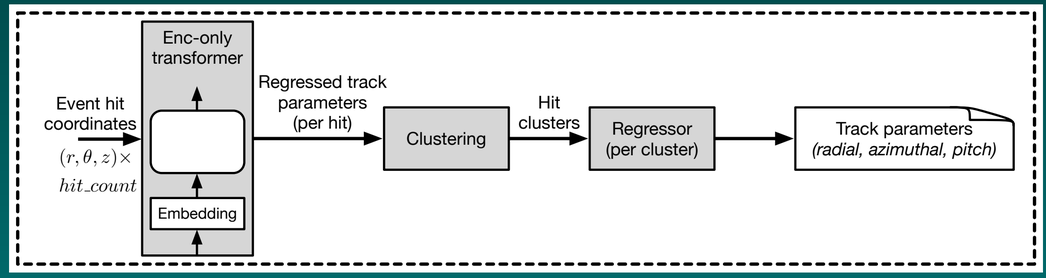
Considered solutions

Using Transformers - One-shot models

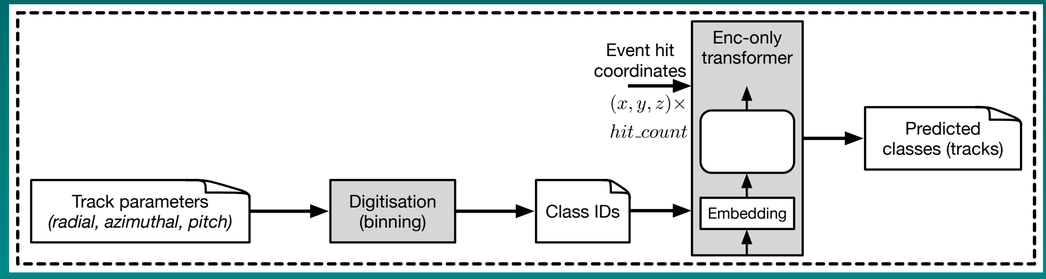
- Two models from the **TrackFormers** project (link to paper)
  => Both associate hits to tracks

- **EncReg** (Transformer - Encoder Regressor)
  => Track parameter regression + clustering (HDBSCAN)
  => Event-wide padding of data
  => HDBSCAN is costly

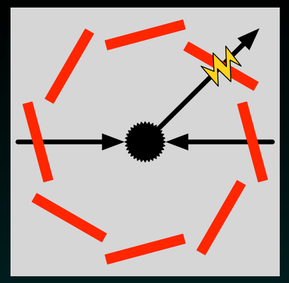
~1.5 Million parameters



- **EncCla** (Transformer - Encoder Classifier)
  => Binning of the detector space
  => Classification of hits
  => Batch-wide padding of data



Excluding Flash-Attention

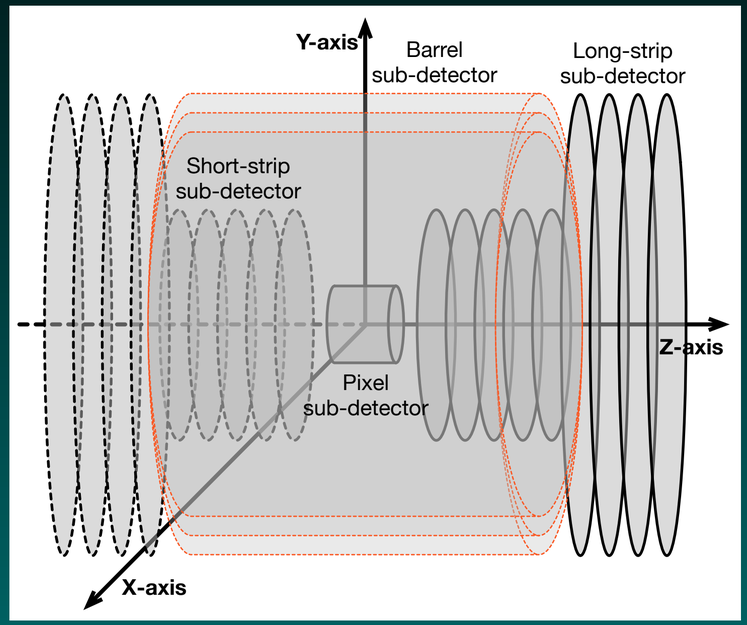


Considered data

- **REDuced VIrtual Detector (REDVID) data**
  => 10-50 (variable count) linear tracks per event
  => 10-50 (variable count) helical tracks per event
  => 50-100 (variable count) helical tracks per event

- **Reduced TrackML data**
  => 10-50 (variable count) tracks per event
  => 200-500 (variable count) tracks per event

- Each model design is trained with datasets
  => 5 trained models per design, 2 model sizes



Lots of randomisation for **non-determinism**

Complete list on website

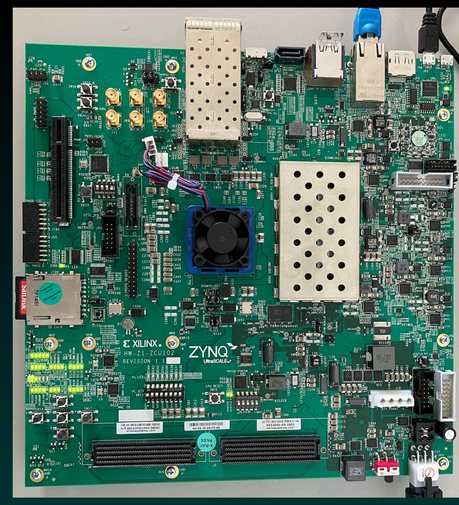
Performance

CPU-time, GPU-time

Training data	Model design	Inference CPU-side	Inference GPU-side
REDVID 10-50 linear tracks	EncCla	0.1 ms	4.0 ms
REDVID 10-50 helical tracks	EncCla	0.1 ms	4.1 ms
REDVID 50-100 helical tracks	EncCla	0.1 ms	4.3 ms
TrackML 10-50 tracks	EncCla	0.1 ms	7.0 ms
TrackML 200-500 tracks	EncCla	0.1 ms	7.0 ms
REDVID 10-50 linear tracks	EncReg	8.3 ms	2.4 ms
REDVID 10-50 helical tracks	EncReg	8.7 ms	2.3 ms
REDVID 50-100 helical tracks	EncReg	18.6 ms	4.1 ms
TrackML 10-50 tracks	EncReg	5.8 ms	2.2 ms
TrackML 200-500 tracks	EncReg	70.5 ms	31.9 ms

Deployment

A challenge on its own



- CPU
  => Not the platform of choice when it comes to ML
  => Pipelines have other steps with CPU as only option …

- **GPU**
  => Very good performance, highly parallel
  => Reliance on HPC assets, clusters, …

- FPGA
  => Very efficient, **low latency**, **low power**
  => Perfect for **embedded** and on-site deployments

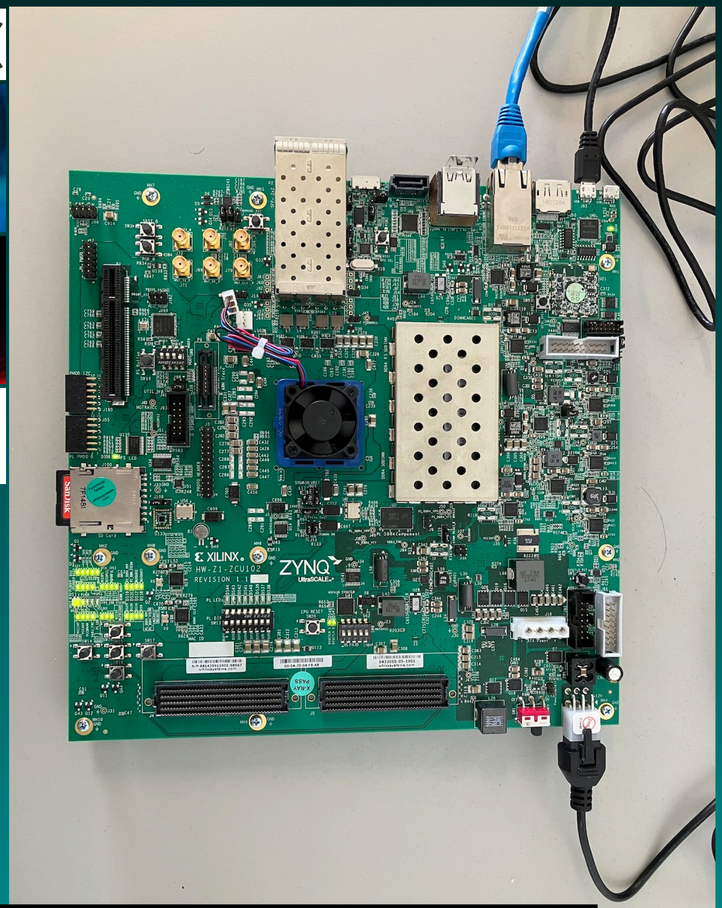
The ultimate goal -> Achieve online or pseudo-online tracking performance

FPGA board and tooling

- AMD Zynq UltraScale+ MPSoC
  => Xilinx is taken over by AMD
  => Hybrid platform (ARM core + FPGA)

- Tooling
  => **PyTorch** and **ONNX** for models
  => **AMD Vitis HLS** for kernel coding
  => **AMD Vivado HLS** for kernel synthesising
  => **PYNQ** for deployment to the board

- hls4ml? -> Transformers are not supported (yet)



Even if there is a tool, a blind synthesis does not involve proper analysis and optimisation.

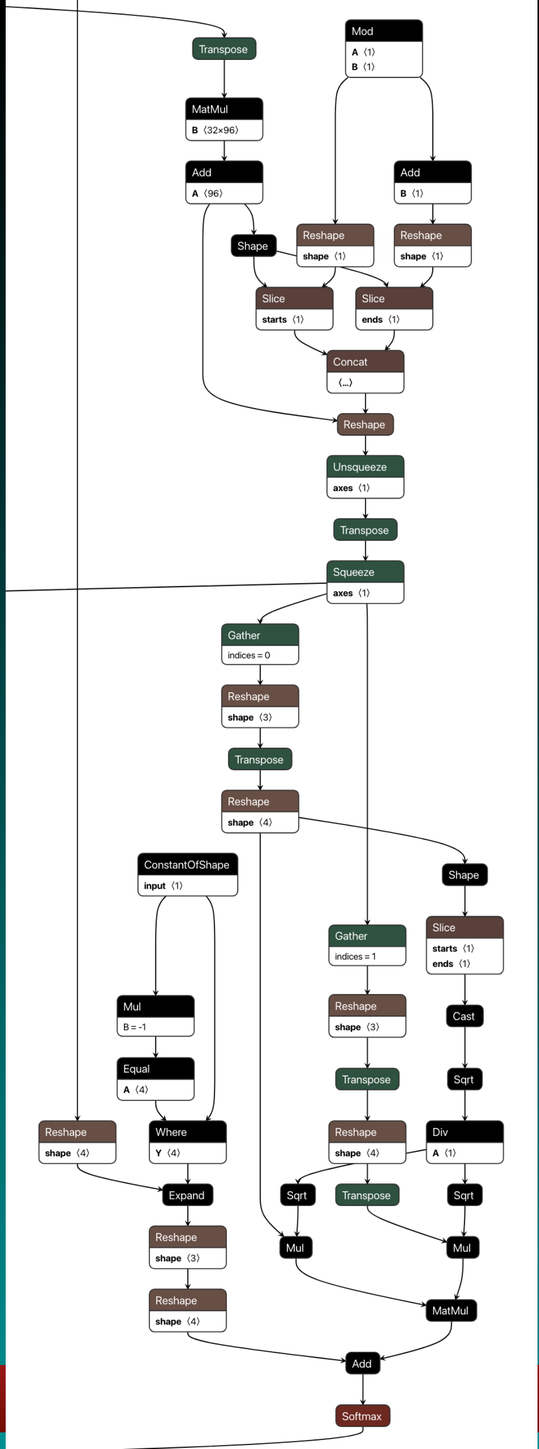
Conversion to ONNX

- PyTorch model format is not granular enough
  => Not enough visibility/individual access
  to inner structure

- **ONNX** (Open Neural Network Exchange)
  => Open-source format for representing ML models
  => Input dimensions is still required
  => Weights, dimensions, data types, …

- A **granular model graph** can be extracted

Supported by PyTorch -> Model + Input shape



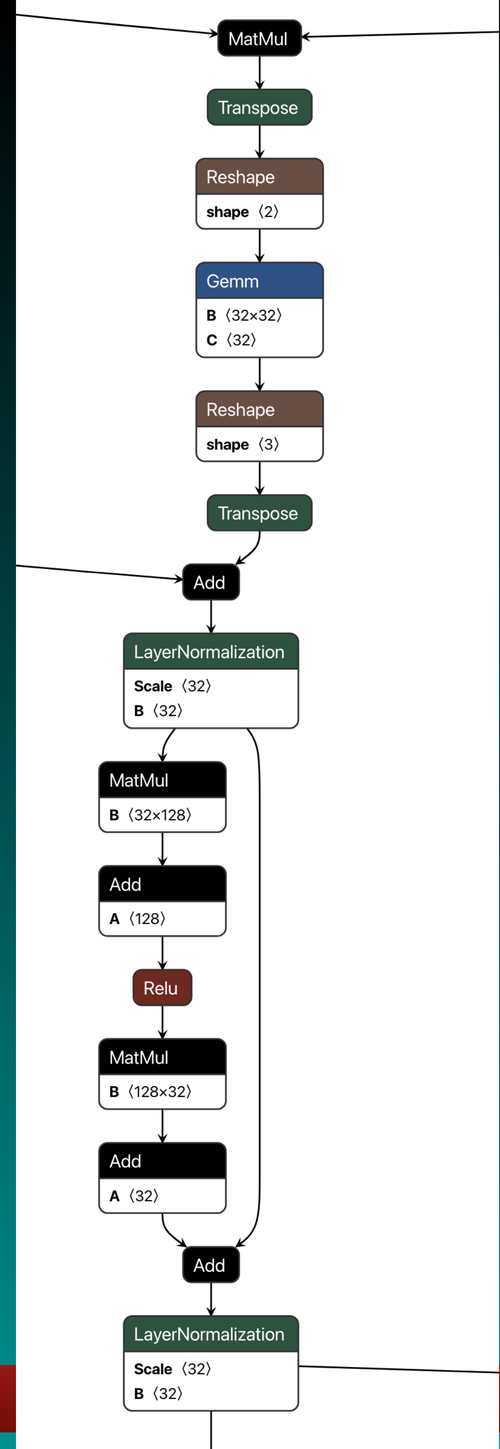
Development strategy

- Single computation steps
  => The most basic you can think of is MatMul

- Followed by sequential slices
  => There is no need to implement individual steps
  => These can be a **combined monolithic kernel**

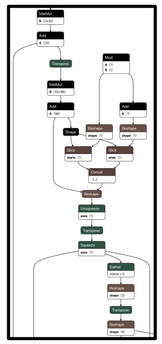
- Partial FPGA deployment (**hybrid**)
  => Repetitive slices (reusable kernel)
  => **Data communication** back and forth (costly)
  => Would be advantageous at scale (**instances in parallel**)

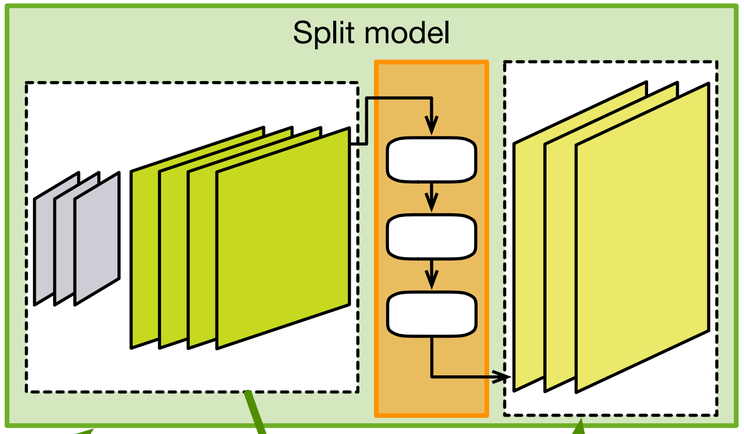
The ultimate goal -> Full model deployment (?)



Development flow

Based on ONNX model graph





ONNX

PyTorch model

ONNX model

Model graph

Computation step(s)

Model converter

Model grapher

Manual slice selection

Model splitter

Fuse dataflow

Kernel input

Kernel output

Coding for FPGA (HLS)

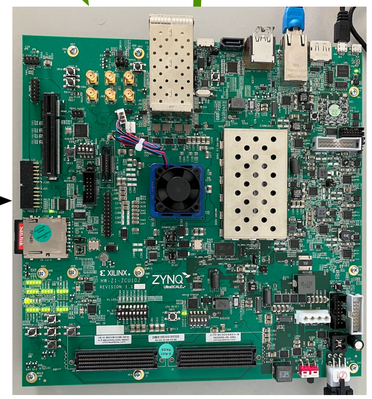
AMD Vitis

Synthesise kernel

AMD Vivado

Deploy

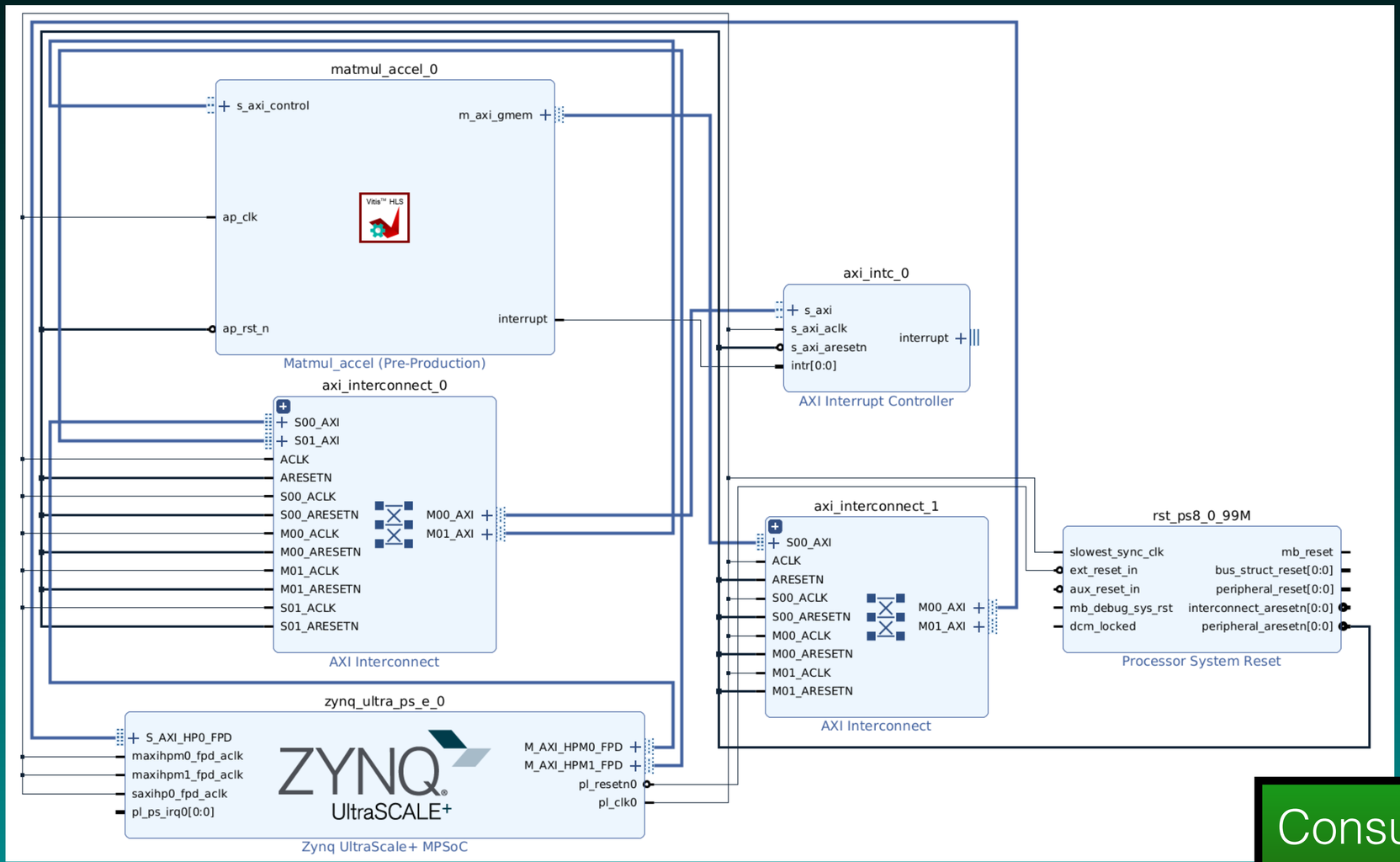
PYNQ™

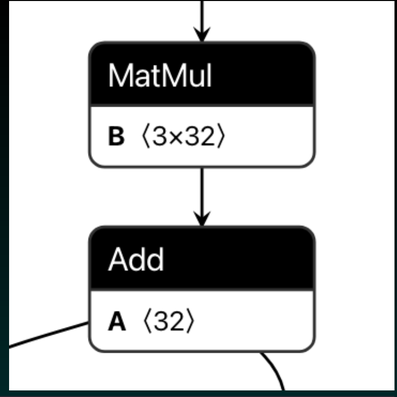


Computation kernel

Synthesised kernel

Synthesised kernel






```
11  void matmul_accel(const float* A, const float* B, float* C) {
12      #pragma HLS INTERFACE m_axi port=A offset=slave
13      #pragma HLS INTERFACE m_axi port=B offset=slave
14      #pragma HLS INTERFACE m_axi port=C offset=slave
15
16      #pragma HLS INTERFACE s_axilite port=return
17
18      float a_local[A_ROWS][A_COLS];
19      float b_local[A_COLS][B_COLS];
20
21      #pragma HLS ARRAY_PARTITION variable=b_local complete dim=1
22
23      // Load A into local buffer
24      for (int i = 0; i < A_ROWS; i++) {
25          for (int j = 0; j < A_COLS; j++) {
26              #pragma HLS PIPELINE
27              a_local[i][j] = A[i * A_COLS + j];
28          }
29      }
30
31      // Load B into local buffer
32      for (int i = 0; i < A_COLS; i++) {
33          for (int j = 0; j < B_COLS; j++) {
34              #pragma HLS PIPELINE
35              b_local[i][j] = B[i * B_COLS + j];
36          }
37      }
38
39      // Matrix multiply
40      for (int i = 0; i < A_ROWS; i++) {
41          for (int j = 0; j < B_COLS; j++) {
42              #pragma HLS PIPELINE
43              float sum = 0;
44              for (int k = 0; k < A_COLS; k++) {
45                  sum += a_local[i][k] * b_local[k][j];
46              }
47              C[i * B_COLS + j] = sum;
48          }
```

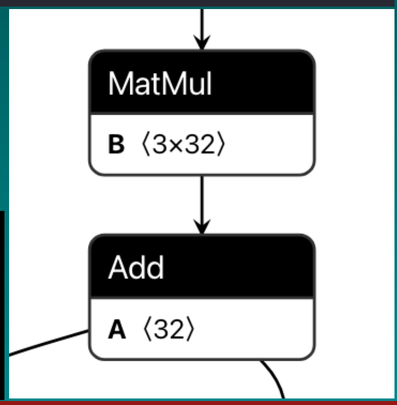
Consumes ~2.1% of Configurable Logic Blocks (CLBs)

Impact analysis

Floating-point precision

- ML models -> Floating-point arithmetics
- CPUs and GPUs
  => Follow **IEEE-754 standard** single (`float32`) or double (`float64`) precision
- FPGAs
  => May use different standards
  (IEEE-754, reduced precision, half-precision, fixed-point)
  => Customised arithmetic pipelines
  => Rounding can cause numerical drift
  => Toolchain-specific optimisations

```
11  void matmul_accel(const float* A, const float* B, float* C) {
12      #pragma HLS INTERFACE m_axi port=A offset=slave
13      #pragma HLS INTERFACE m_axi port=B offset=slave
14      #pragma HLS INTERFACE m_axi port=C offset=slave
15
16      #pragma HLS INTERFACE s_axilite port=return
17
18      float a_local[A_ROWS][A_COLS];
19      float b_local[A_COLS][B_COLS];
20
21      #pragma HLS ARRAY_PARTITION variable=b_local complete dim=1
22
23      // Load A into local buffer
24      for (int i = 0; i < A_ROWS; i++) {
25          for (int j = 0; j < A_COLS; j++) {
26              #pragma HLS PIPELINE
27              a_local[i][j] = A[i * A_COLS + j];
28          }
29      }
30
31      // Load B into local buffer
32      for (int i = 0; i < A_COLS; i++) {
33          for (int j = 0; j < B_COLS; j++) {
34              #pragma HLS PIPELINE
35              b_local[i][j] = B[i * B_COLS + j];
36          }
37      }
38
39      // Matrix multiply
40      for (int i = 0; i < A_ROWS; i++) {
41          for (int j = 0; j < B_COLS; j++) {
42              #pragma HLS PIPELINE
43              float sum = 0;
44              for (int k = 0; k < A_COLS; k++) {
45                  sum += a_local[i][k] * b_local[k][j];
46              }
47              C[i * B_COLS + j] = sum;
48          }
49      }
50  }
51
52  }
```

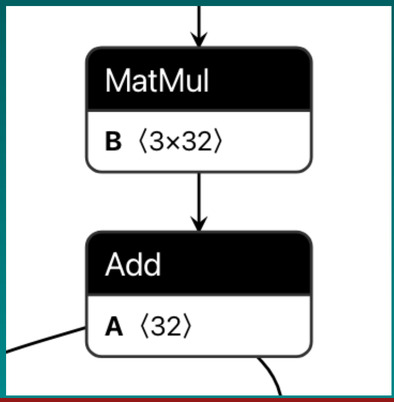


Attached memory on our device is: `float16`

Quantisation

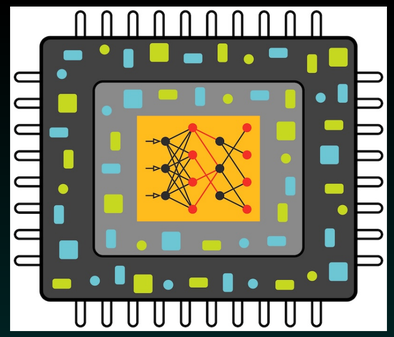
- Quantisation: Reduce the precision of model parameters (such as weights and activations), making models smaller, faster, and more energy-efficient

- Types:
  => **Post-Training Quantisation (PTQ)**
  (convert a pre-trained model to a lower precision format)
  => Quantisation-Aware Training (QAT)
  => Dynamic Quantisation
  => Static Quantisation



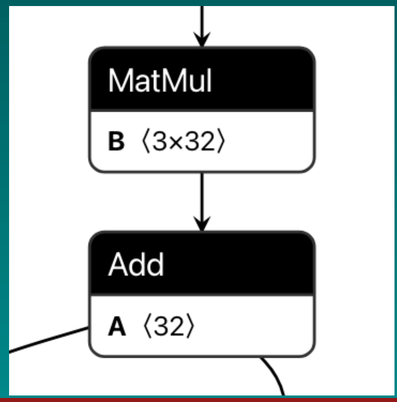
Impact analysis

In conjunction with quantisation



- Quantisation reduces precision
  => Full quantisation? Usually not the case …
  => Partial quantisation -> FPGA FP precision matters (baseline)
  => **Different layers have different sensitivities**

- All of this means:
  => There is a large search-space for an effective implementation
  => Hard to find a universal approach
  => Based on use-case or Transformer design style



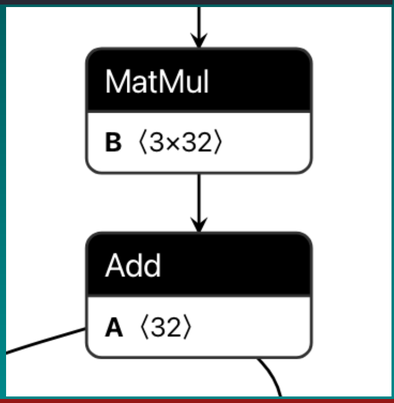
Impact analysis

Output comparison

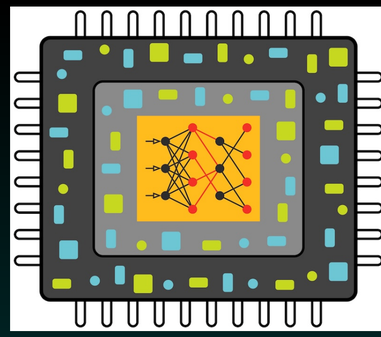
```
Instance 0:

    [MatMul output]
        Mismatches: 3876 / 16000 (24.23%)
        Min Diff:   0.00000000
        Max Diff:   0.00000048
        Mean Diff:  0.00000002
        Std Dev:    0.00000006

    [MatMul output - Ignoring sequence padding]
        Mismatches: 3876 / 11200 (34.61%)
        Min Diff:   0.00000000
        Max Diff:   0.00000048
        Mean Diff:  0.00000003
        Std Dev:    0.00000007

    [Model output]
        Mismatches: 1432 / 1500 (95.47%)
        Min Diff:   0.00000000
        Max Diff:   0.00000755
        Mean Diff:  0.00000074
        Std Dev:    0.00000062

    [Model output - Ignoring sequence padding]
        Mismatches: 982 / 1050 (93.52%)
        Min Diff:   0.00000000
        Max Diff:   0.00000755
        Mean Diff:  0.00000064
        Std Dev:    0.00000069
```

- **ARM CPU vs FPGA**:
  => MatMul step output
  => MatMul step output ignoring **seq. padding**
  => Model output (with MatMul in the loop)
  => Model output ignoring seq. padding

- What does it mean?
  => Changes in one step's result will have **cascading effects**
  => Sensitivity to precision? -> Partial deployment

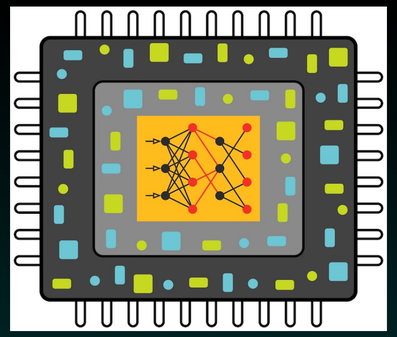


Optimisations to consider



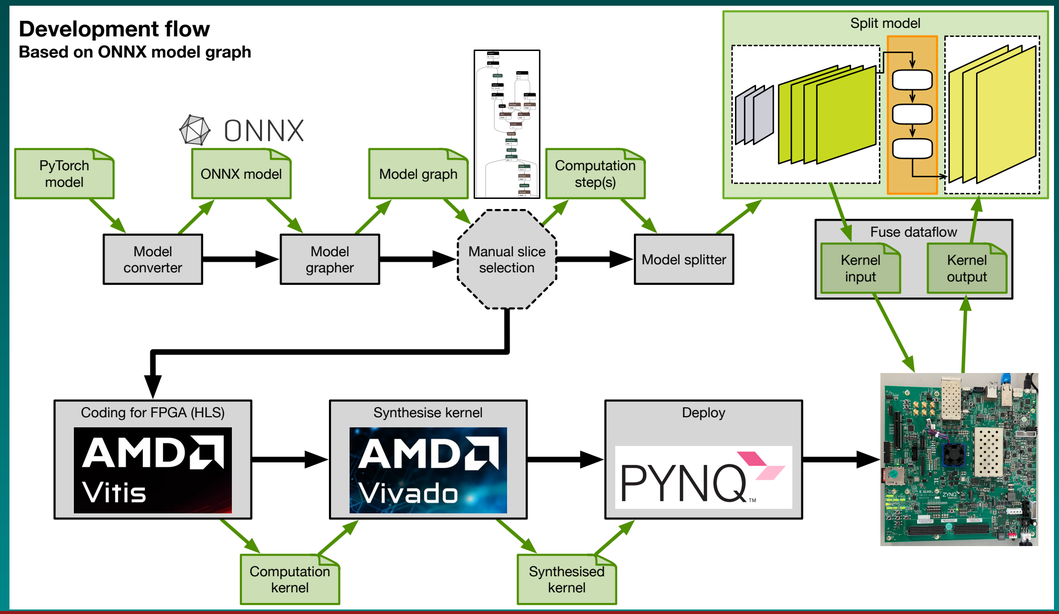
- Which computation steps are to be deployed on FPGA?
  => Full model? Partial, hybrid?
- Which computation steps are sensitive to FP precision loss?
  => What is the impact on final hit associations?
- Added cost of data communication vs computation gains? (hybrid)
- Generality?
  => Different model sizes? Different Transformer designs?

Ongoing work



- Focus on repetitive model segments (Encoder layers)
  => **Low-level implementation** (NumPy) ✅
- Experiment with Post-Training Quantisation
  => Full? Partial?
- Hardware memory management
  => **DMA processor**
- Automation …
  => Slice selection, impact analysis

Extensive benchmarking:
- Latency
- CLB utilisation
- Energy
- Tracking performance



Thanks!
Questions?