NNs on Hardware

An AIE implementation of the GRU

Michail Sapkas - University/INFN Padova

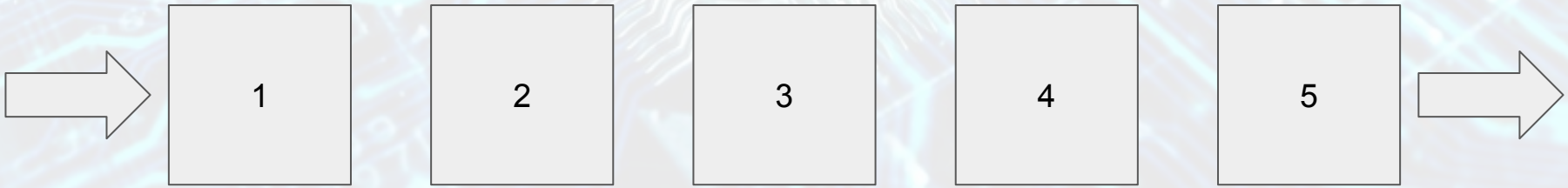
RL4AA Workshop

30 Mar - 1 Apr 2026, Liverpool

FPGAs

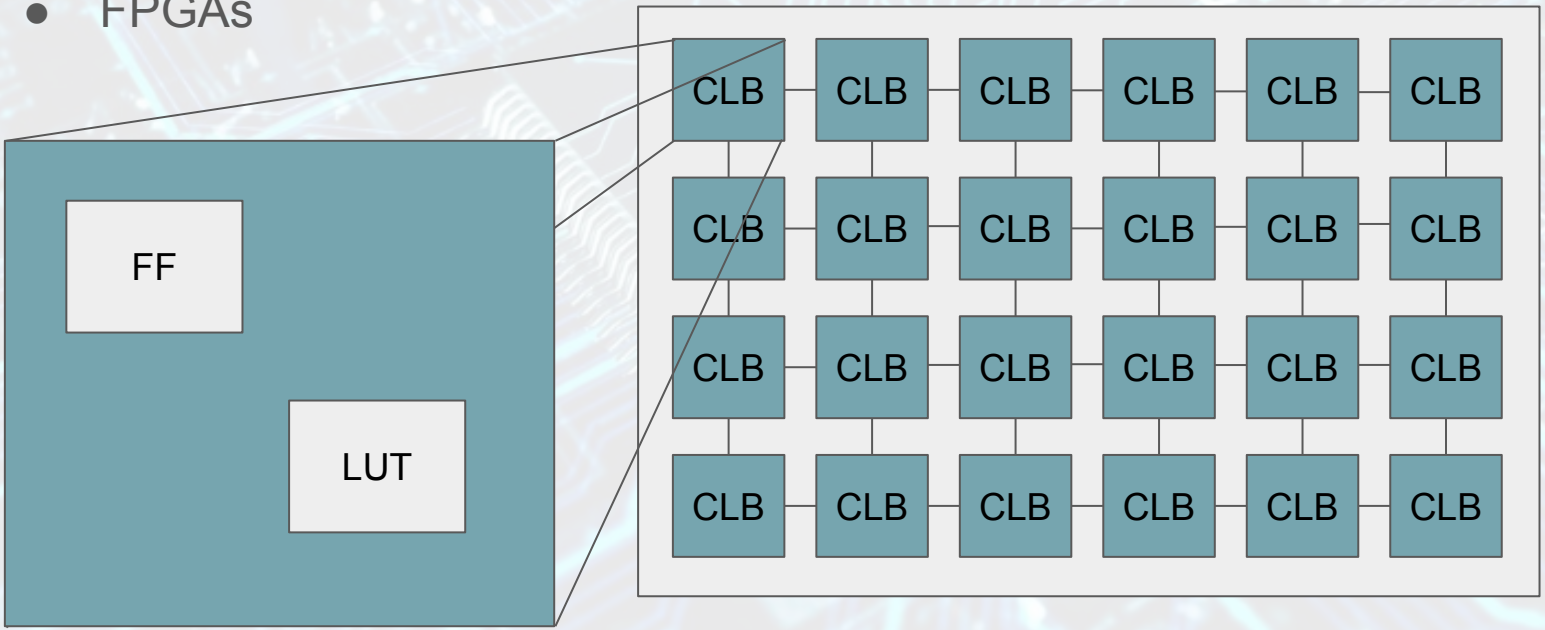
- **Field Programmable Gate Arrays**
- An array (2D) of gates
- Basic blocks of “Configurable Logic Blocks” - CLBs
- That are re-programmable

PIPELINE

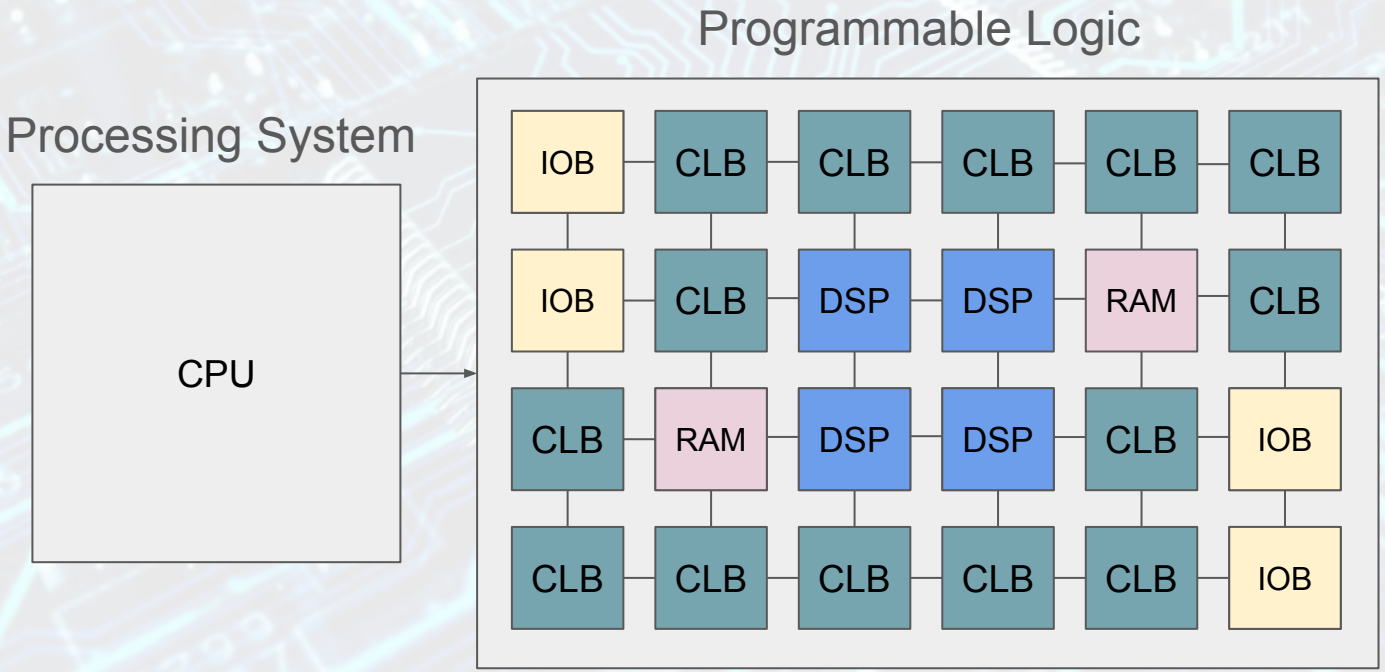


Inside an FPGA

- FPGAs



Modern FPGAs - System on Chip



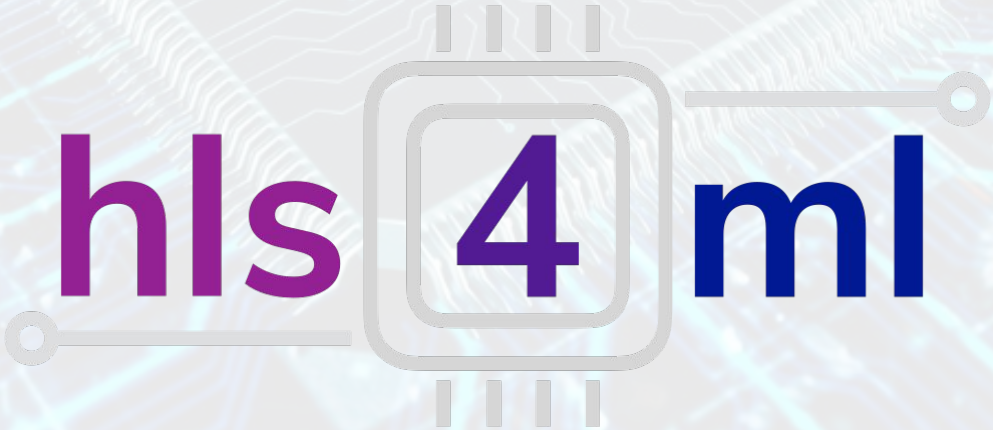
How to Program?

- Hardware Description Languages
  - VHDL
  - Verilog

- High Level Synthesis
  - C++ “like” that compiles to VHDL / Verilog
  - Heavy use of directives
  - Still need to know how HDL works

FPGAs are great hardware for low-latency

- How can we deploy Neural Networks in FPGAs?
- I am using:



“Catch”

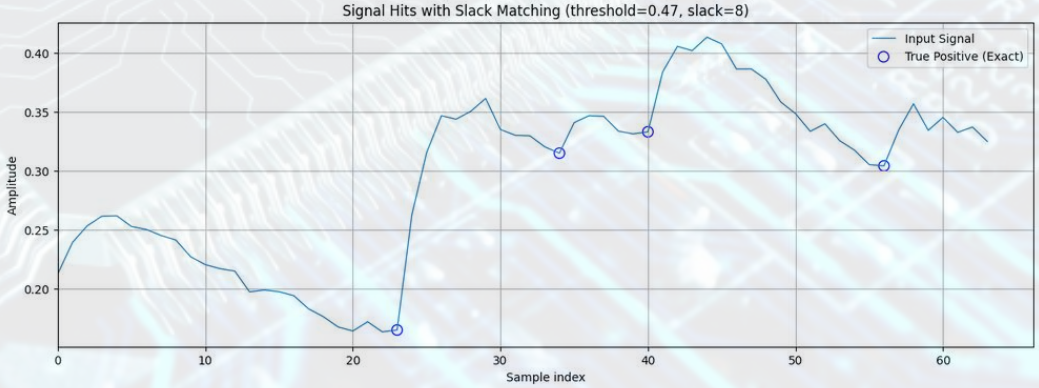
- In FPGAs we are using quantized data types
- And this is the key of being fast
- FPGA resources are very fast exhausted with model size
- So FPGAs can accommodate only “**small**” and **quantized** models

- So techniques exist to either **take a big model and make it smaller**
  - Regularization
  - Pruning
- And to quantize data types
  - post-training quantization
  - quantization aware training

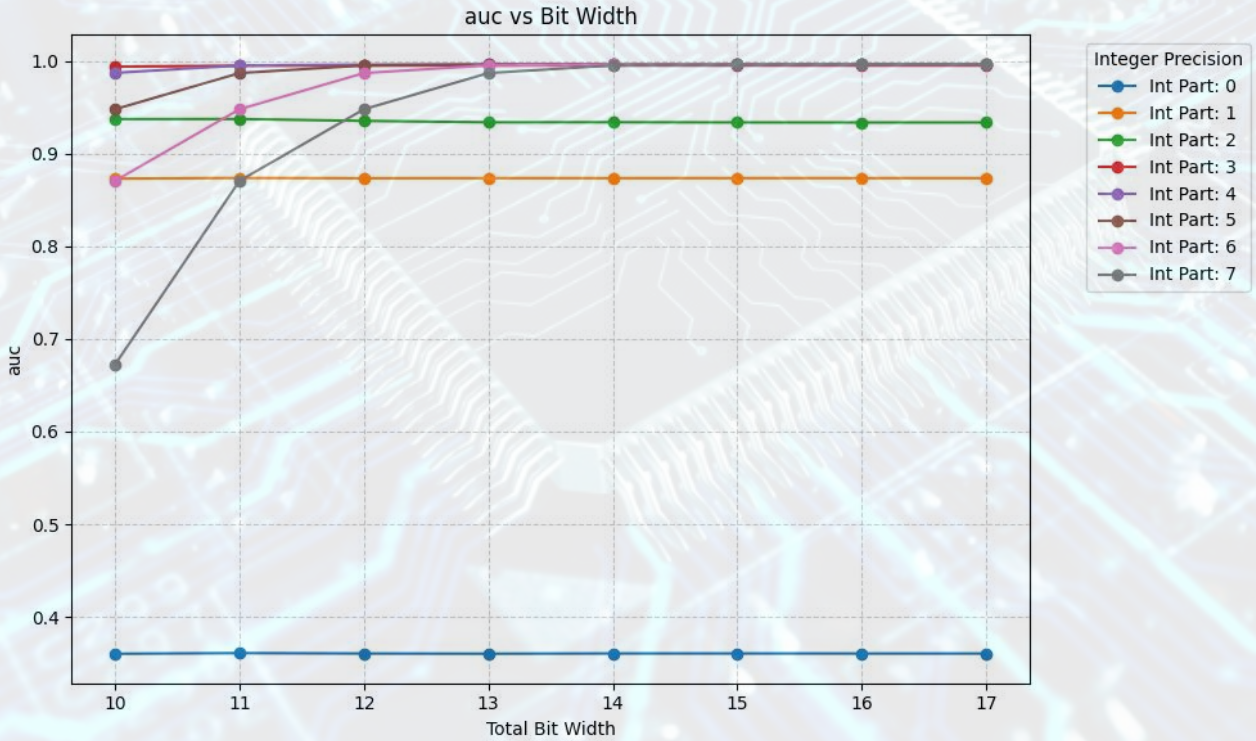
Example: real time 1D CNN in Waveform

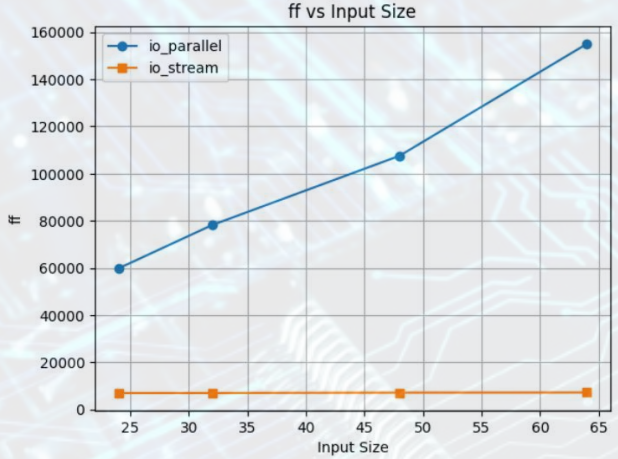
```
Model: "peak_det"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_layer (InputLayer)    [(None, 64, 1)]           0

 conv1 (Conv1D)              (None, 64, 4)             44

 pool1 (MaxPooling1D)        (None, 32, 4)             0

 conv2 (Conv1D)              (None, 32, 12)            204

 upsampl2 (UpSampling1D)     (None, 64, 12)            0

 conv4 (Conv1D)              (None, 64, 1)             121

=================================================================
Total params: 369 (1.44 KB)
Trainable params: 369 (1.44 KB)
Non-trainable params: 0 (0.00 Byte)
```

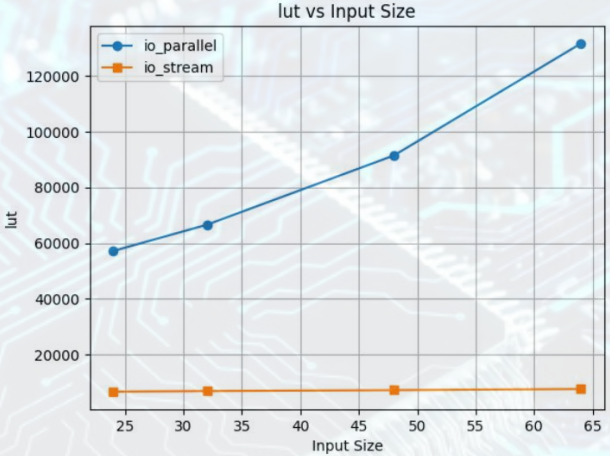
```
TP exact: 4
TP slack-corrected: 0
False Negatives (FN): 0
False Positives (FP): 0
Precision: 1.0000
Recall: 1.0000
F1 Score: 1.0000
(64, 1)
```

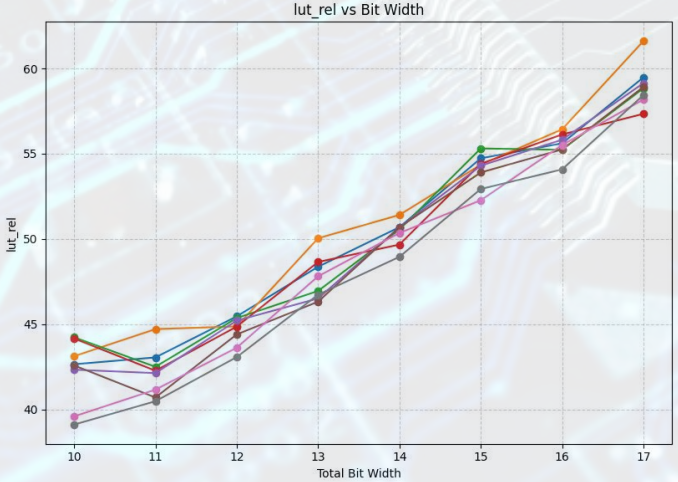


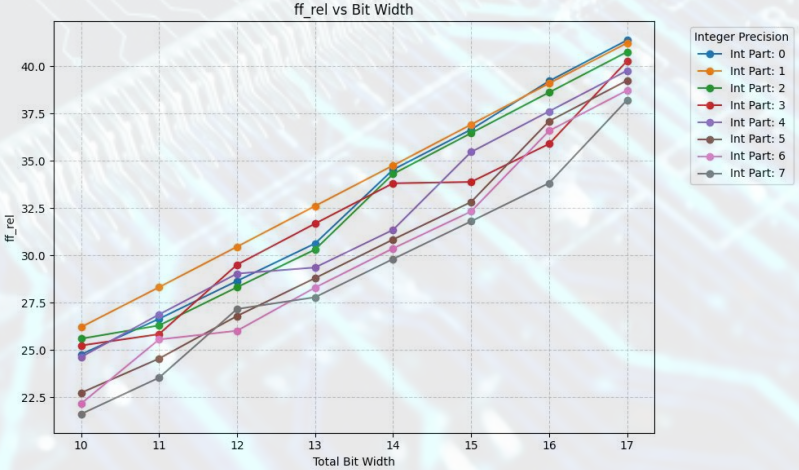
And pick a data type based on ROC - AUC

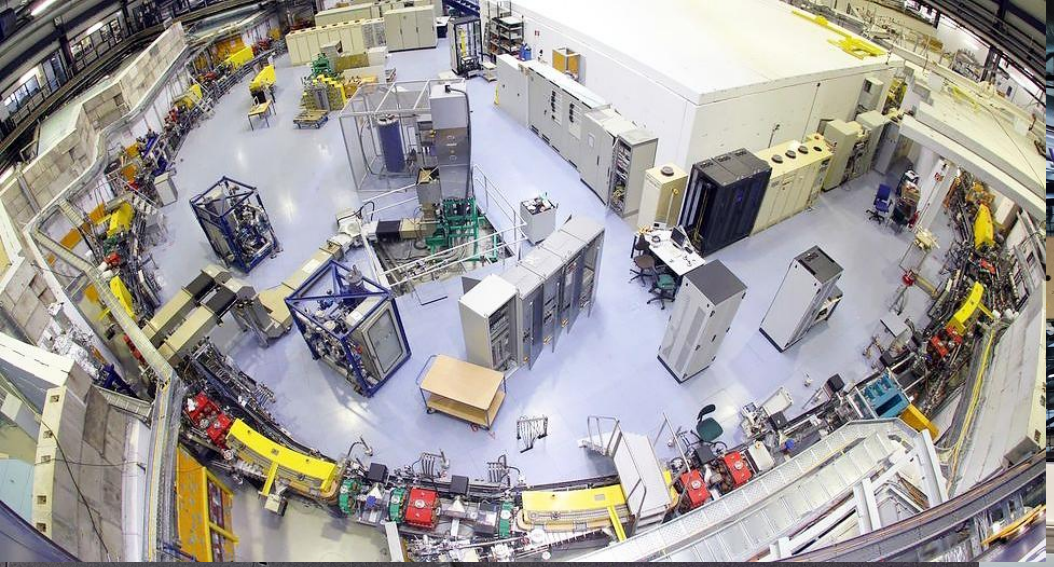



ff vs Input Size
io_parallel
io_stream
ff
Input Size

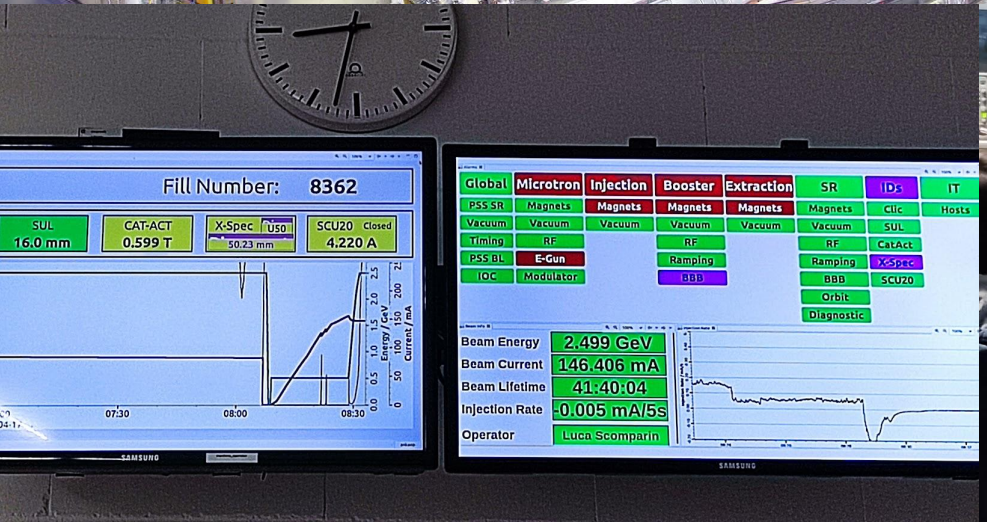

lut vs Input Size
io_parallel
io_stream
lut
Input Size


lut_rel vs Bit Width
lut_rel
Total Bit Width

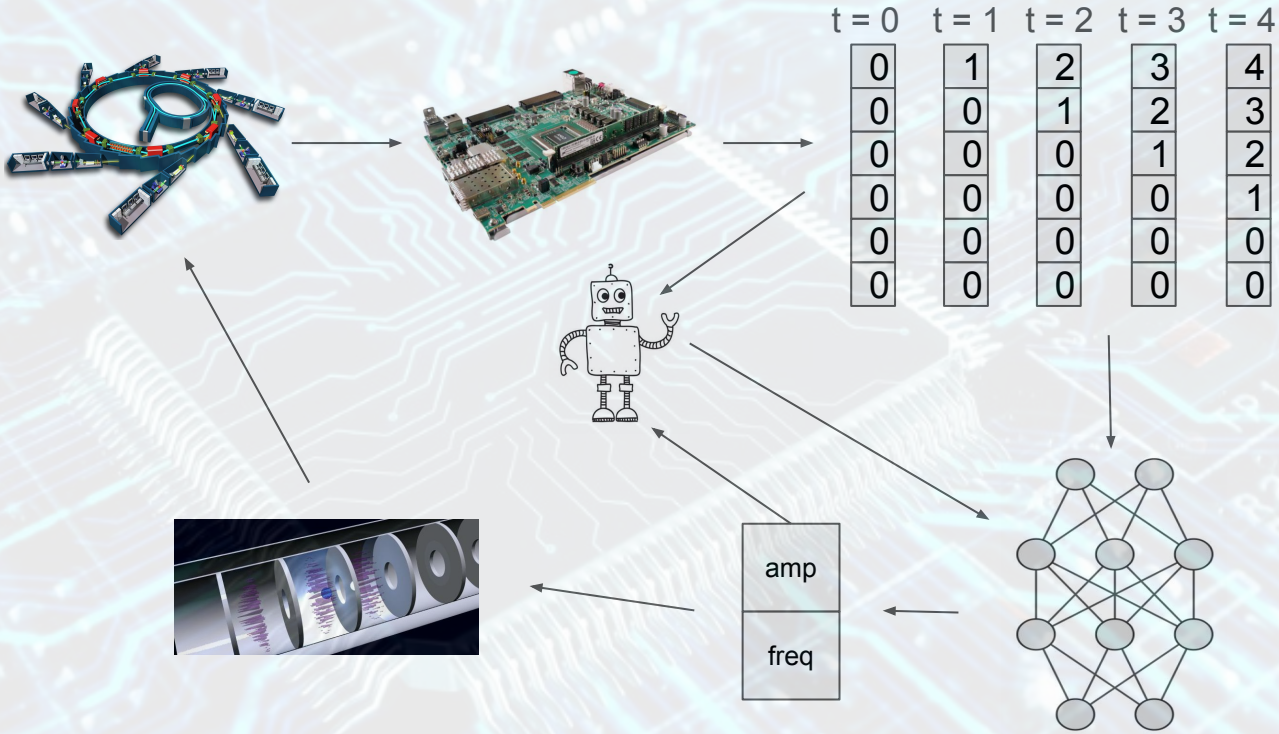

ff_rel vs Bit Width
ff_rel
Total Bit Width
Integer Precision
Int Part: 0
Int Part: 1
Int Part: 2
Int Part: 3
Int Part: 4
Int Part: 5
Int Part: 6
Int Part: 7




SPERRBEREICH
-KEIN ZUTRITT-


Fill Number: 8362
SUL
16.0 mm
CAT-ACT
0.599 T
X-Spec
50.23 mm
SCU20
Closed
4.220 A
Global
Microtron
Injection
Booster
Extraction
SR
IDs
IT
PSS SR
Magnets
Vacuum
Timing
PSS BL
IOC
RF
E-Gun
Modulator
Ramping
BBB
Orbit
Diagnostic
CLIC
SUL
CatAct
Hosts
Beam Energy
2.499 GeV
Beam Current
146.406 mA
Beam Lifetime
41:40:04
Injection Rate
-0.005 mA/5s
Operator
Luca Scomparin
SAMSUNG

Motivation:

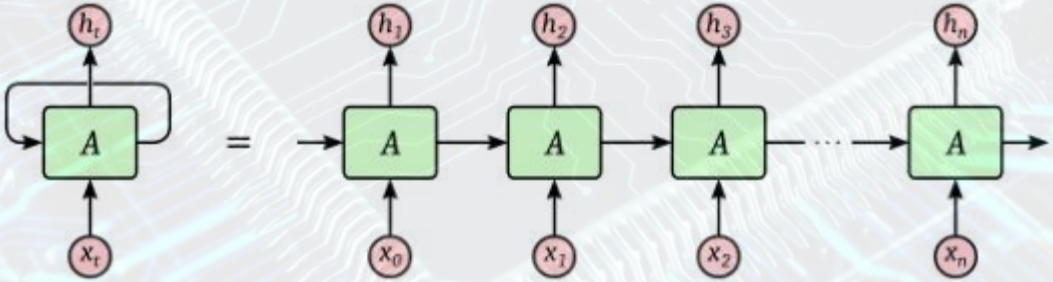


We need:

- To be faster than the timescale of the physics ~ tens of μs
- Our model must be sophisticated enough to capture time correlations and keep memory of previous modulations

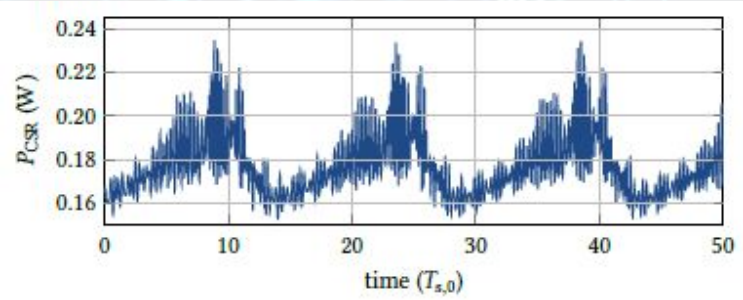
The Gated Recurrent Unit (LSTMs little cousin)

- This is the “temporal unrolled view” for RNNs



- The “hidden state” is an encoding of observations through time, some may even call it “memory”. At each time step (or after many), it can be fed to a Feed Forward NN to do regression or classification.

Unfortunately, RNNs are probably the worst models to implement in Hardware



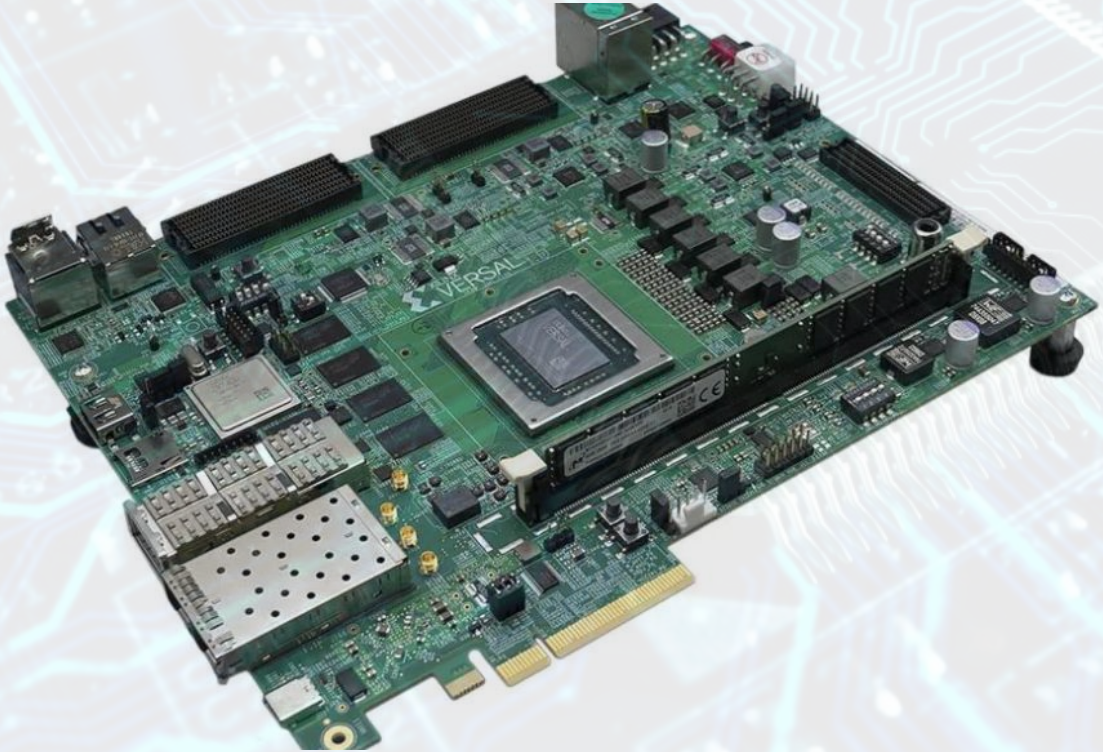
N - temporal window

GRU

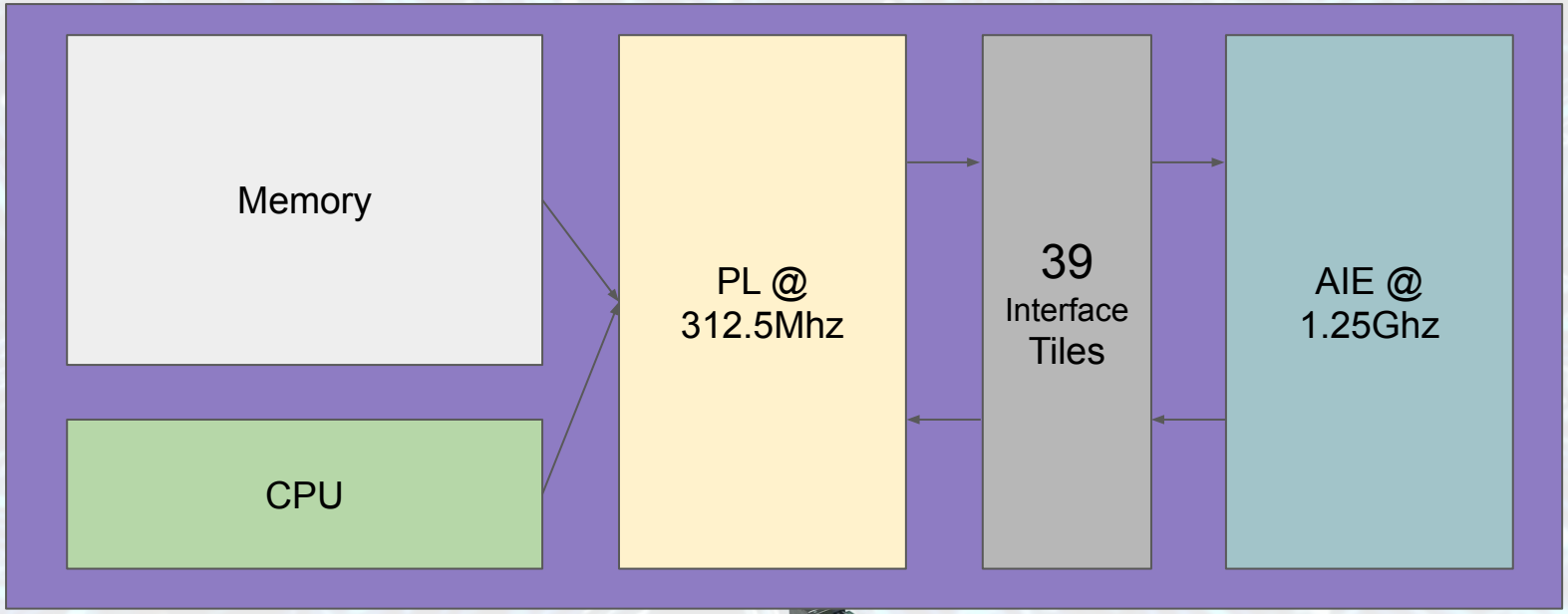
amp

freq

The Versal VCK190



The benefits of printing smaller and smaller



AN FPGA ON STEROIDS BASICALLY

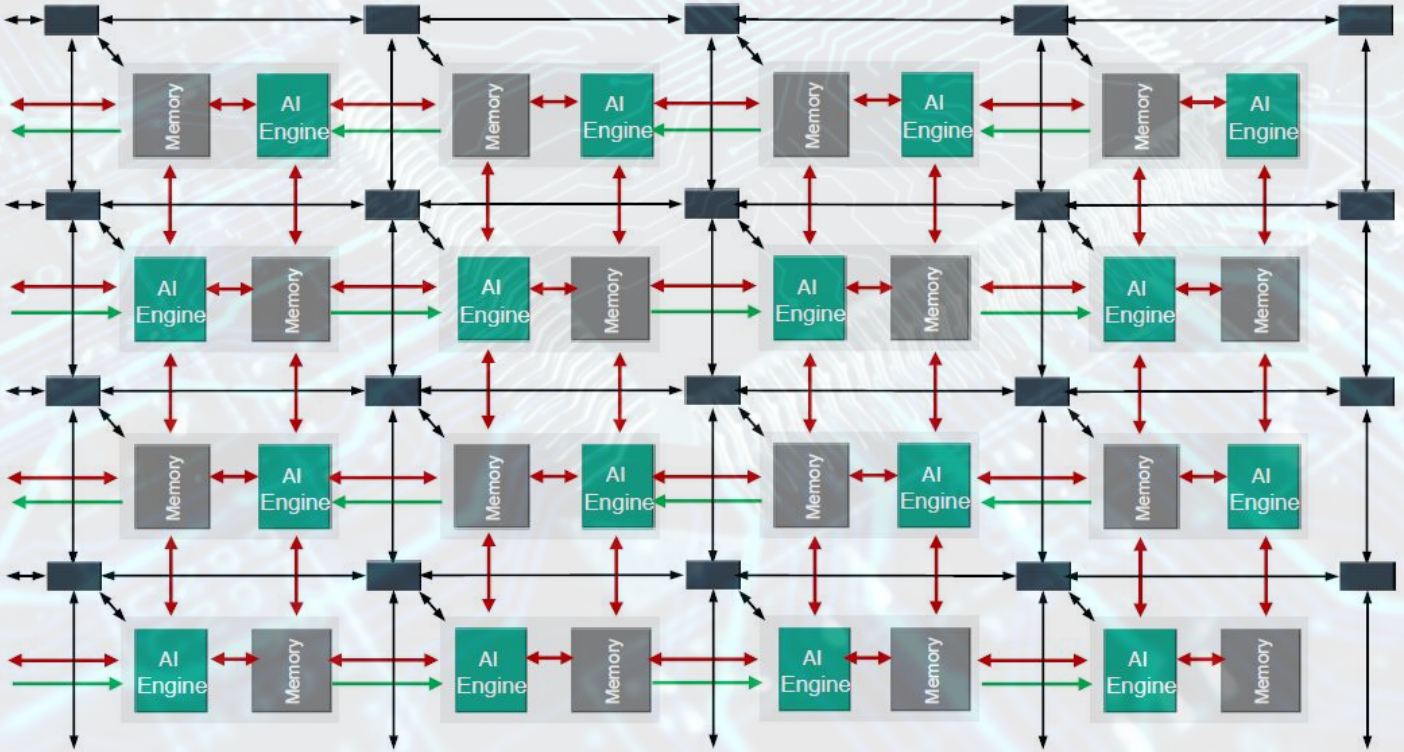
Memory

CPU

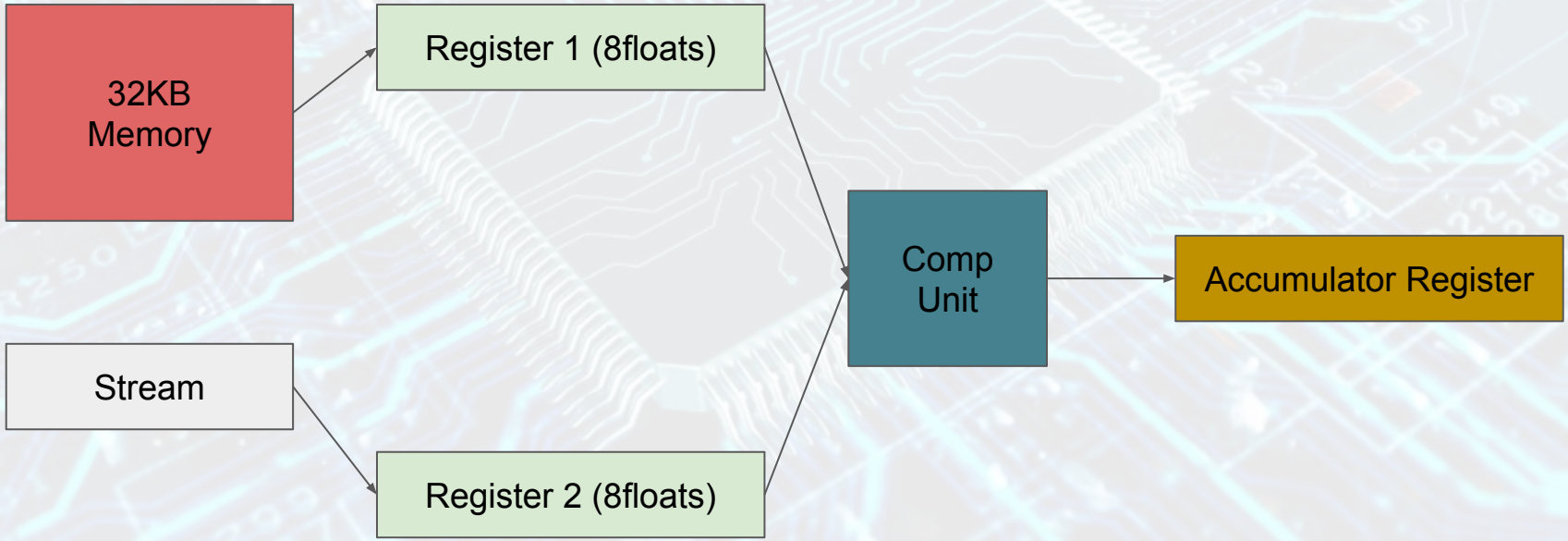
AIE @ 1.25Ghz

The Versal AIE

- 400 Vector Processors with local memory



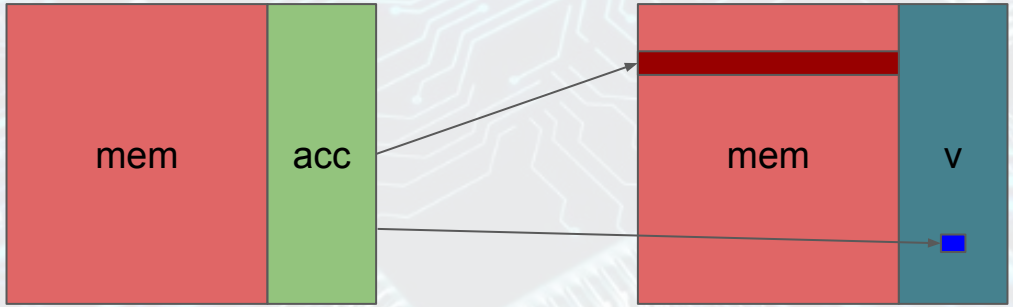
A (very) simplified insides of a AIE Tile

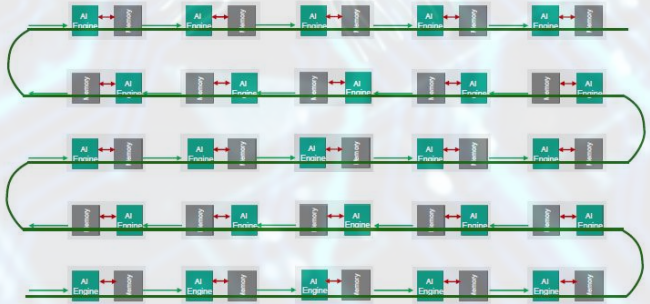


Tile - Tile Connectivity: Cascade Stream (Local)

- Only neighbouring tiles (right for even, left for odds, up for edge)

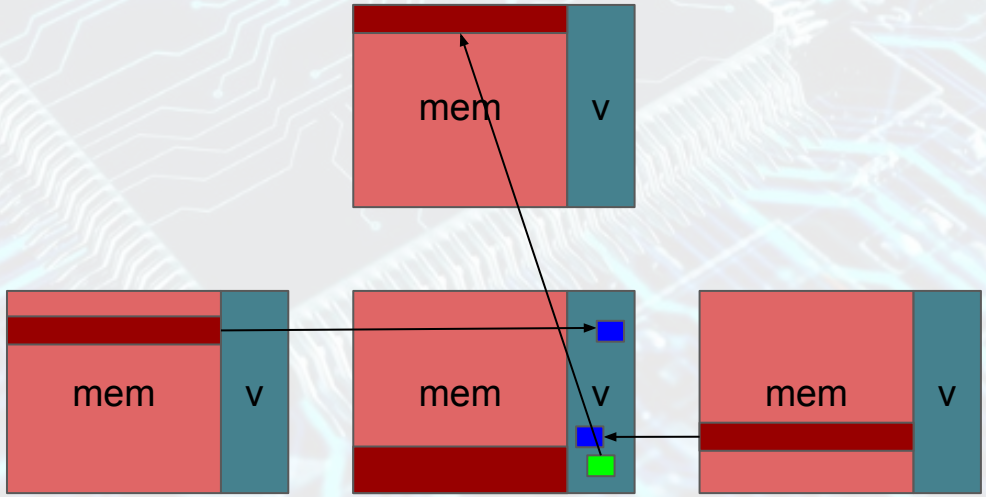
Accumulator





Tile - Tile Connectivity: Memory Sharing (Local)

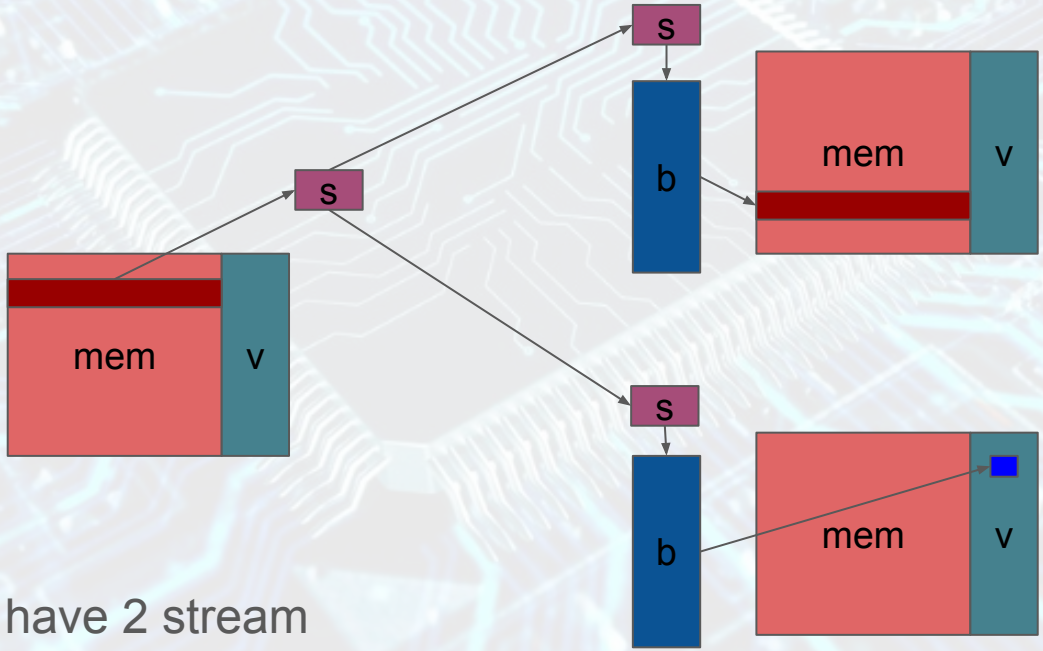
- Block of Memory banks
- Stored data
- Vectorized Processor
- VecProc Input Port
- VecProc Output Port



Tile - Tile Connectivity: AXI4 Stream Interconnect

Switch

Buffers



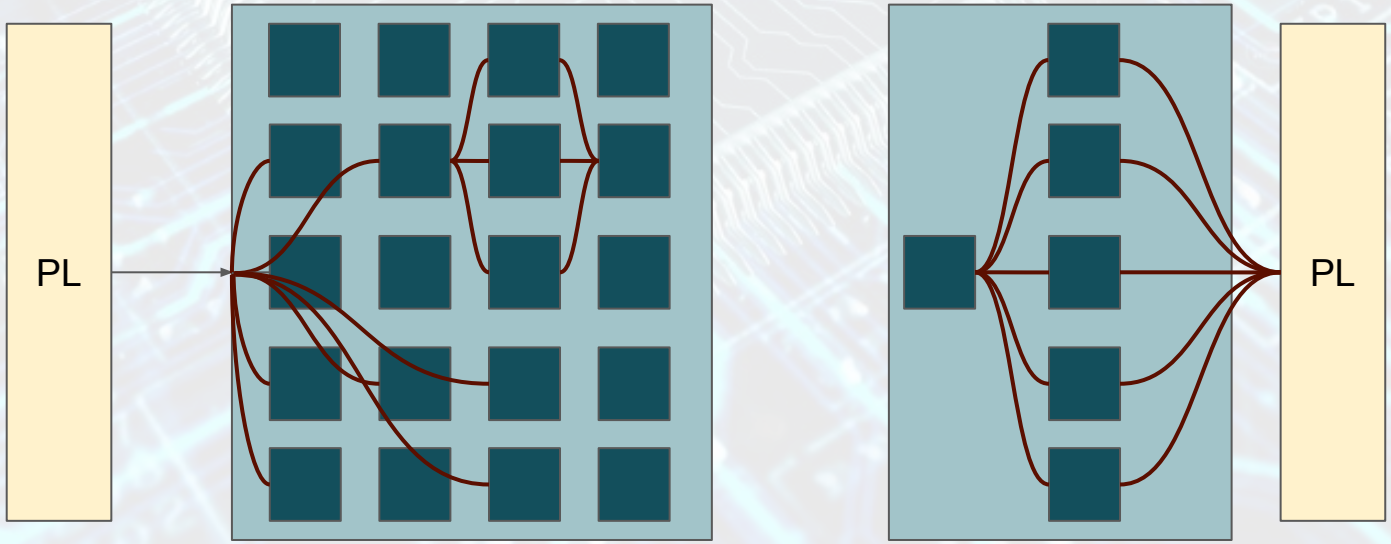
- Each tile can have 2 stream inputs and 2 stream outputs

Packet Stream (split - merge)

- To use the same physical AXI4Stream for multiple source or destination
- Headers can either be sent from PL targeting a AIE Tile or from Tile to Tile

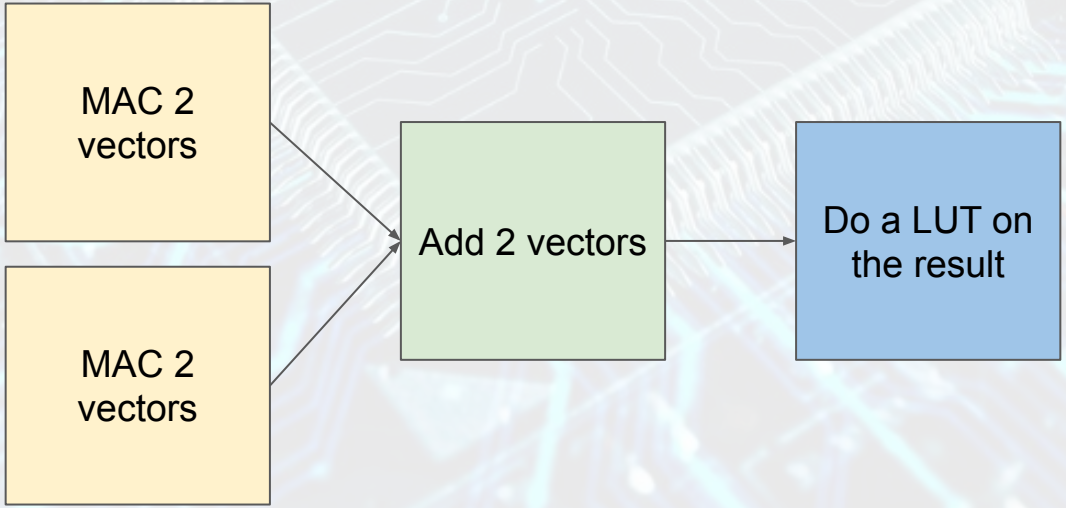
AIE Tile

Packet Stream Routings over AXI4Stream



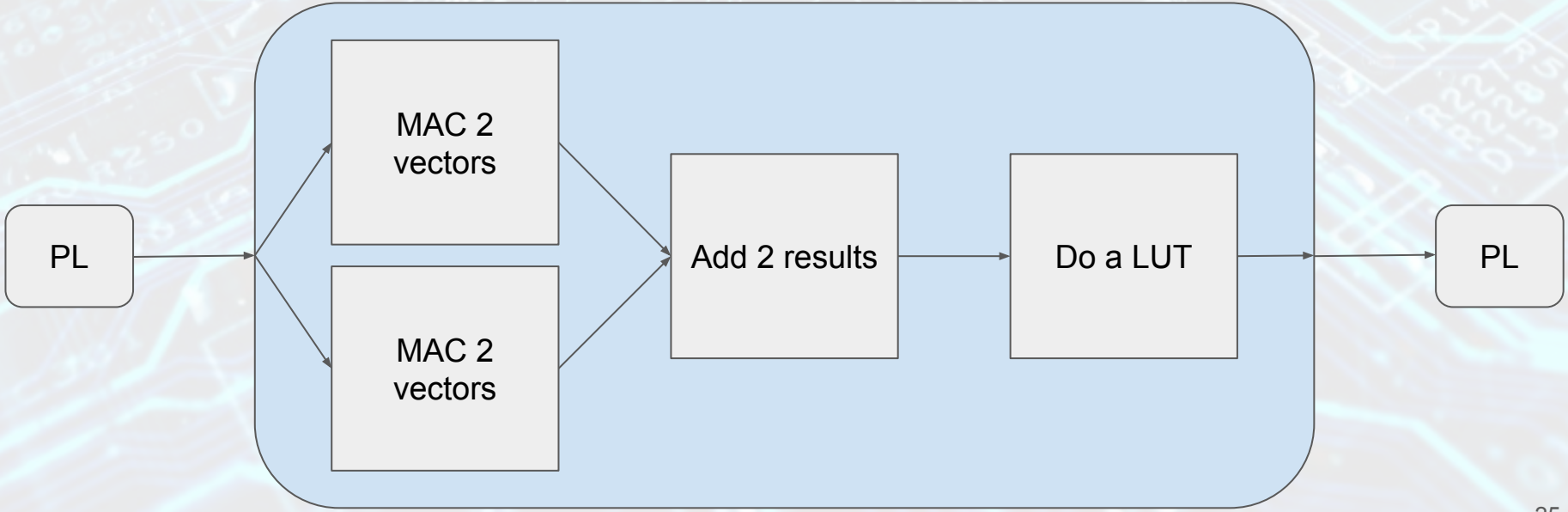
Programing paradigm

- You start with writing the program of a single tile called “kernel”
- A kernel is a function in C++ that also declares connectivity
- Chess compiler pragmas are essential

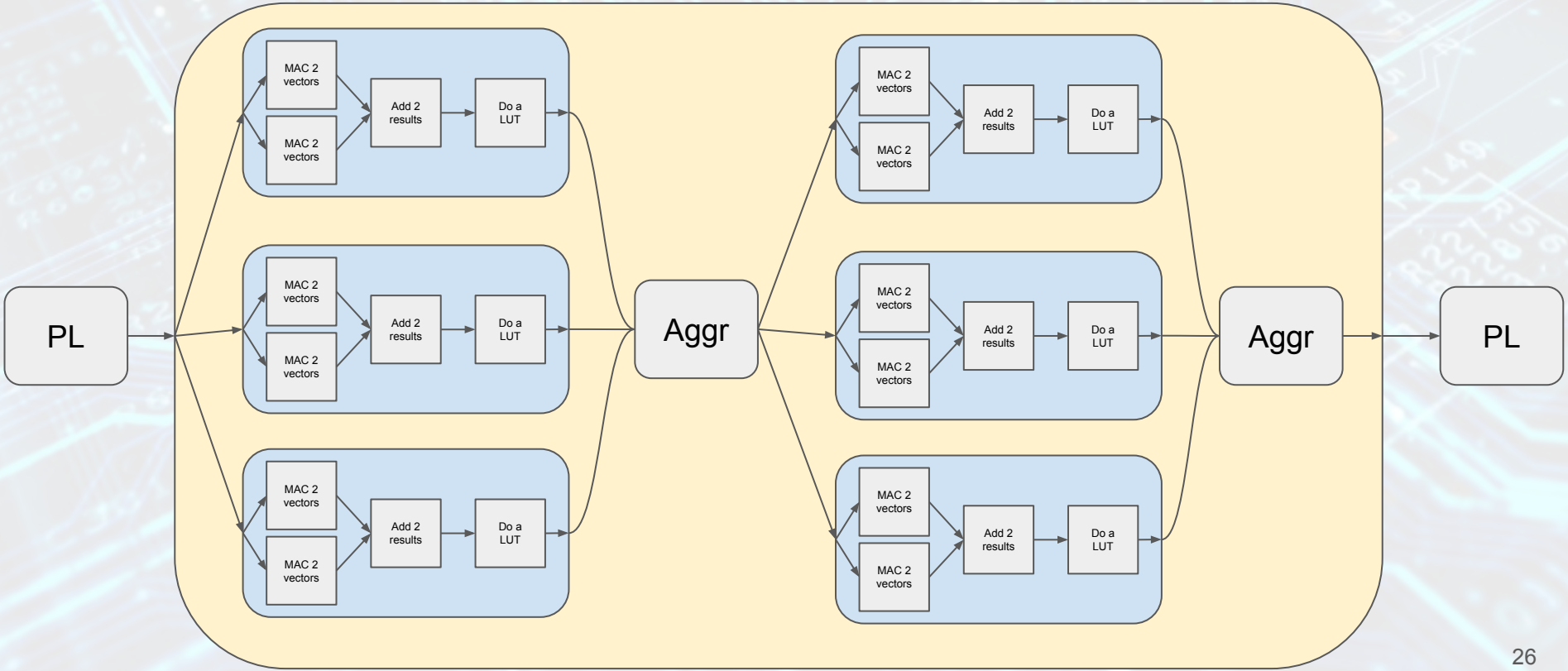


Organize multiple kernels into graphs

- Computational graphs are the top level abstractions that allow you to organize the execution of multiple kernels to obtain a result

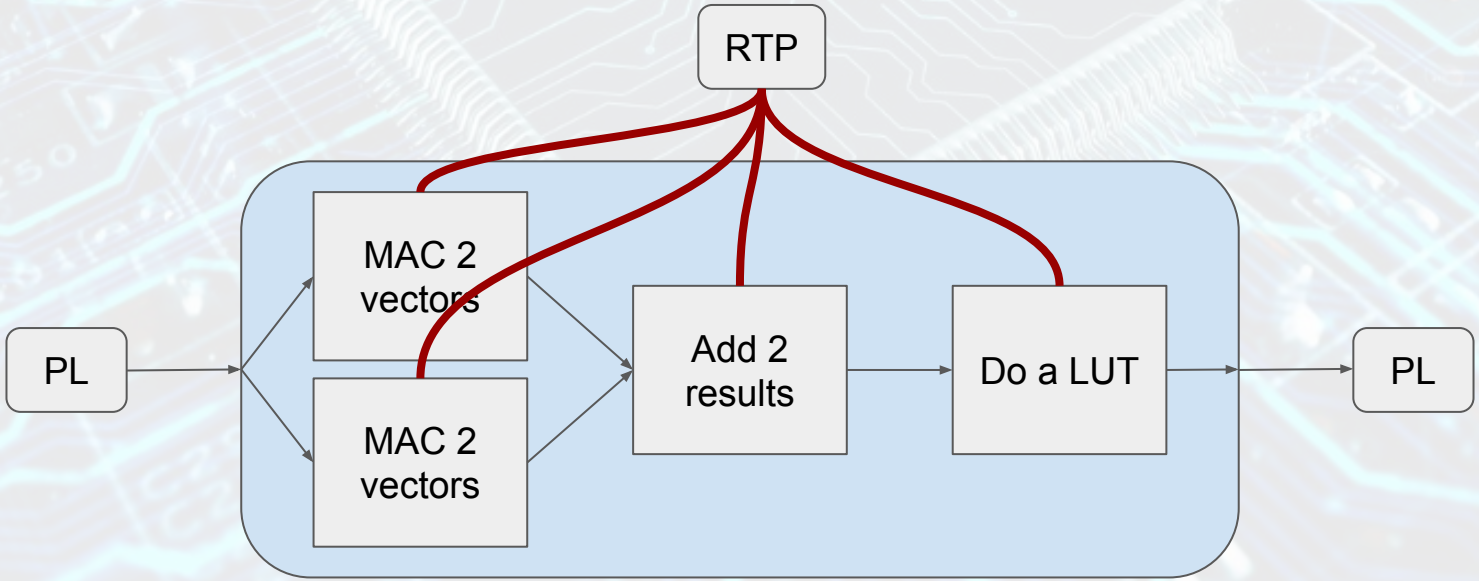


Organize into a top graph with multiple subgraphs



Run Time Parameters

- Passed to the memory of the AI Engine Tiles by the PS (txt file, shell, std::cin)
- Can be changed - read in run time



Gated Recurrent Unit

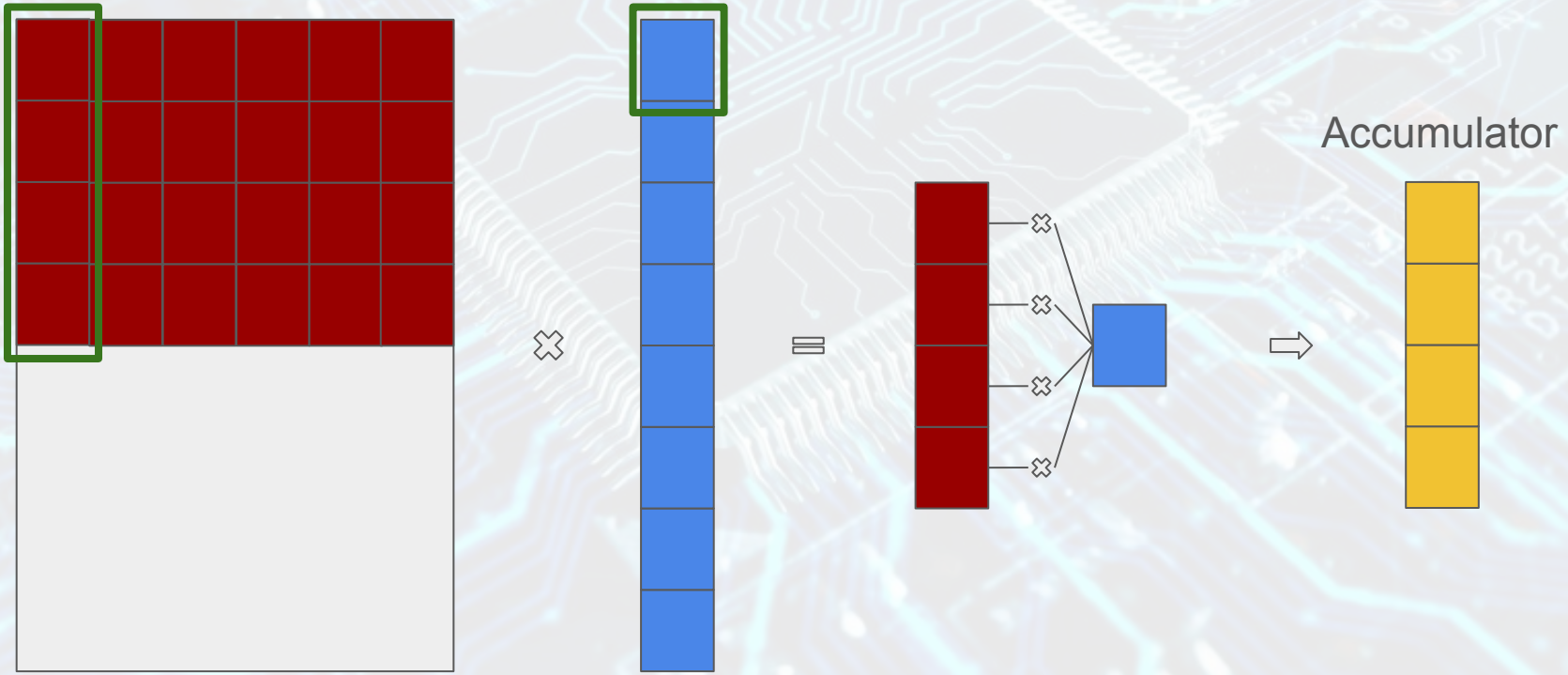
$$
\begin{aligned}
z_t &= \sigma(W_z x_t + U_z h_{t-1} + b_z) \\
r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r) \\
\hat{h}_t &= \phi(W_h x_t + U_h (r_t \odot h_{t-1}) + b_h) \\
h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \hat{h}_t
\end{aligned}
$$

if a Matrix wore pants

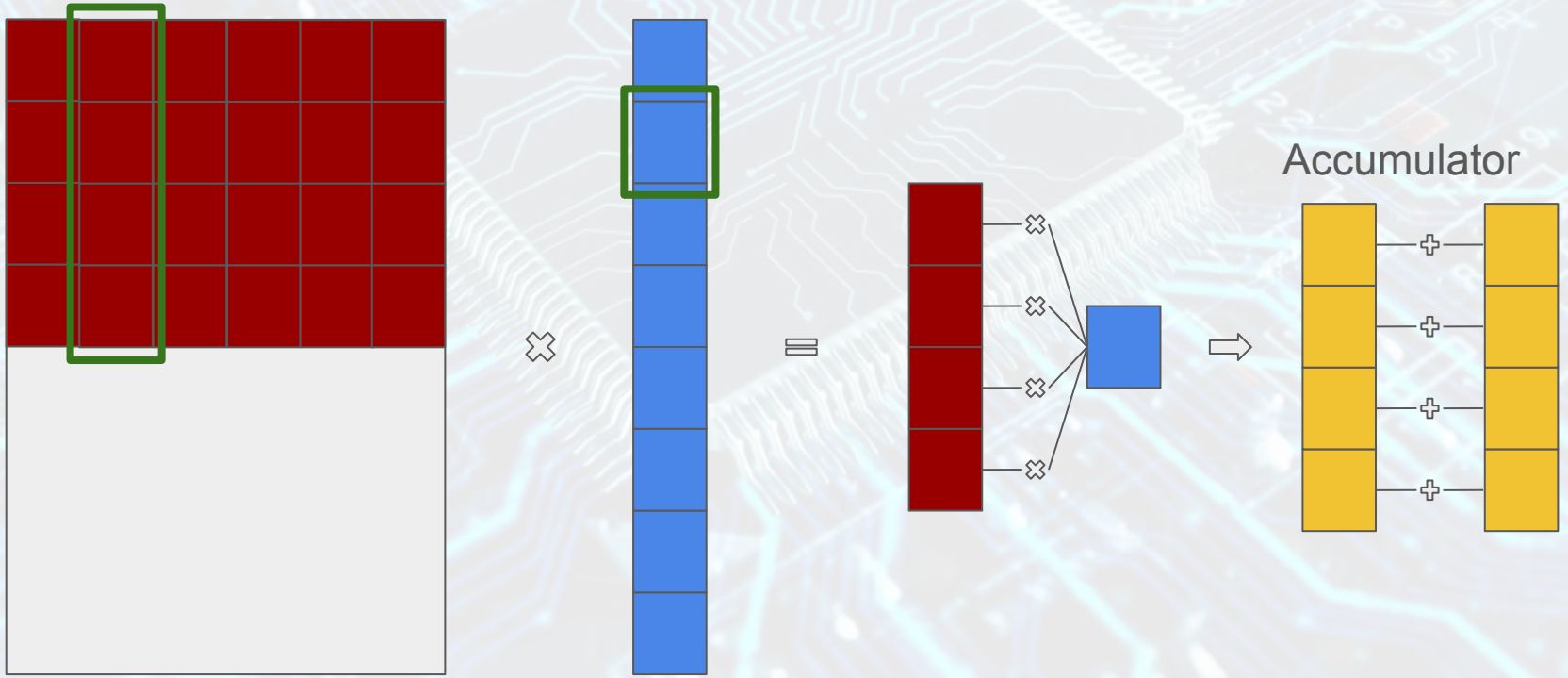
would it wear them like this

or like this

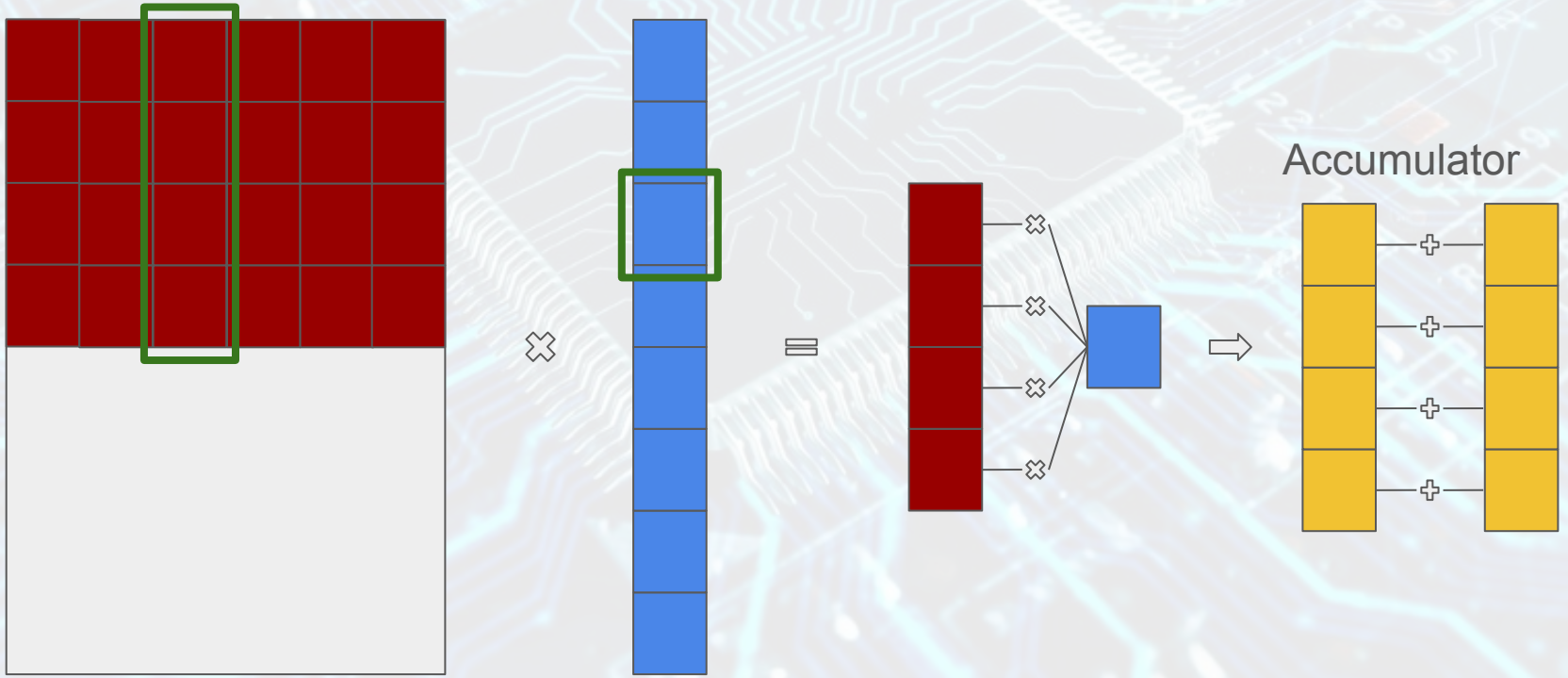
Multiply Accumulate: Columns



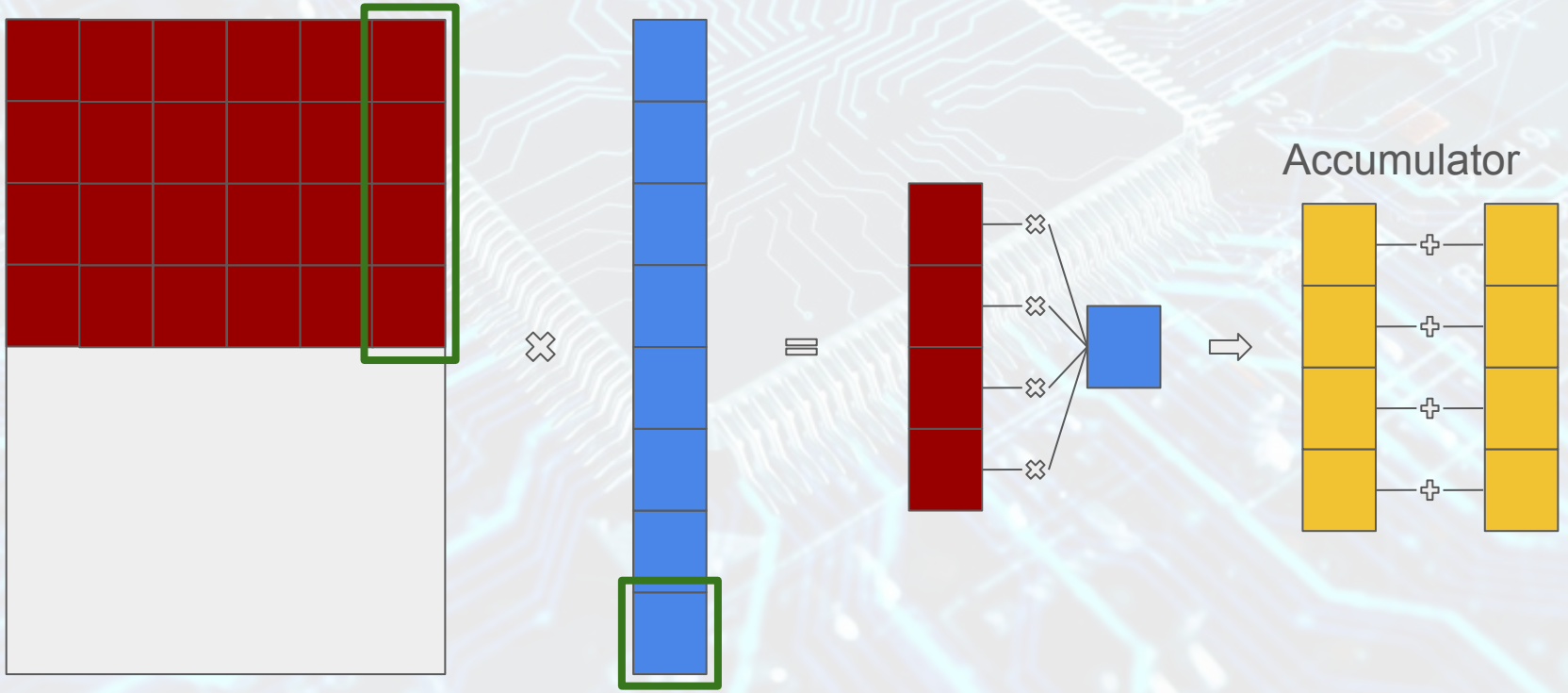
Multiply Accumulate: Columns



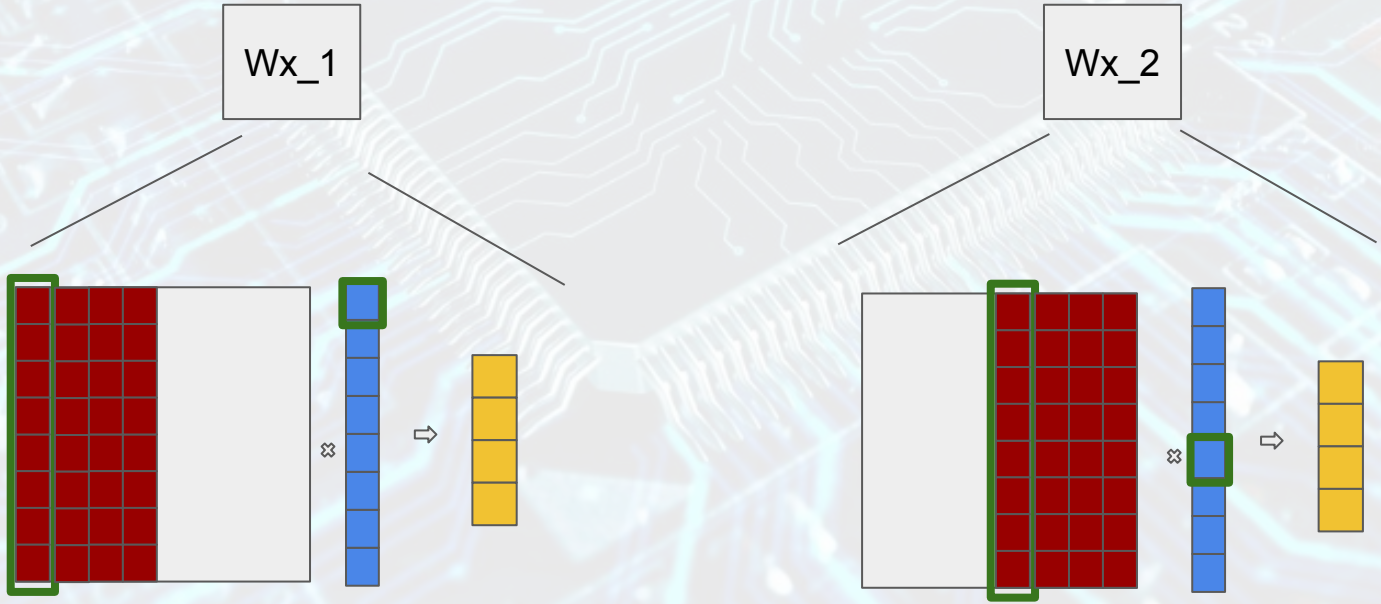
Multiply Accumulate: Columns



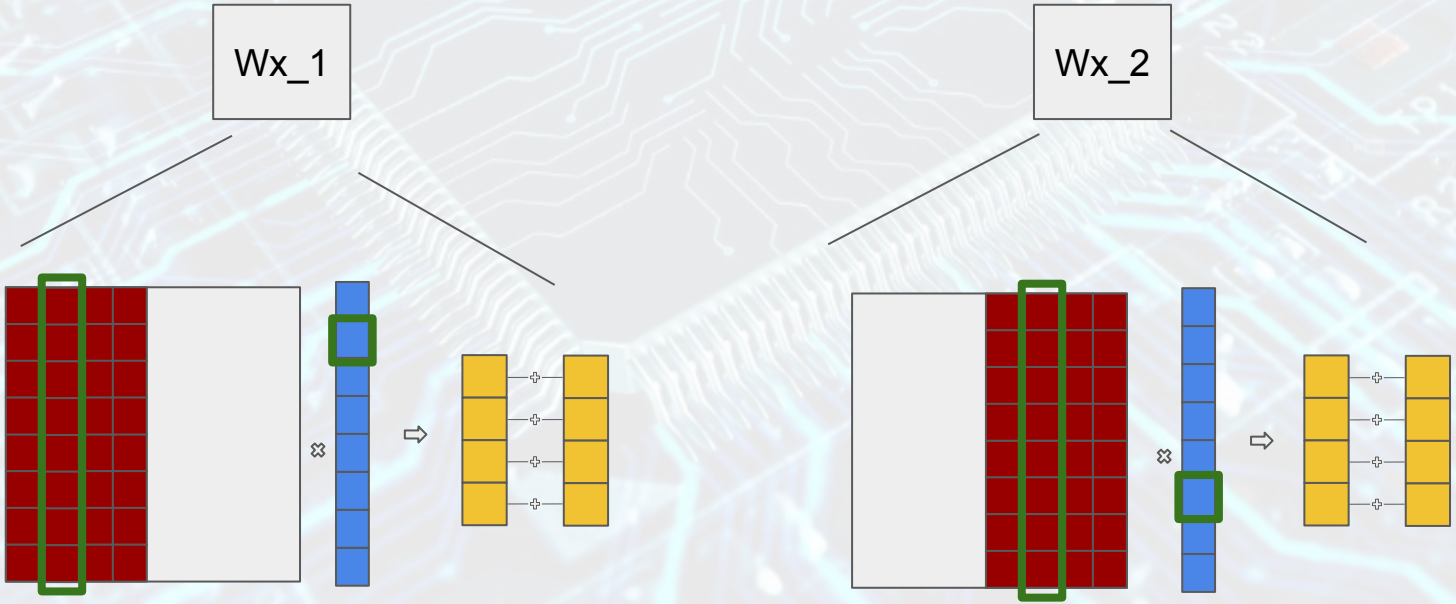
Multiply Accumulate: Columns



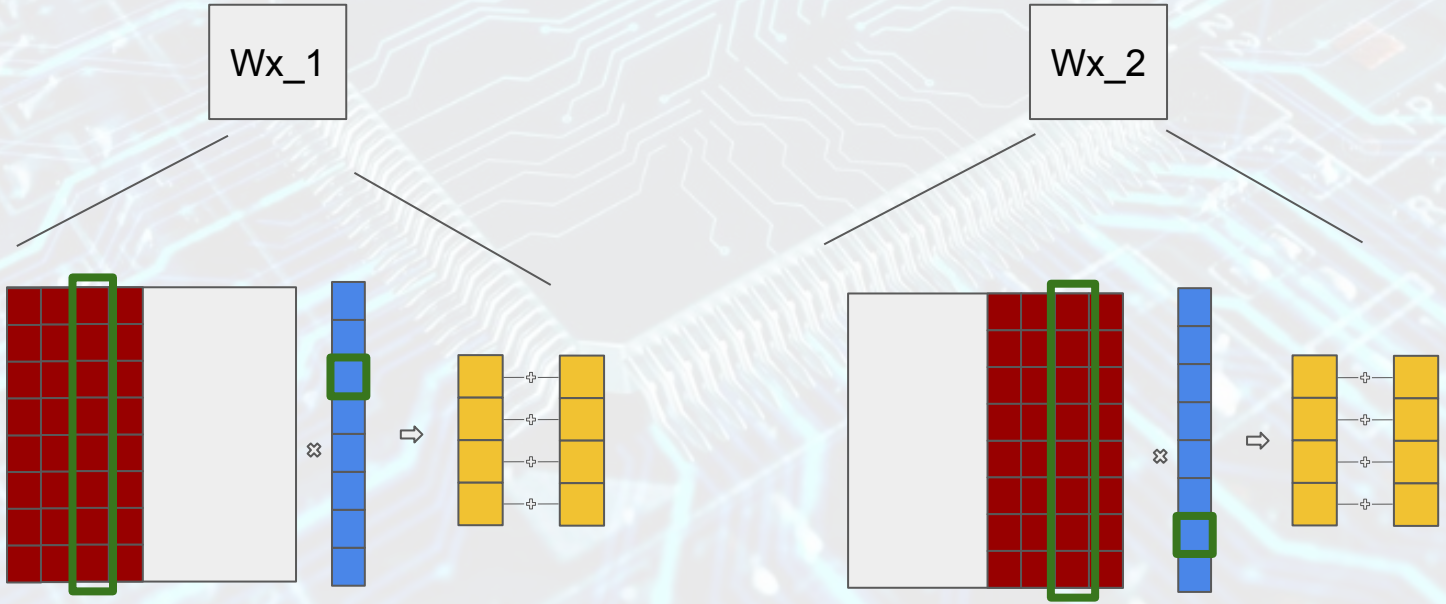
Split the columns into multiple tiles - Cascade the result



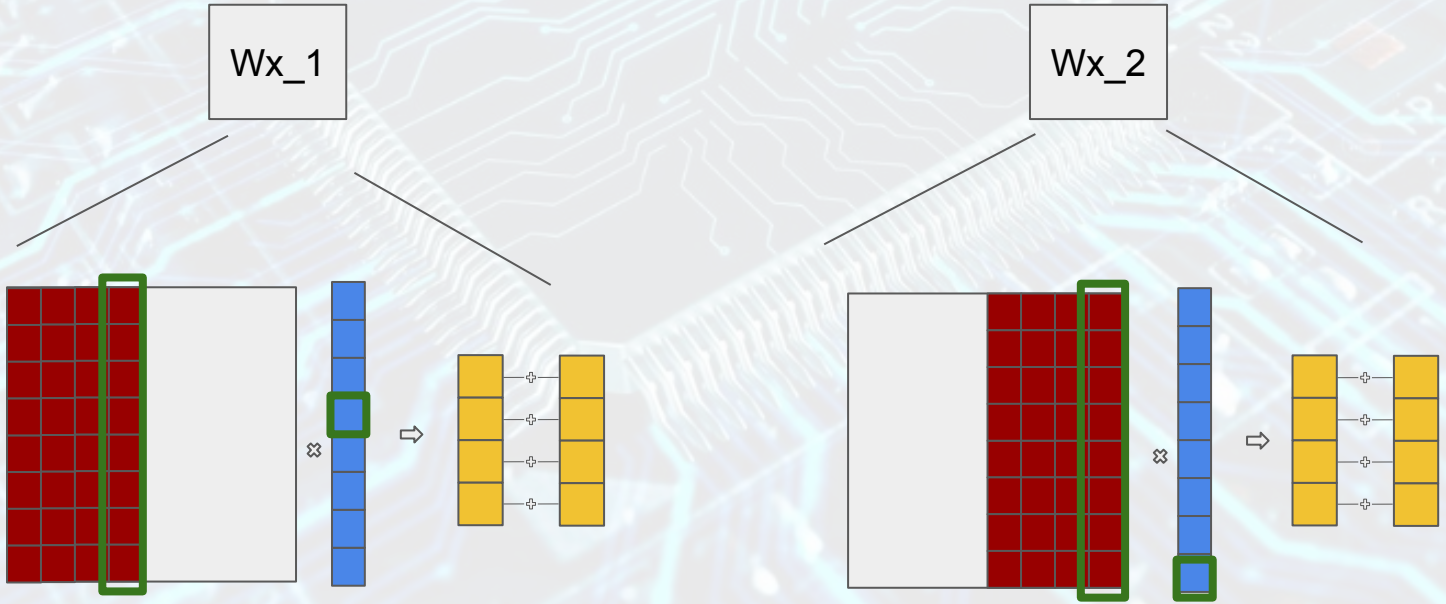
Split the columns into multiple tiles - cascade the result



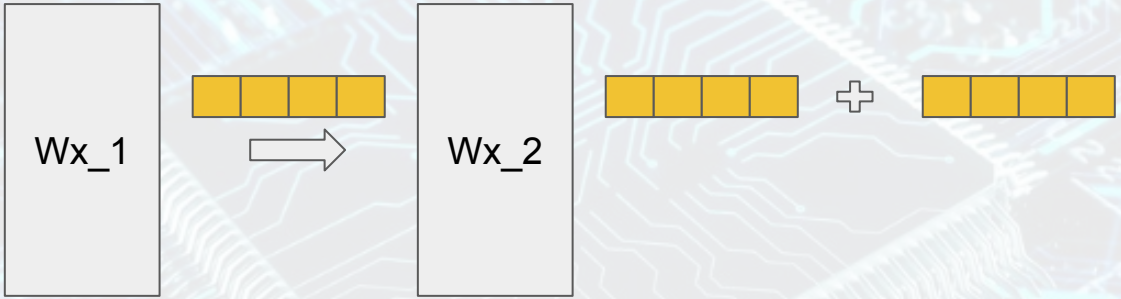
Split the columns into multiple tiles - Cascade the result

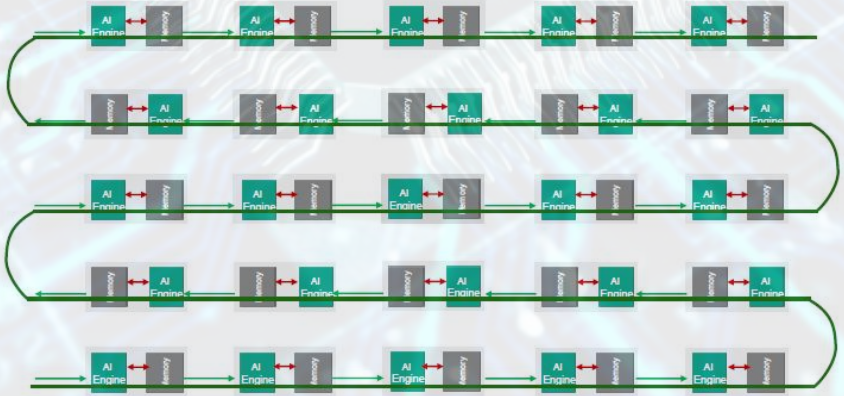


Split the columns into multiple tiles - Cascade the result



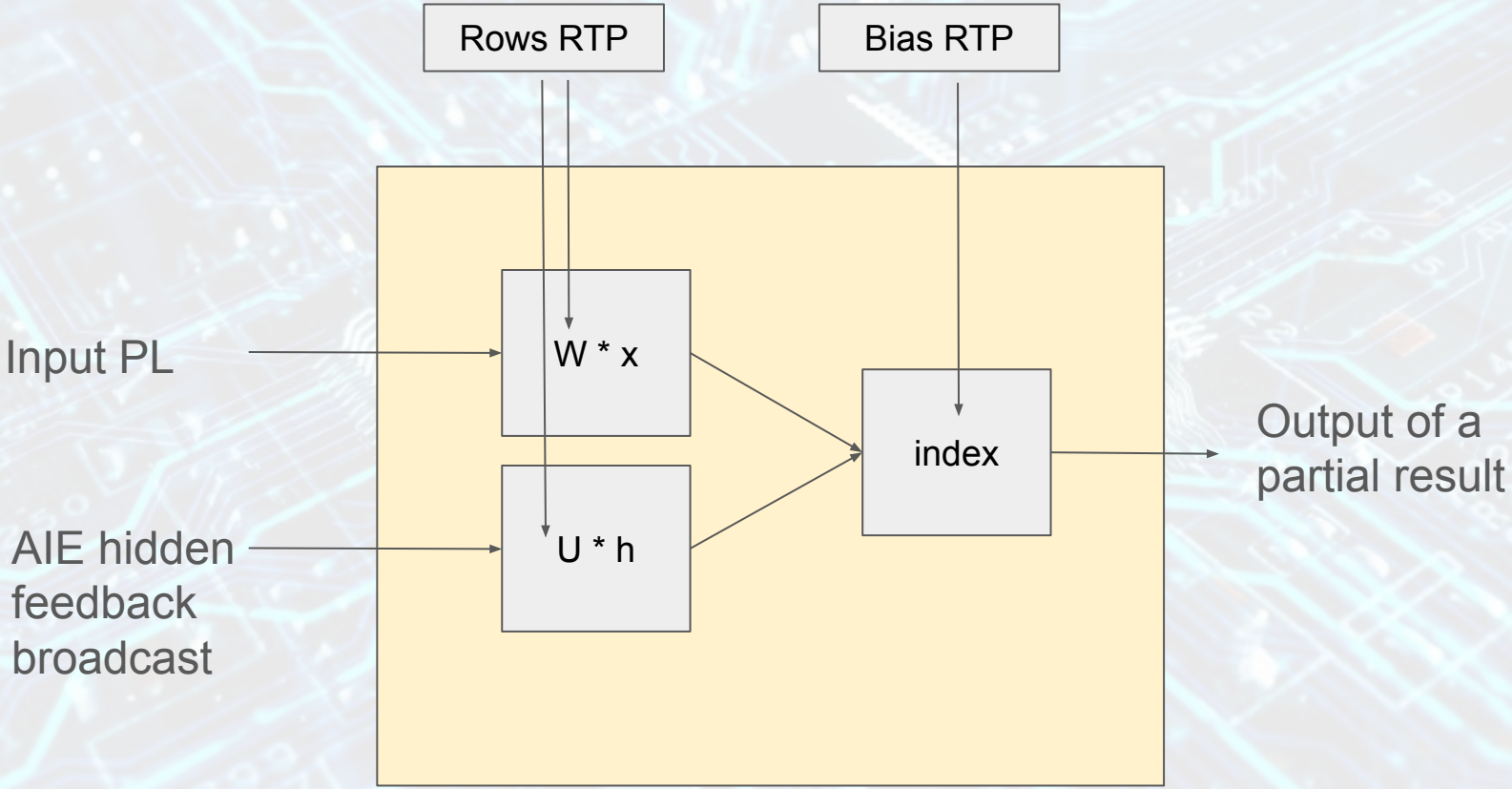
Cascade



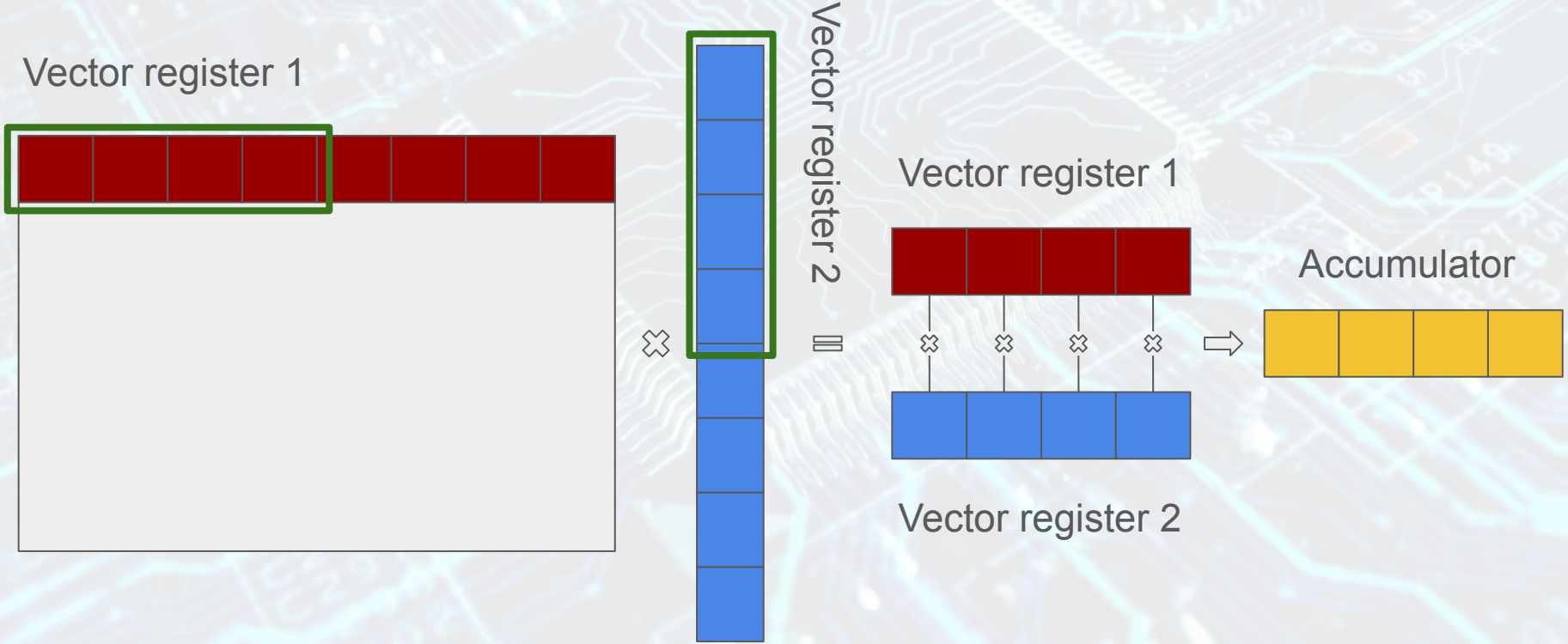


Problem

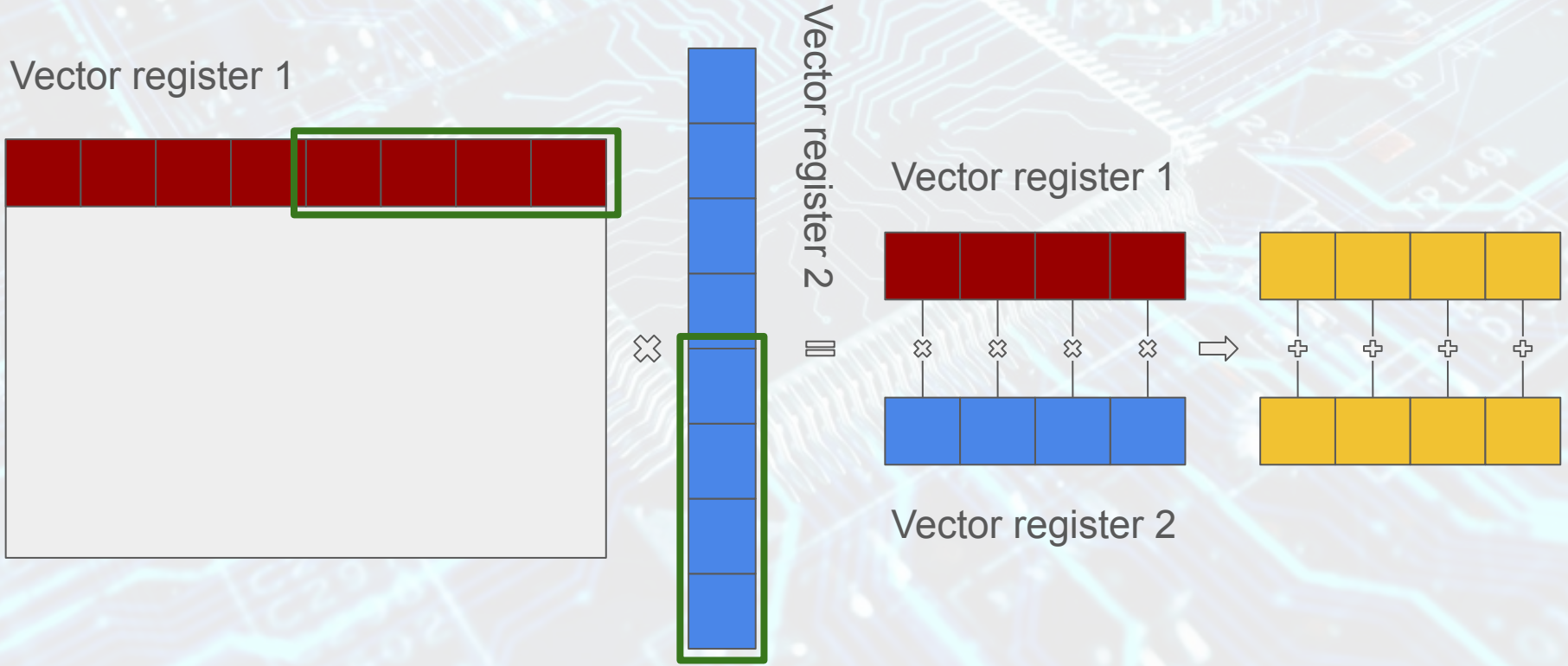
- The distribution happens on the “input size” of the model
- The bigger the chain of AI Tiles
- The more you need to wait for the partial results to be aggregated


Rows RTP
Bias RTP
Input PL
W * x
index
Output of a
partial result
AIE hidden
feedback
broadcast
U * h

Multiply Accumulate: Rows

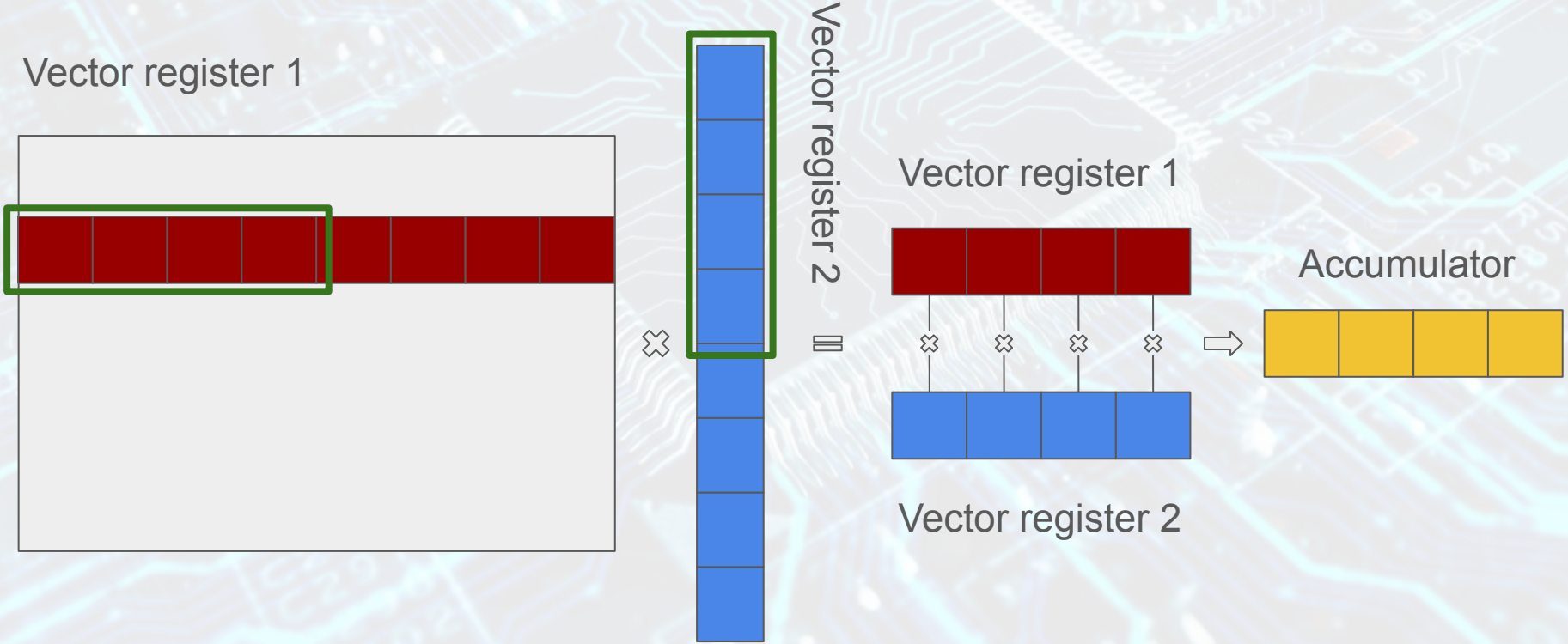


Multiply Accumulate: Rows

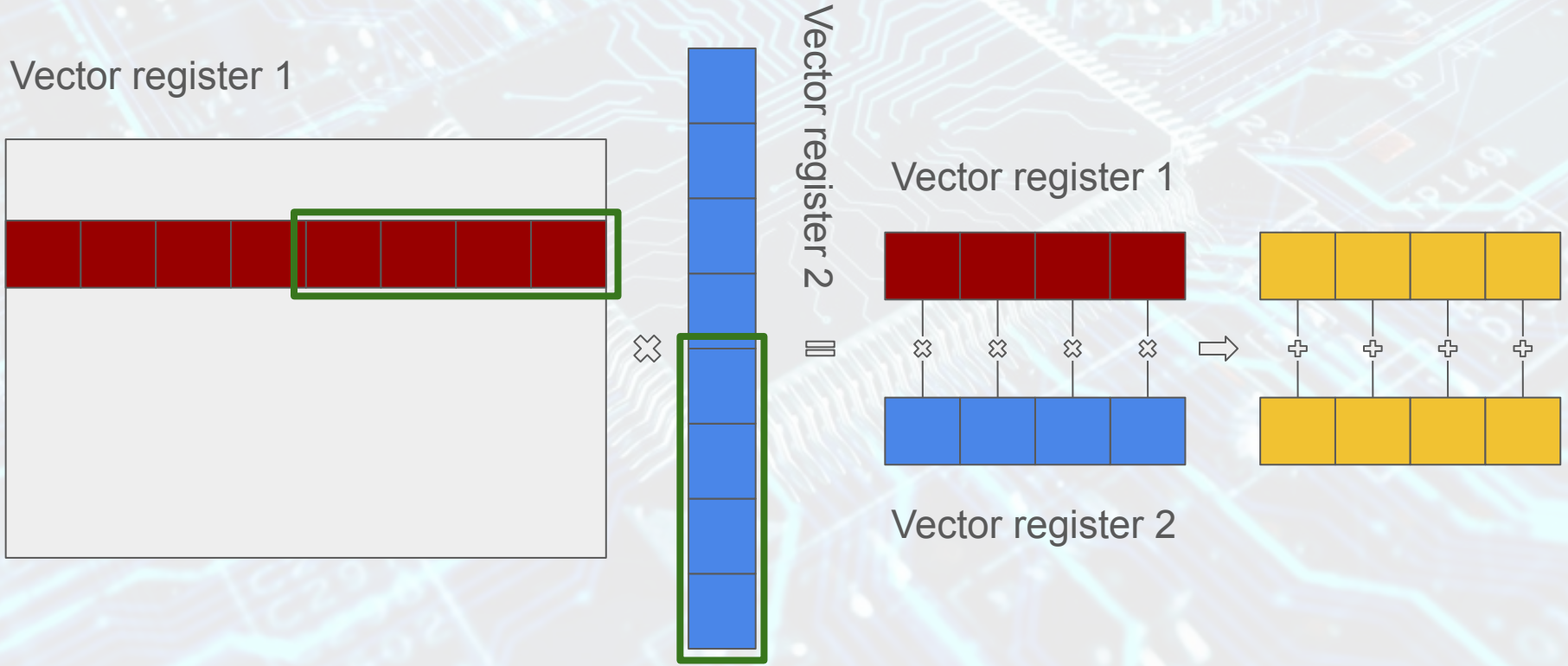


Multiply Accumulate: Rows

Multiply Accumulate: Rows

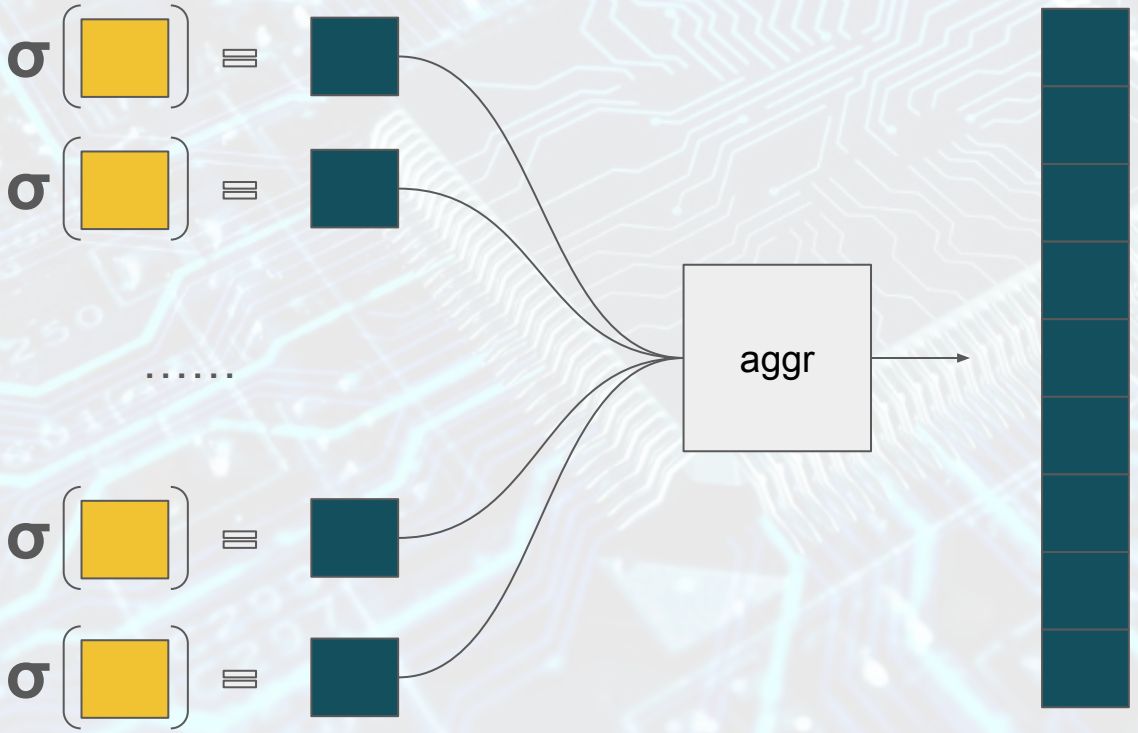


Multiply Accumulate: Rows



Multiply Accumulate: Rows

Aggregate partial results w Packet Stream: Merge



$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z)$$
$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$$
$$\hat{h}_t = \phi(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h)$$
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \hat{h}_t$$

Aggregation problem

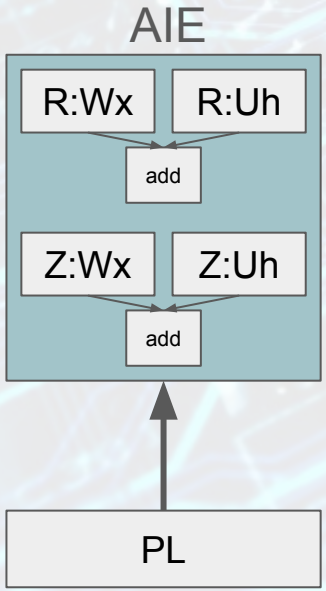
- Pkt Split headers take too long to decode ~ 9.4 ns with data read
- This scales bad

Can we do better? Yes, use PL

- Calculate rows, all at the same time
- Utilize interface tiles as aggregators
- You need to sort because Packets have non-deterministic behavior
- Sort and apply LUT in the FPGA
- Broadcast the result into the AIE kernels of the next gate

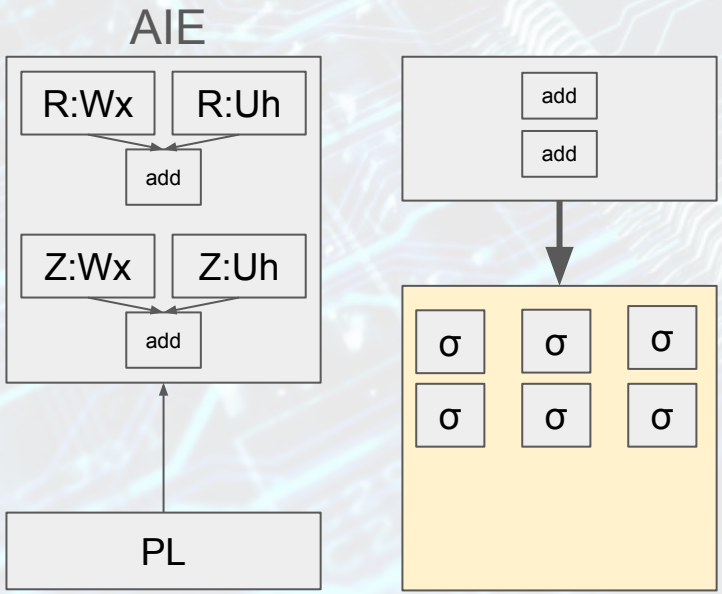
Going forward: Hybrid Solution

- Use the AI Engine only to distribute Matrix - Vector calculations



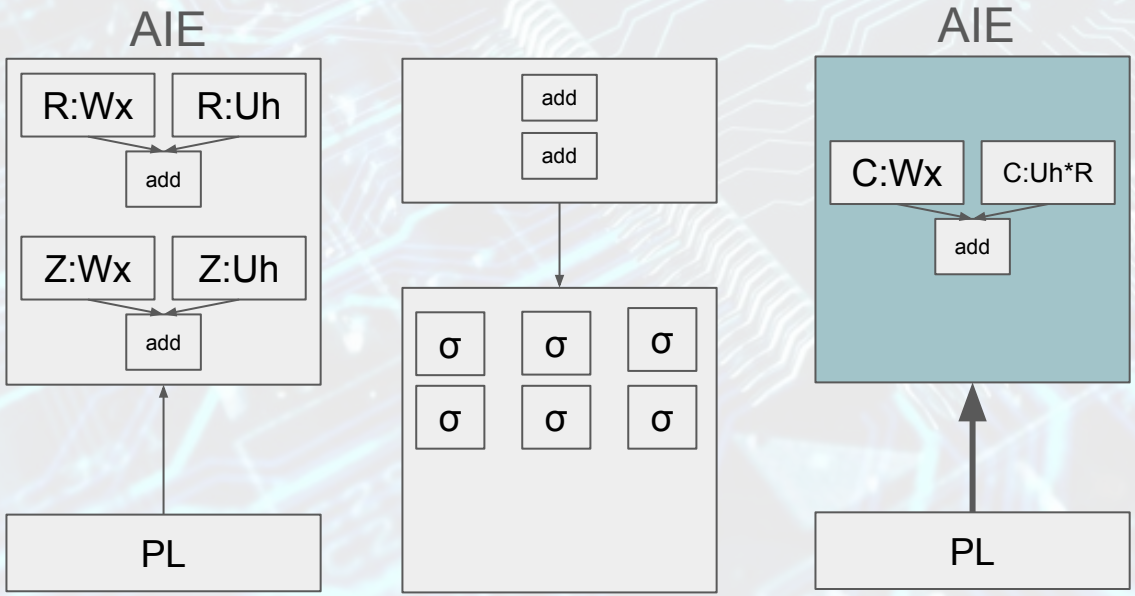
Going forward: Hybrid Solution

- Use the AI Engine only to distribute Matrix - Vector calculations



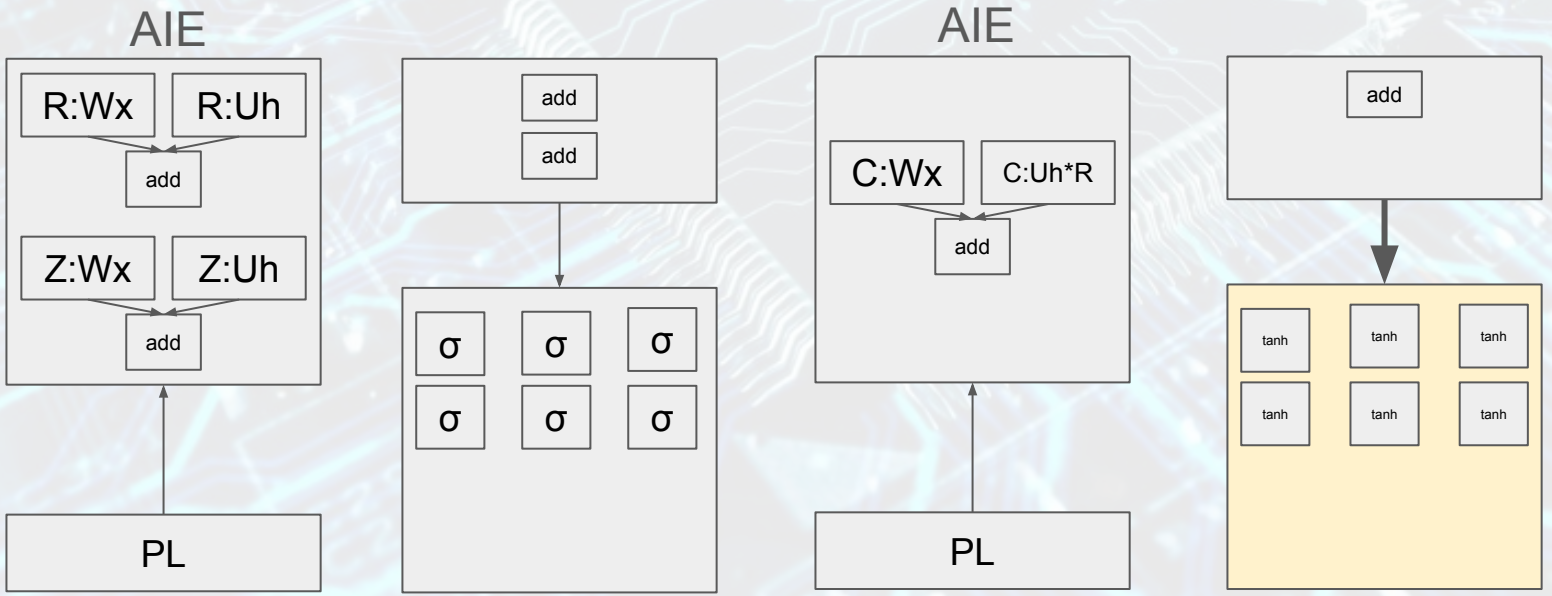
Going forward: Hybrid Solution

- Use the AI Engine only to distribute Matrix - Vector calculations



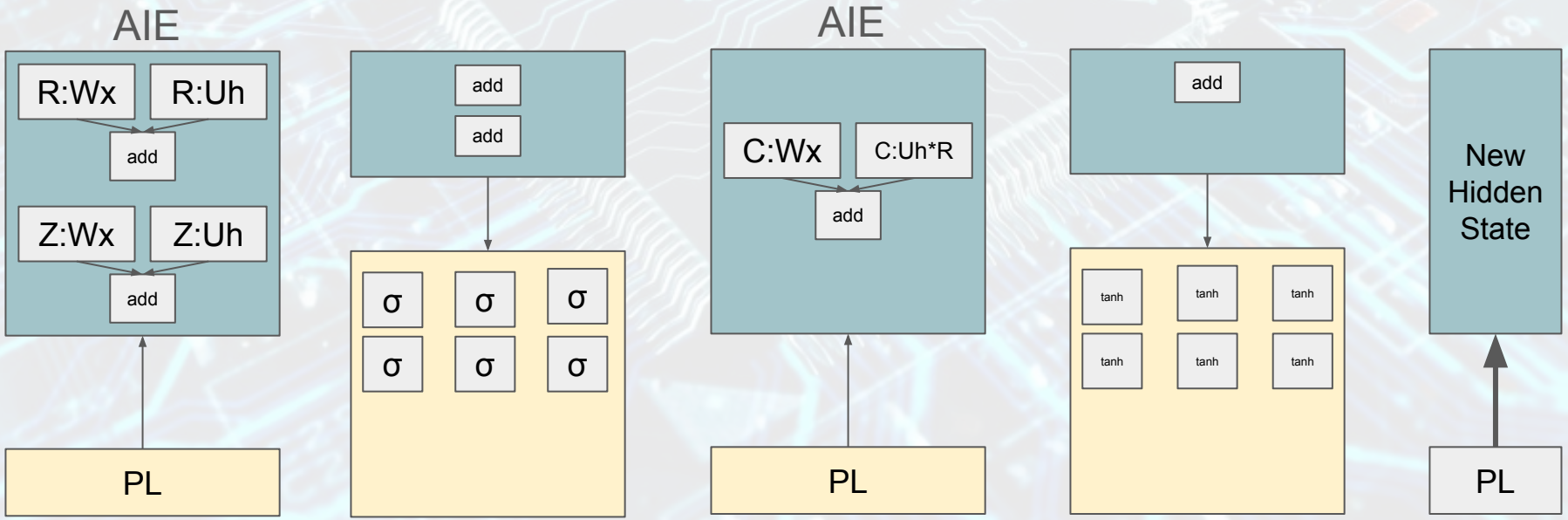
Going forward: Hybrid Solution

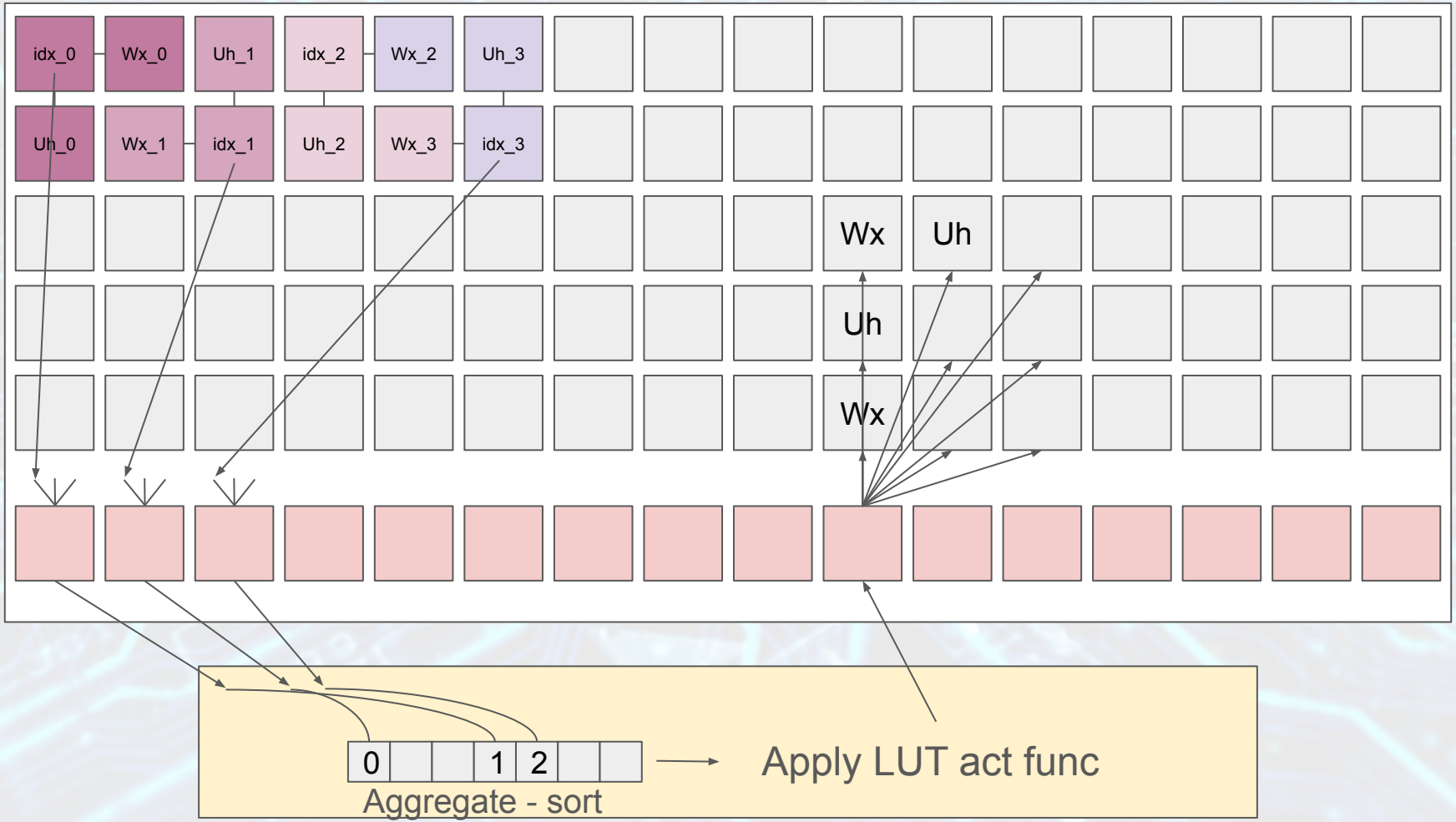
- Use the AI Engine only to distribute Matrix - Vector calculations



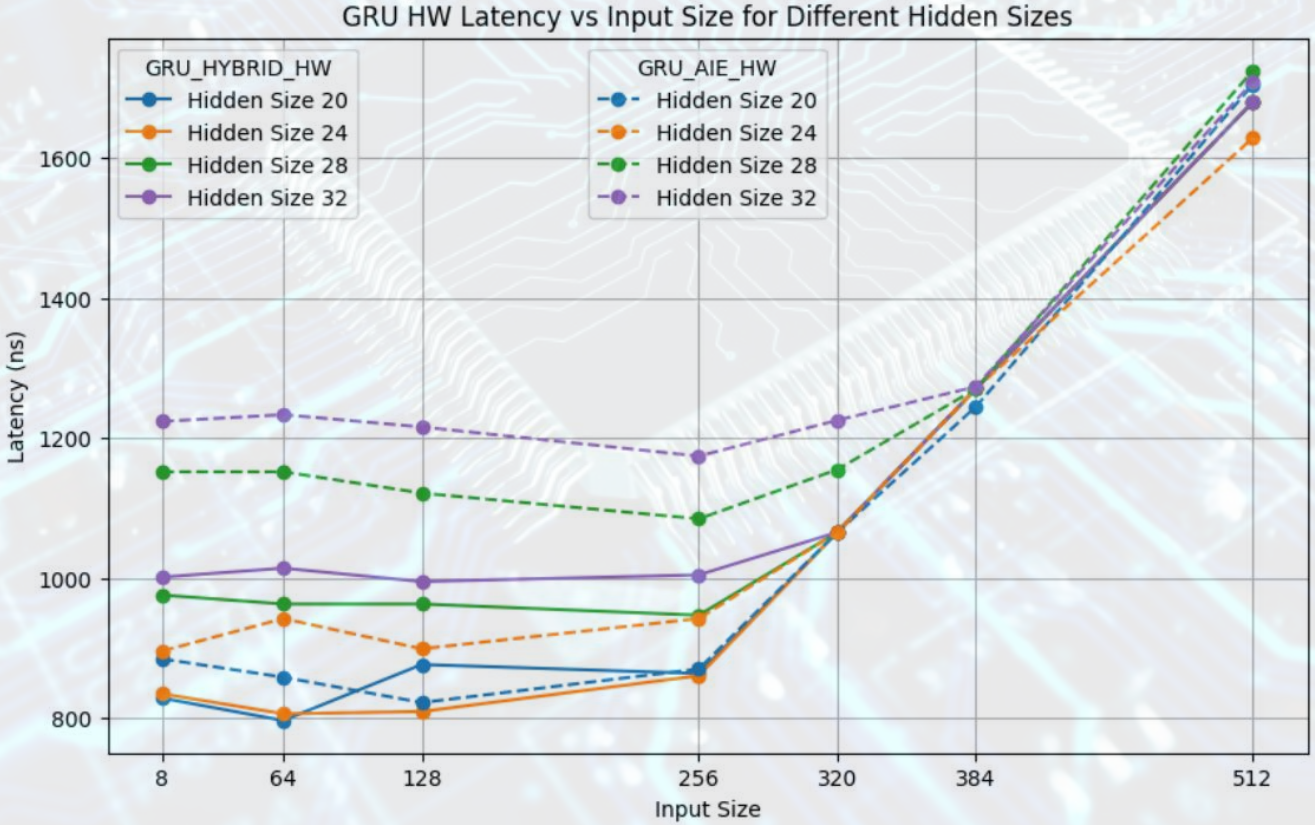
Going forward: Hybrid Solution

- Use the AI Engine only to distribute Matrix - Vector calculations




idx_0
Wx_0
Uh_1
idx_2
Wx_2
Uh_3
Uh_0
Wx_1
idx_1
Uh_2
Wx_3
idx_3
Wx
Uh
Uh
Wx
0
1
2
Aggregate - sort
Apply LUT act func

*** All numerical computations are done in FP32s ***



Conclusion

- Unquantized FP32 model inference
- We can run “big” RNN models in microseconds
- That would never fit into an FPGA

- Difficult to program
- Insanely difficult to debug, especially for complex designs
- If you have the efficiency bug you’re cooked

- You have to write PL (FPGA) code and Host (CPU) code to interact with

aiecompiler

```
void mat_hidden_vec_mul(input_stream<float> * __restrict in,
                        output_stream<float> * __restrict out,
                        const float (&weights)[HW_HIDDEN_SIZE*DIST_COEFF],
                        const float (&h_init)[HW_HIDDEN_SIZE]

){  constexpr int H_ITER = HW_HIDDEN_SIZE / VECTOR_LANES; //how many times to loop
    alignas(32) aie::accum<accfloat, VECTOR_LANES> acc;
    alignas(32) aie::vector<float, VECTOR_LANES> hidden[H_ITER];
    alignas(32) aie::vector<float, VECTOR_LANES> * v_weights = (aie::vector<float, VECTOR_LANES>*) &weights;
    alignas(32) aie::vector<float, VECTOR_LANES> * v_hidden_init  = (aie::vector<float, VECTOR_LANES>*) &h_init;
    // bool tlast = false;

    for (int i = 0; i < H_ITER; i++) chess_loop_count(H_ITER)
        {
        hidden[i].insert(0,v_hidden_init[i]);
    }

    // while (!tlast){
    for(int i = 0; i< INF_REC_N; i++ ) chess_prepare_for_pipelining {
        for (int i = 0; i < DIST_COEFF; i++) chess_loop_count(DIST_COEFF)
            {
            acc = aie::zeros<accfloat, VECTOR_LANES>();
            for (int j = 0; j < H_ITER ; j++) chess_loop_count(H_ITER) chess_prepare_for_pipelining
                {
                acc = aie::mac( acc,
                                hidden[j],
                                v_weights[i*(H_ITER) + j]
                                );
            }
            writeincr(out, aie::reduce_add(acc.to_vector<float>(0)));
        }

        for (int i = 0; i < H_ITER; i++) chess_loop_count(H_ITER) chess_prepare_for_pipelining
            {
            hidden[i].insert(0,readincr_v<4>(in));
            if (VECTOR_LANES == 8) { hidden[i].insert(1, readincr_v<4>(in)); }
        }
    }
}
```

The End!