

hls4ml Tutorial

Part I. Introduction

Introduction

- In this session you will get hands on experience with the **hls4ml** package
- Translate pre-trained models into FPGA code
- Explore the different handles provided by the tool to optimize the inference
 - Latency, throughput, resource usage
- Run inference on an FPGA with AWS
- Make our inference more computationally efficient with pruning
- But first...

What are FPGAs?

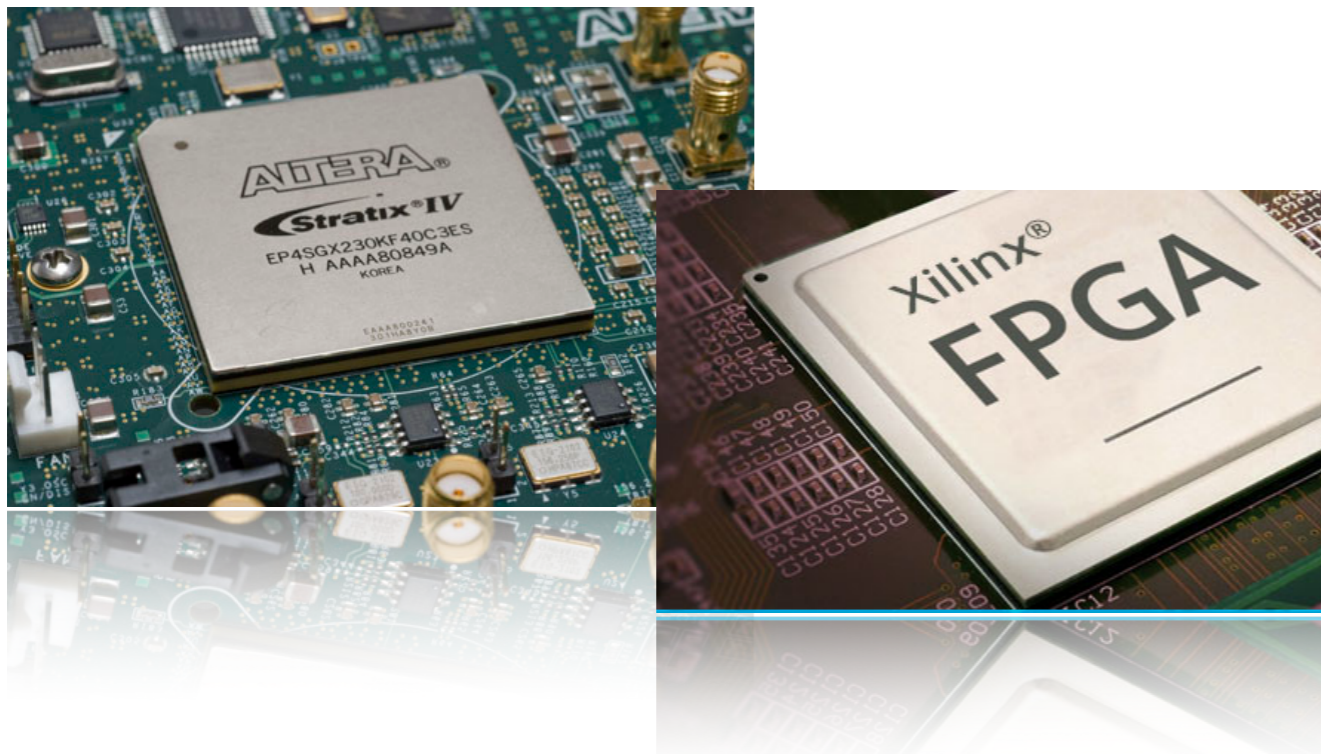
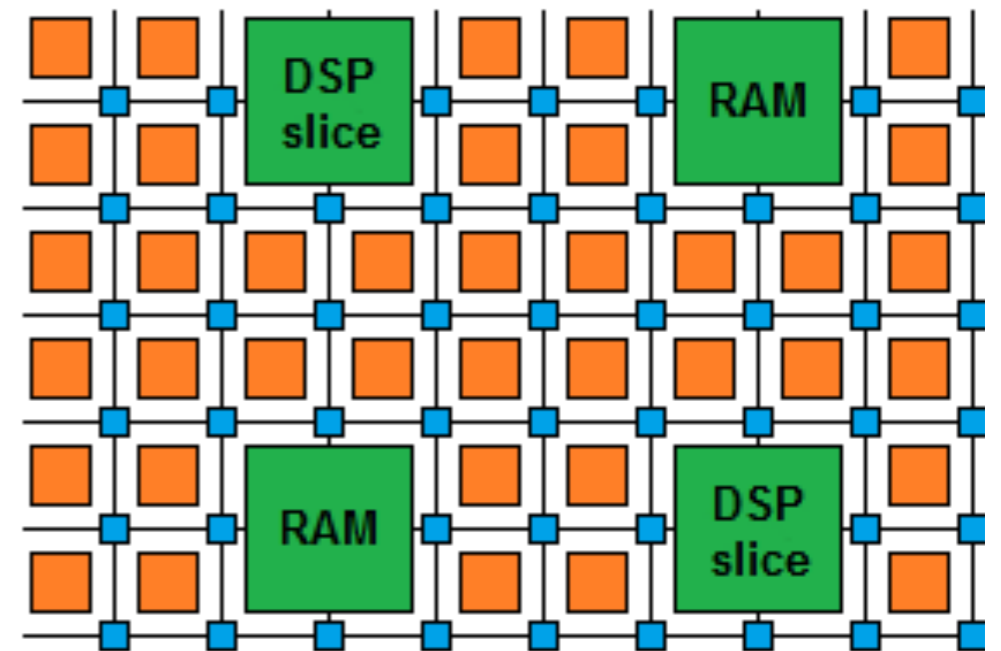
Field Programmable Gate Arrays are reprogrammable integrated circuits

Contain many different building blocks ('resources') which are connected together as you desire

Originally popular for prototyping ASICs, but now also for high performance computing

'Computing in space as well as time'

FPGA diagram



What are FPGAs?

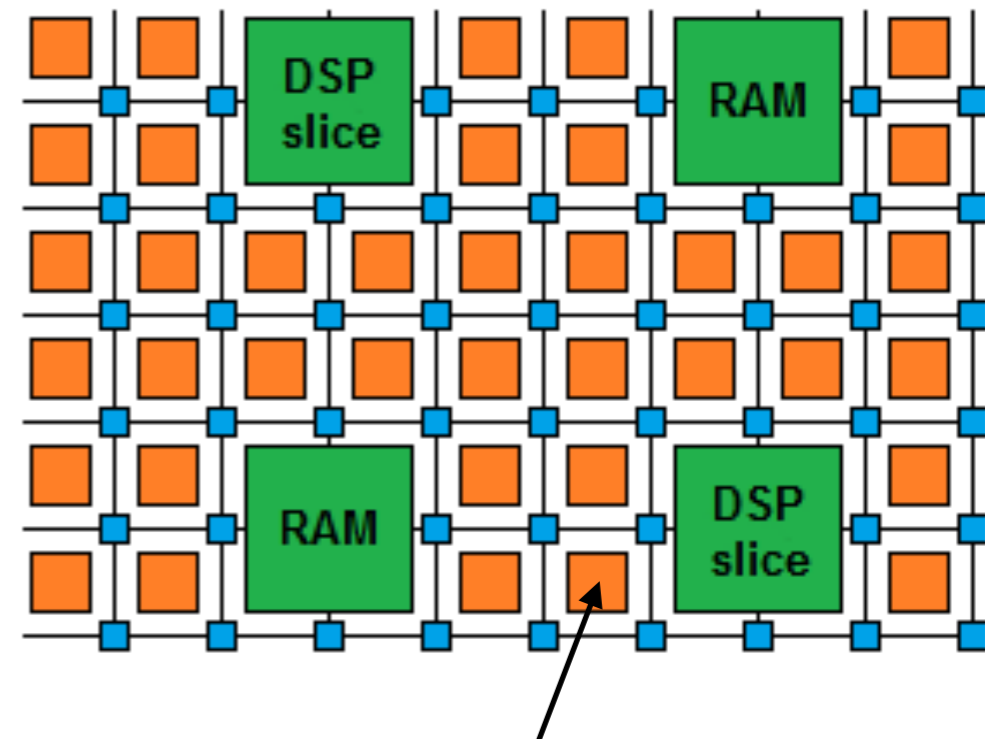
Field Programmable Gate Arrays are reprogrammable integrated circuits

Logic cells / Look Up Tables perform arbitrary functions on small bitwidth inputs (2-6)

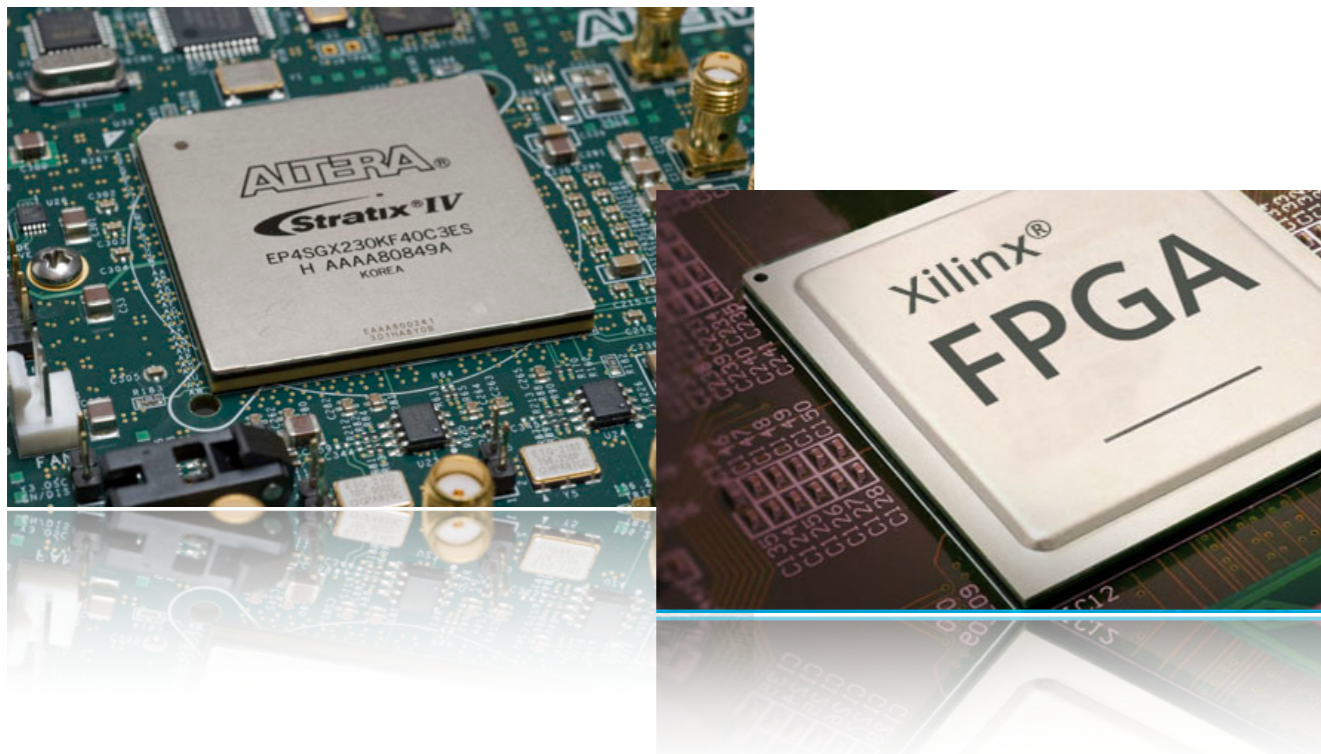
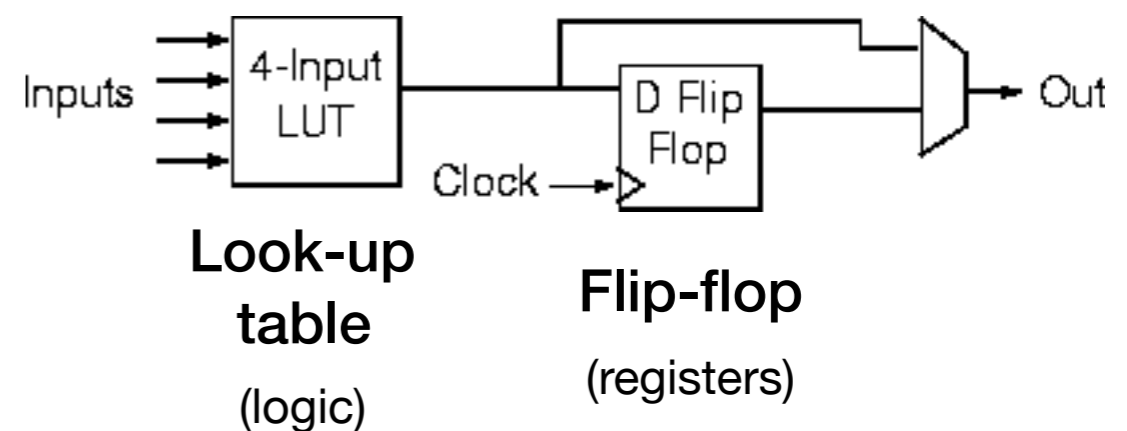
These can be used for boolean operations, arithmetic, memory

Flip-Flops register data in time with the clock pulse

FPGA diagram



Logic cell



What are FPGAs?

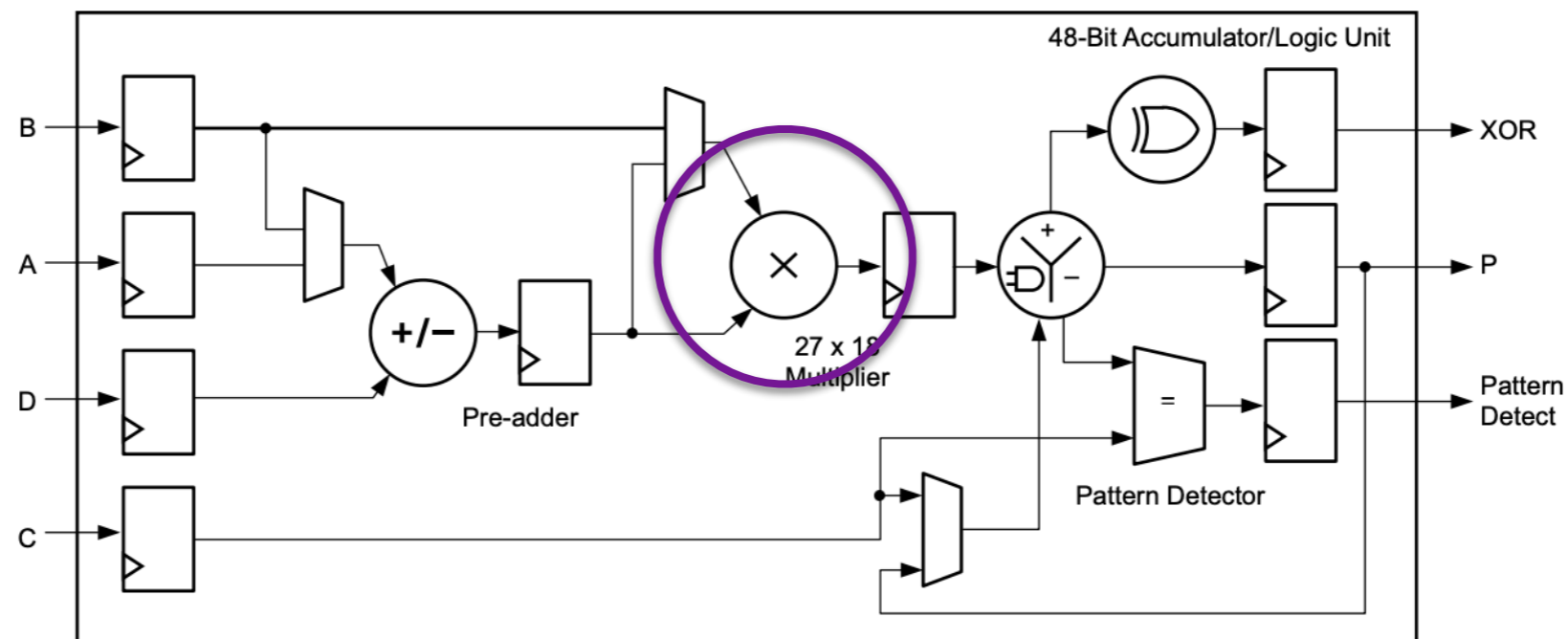
Field Programmable Gate Arrays are reprogrammable integrated circuits

DSPs are specialized units for multiplication and arithmetic

Faster and more efficient than using LUTs for these types of operations

And for Neural Nets, DSPs are often the most precious

DSP diagram



What are FPGAs?

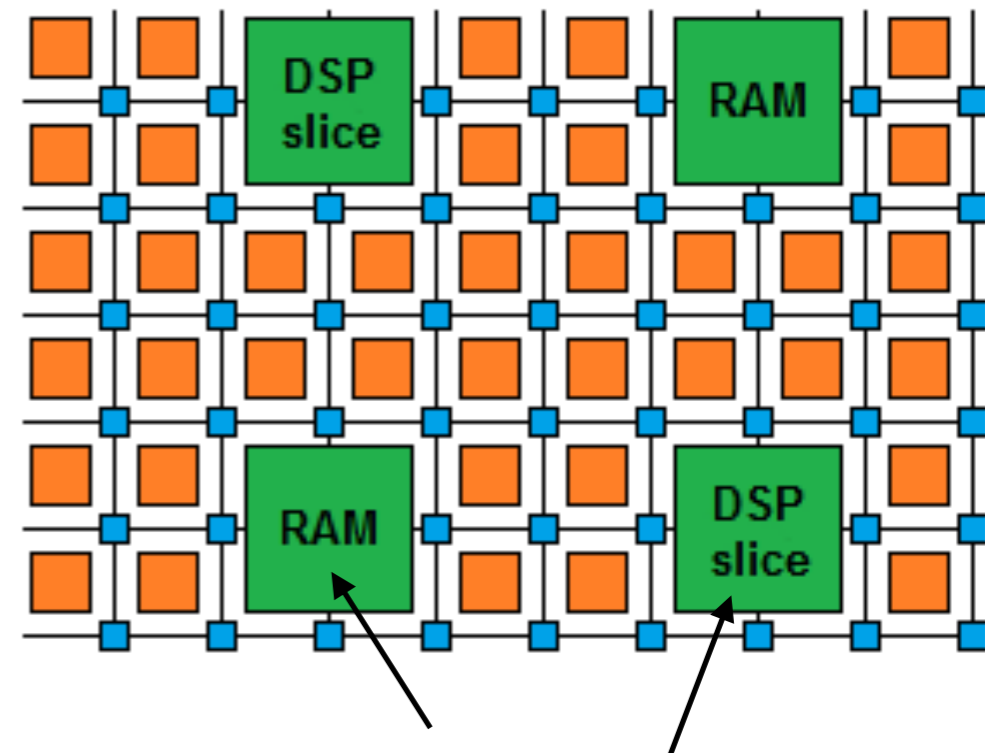
Field Programmable Gate Arrays are reprogrammable integrated circuits

BRAMs are small, fast memories - RAMs, ROMs, FIFOs (18Kb each in Xilinx)

Again, memories using BRAMs are more efficient than using LUTs

A big FPGA has nearly 100Mb of BRAM, chained together as needed

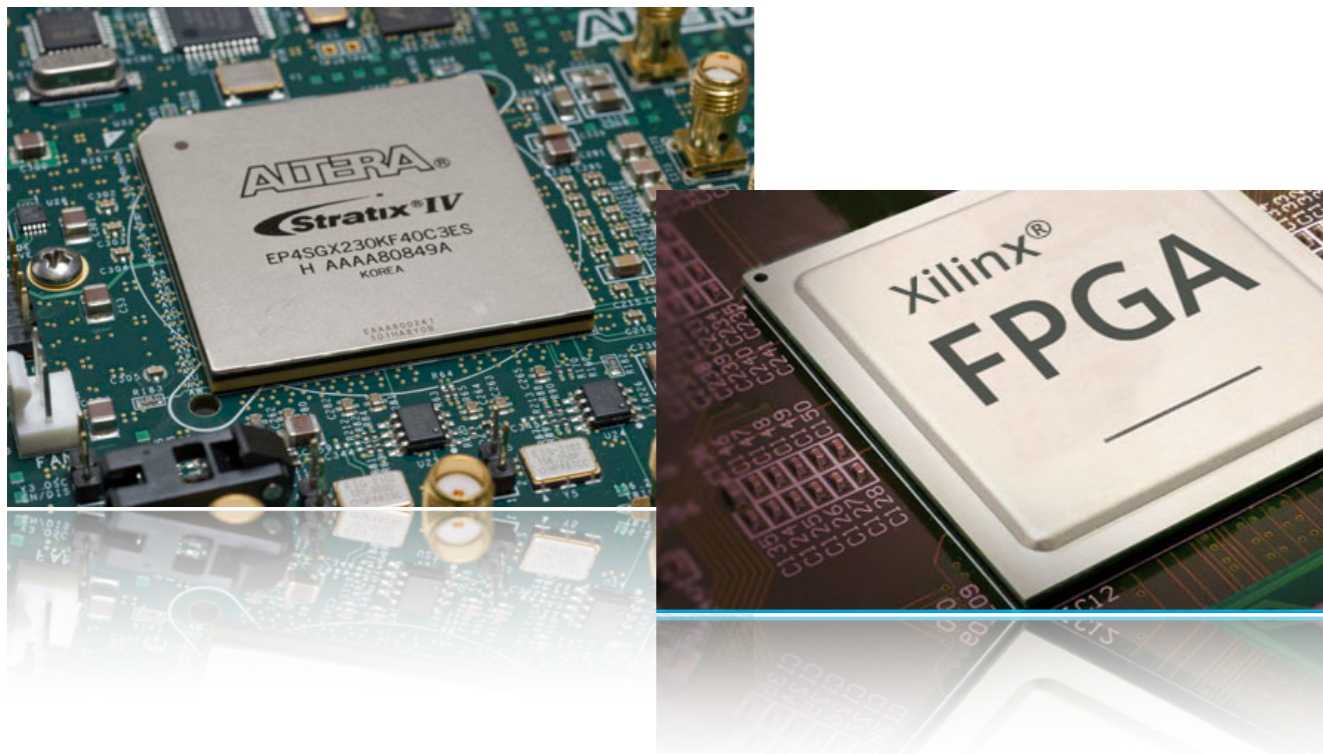
FPGA diagram



Also contain embedded components:

Digital Signal Processors (DSPs):
logic units used for multiplications

Random-access memories (RAMs):
embedded memory elements



What are FPGAs?

In addition, there are specialised blocks for I/O, making FPGAs popular in embedded systems and HEP triggers

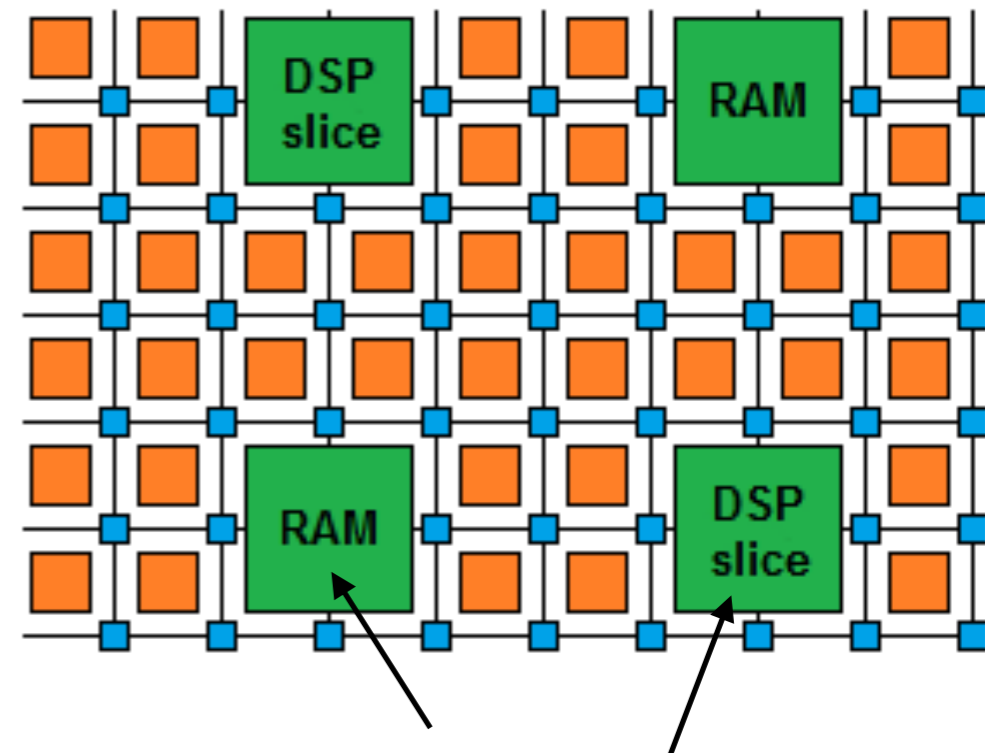
High speed transceivers with Tb/s total bandwidth

PCIe, (Multi) Gigabit Ethernet, Infiniband

AND: Support **highly parallel** algorithm implementations

Low power per Op (relative to CPU/GPU)

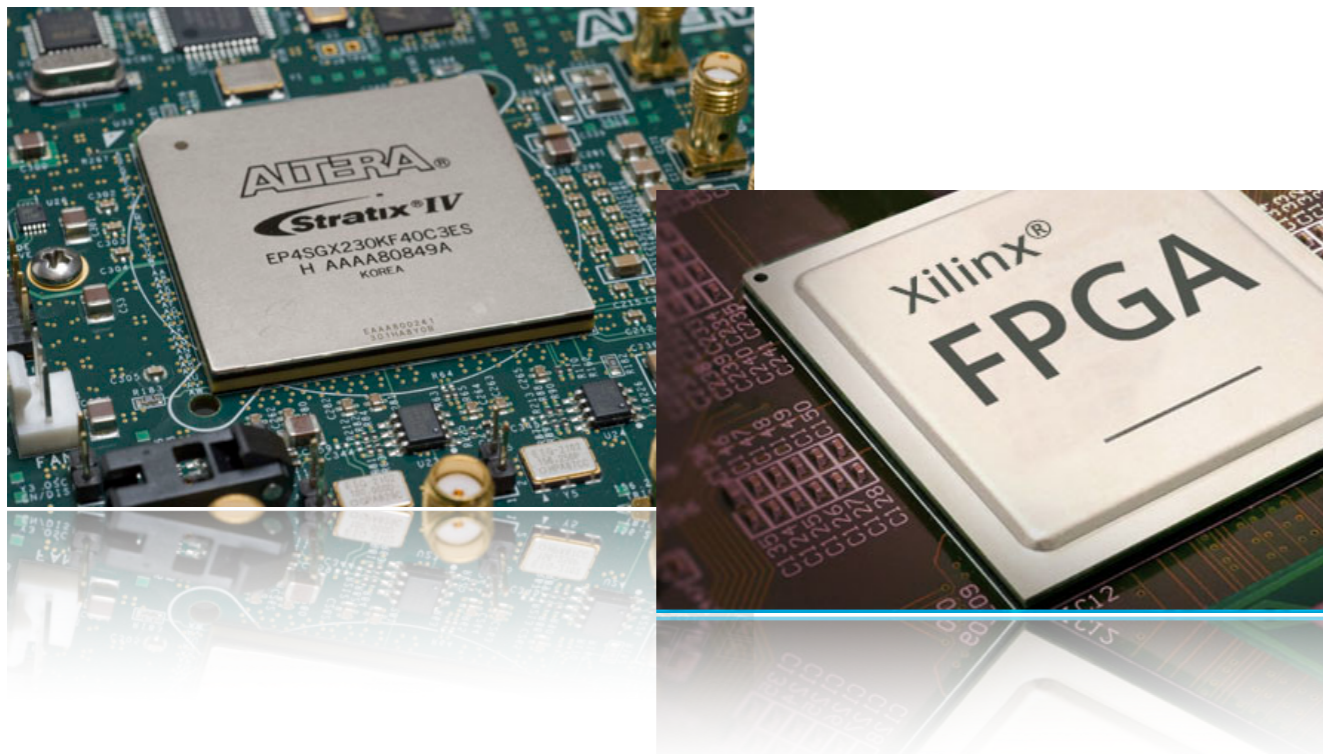
FPGA diagram



Digital Signal Processors (DSPs):
logic units used for multiplications

Random-access memories (RAMs):
embedded memory elements

Flip-flops (FF) and look up tables (LUTs) for additions



Why are FPGAs *Fast*?

- Fine-grained / resource parallelism
 - Use the many resources to work on different parts of the problem simultaneously
 - Allows us to achieve **low latency**
- Most problems have at least some sequential aspect, limiting how low latency we can go
 - But we can still take advantage of it with...
- Pipeline parallelism
 - Use the register pipeline to work on different data simultaneously
 - Allows us to achieve **high throughput**



Like a production line for data...

How are FPGAs programmed?

Hardware Description Languages

HDLs are programming languages which describe electronic circuits

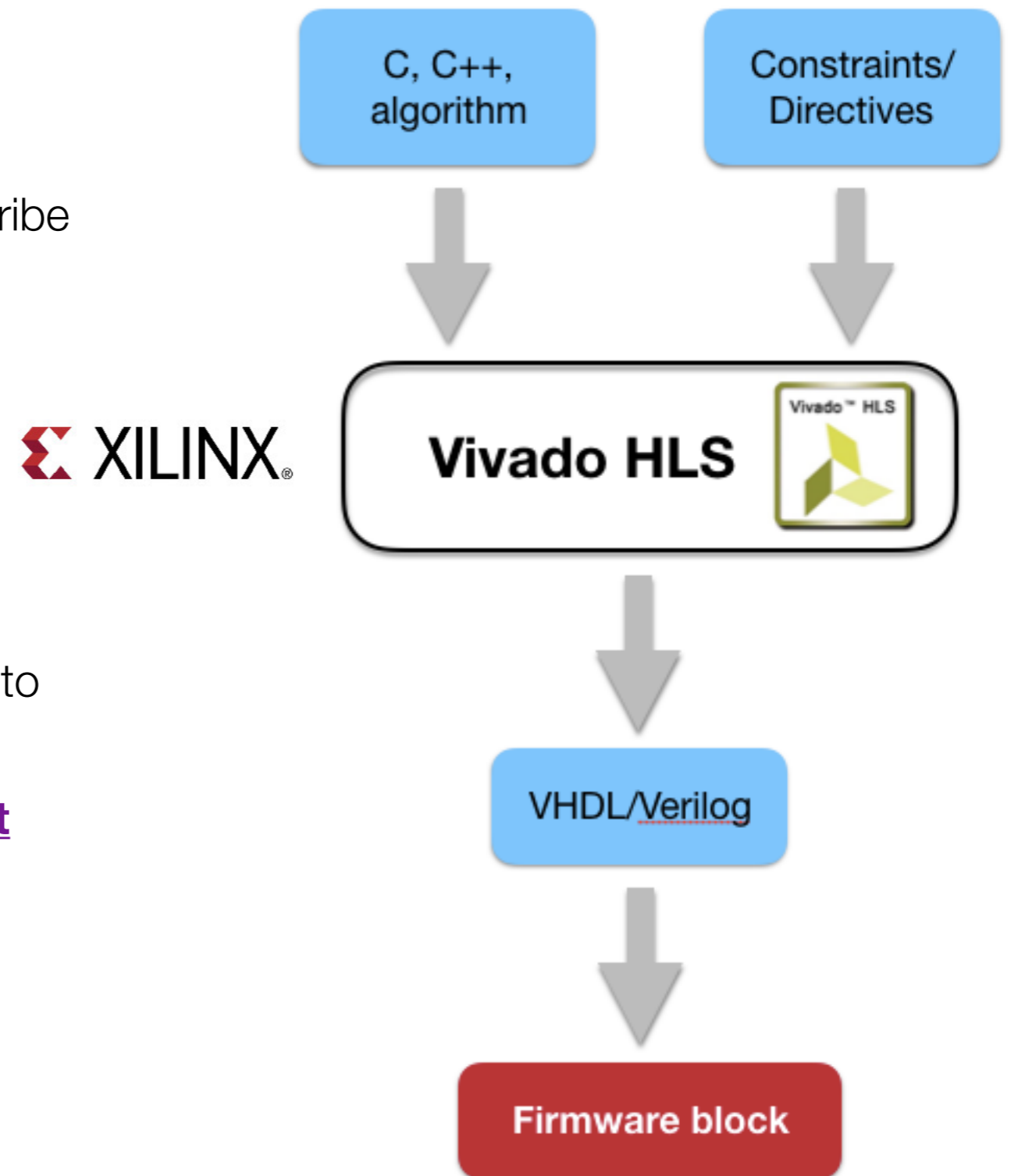
High Level Synthesis

Compile from C/C++ to VHDL

Pre-processor directives and constraints used to optimize the design

Drastic decrease in firmware development time!

Today we'll use **Xilinx Vivado HLS** [*]

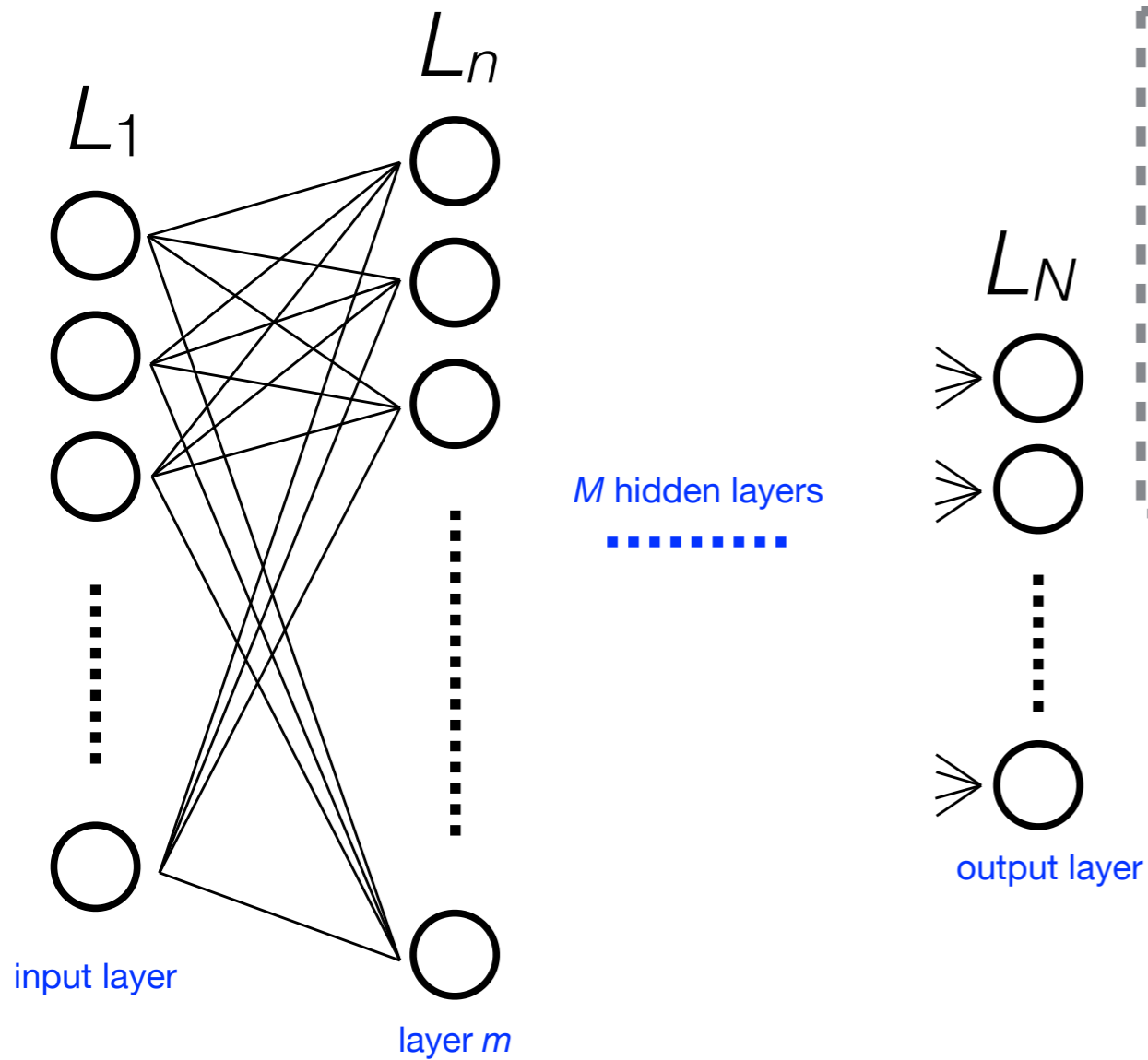


[*] https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf

Jargon

- LUT - Look Up Table aka 'logic' - generic functions on small bitwidth inputs. Combine many to build the algorithm
- FF - Flip Flops - control the flow of data with the clock pulse. Used to build the pipeline and achieve high throughput
- DSP - Digital Signal Processor - performs multiplication and other arithmetic in the FPGA
- BRAM - Block RAM - hardened RAM resource. More efficient memories than using LUTs for more than a few elements
- HLS - High Level Synthesis - compiler for C, C++, SystemC into FPGA IP cores
- HDL - Hardware Description Language - low level language for describing circuits
- RTL - Register Transfer Level - the very low level description of the function and connection of logic gates
- Latency - time between starting processing and receiving the result
 - Measured in clock cycles or seconds

Neural network inference

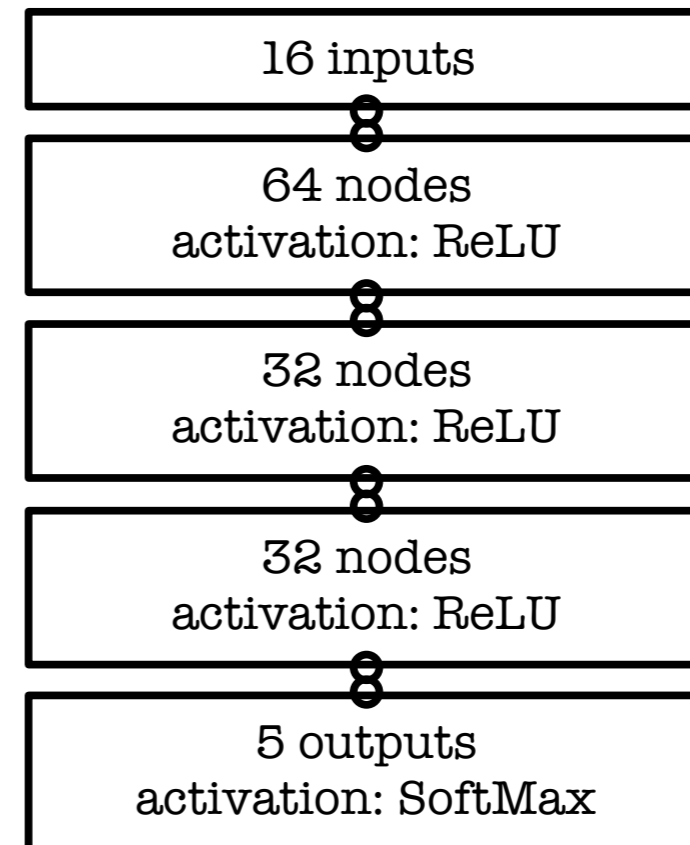


$$\mathbf{x}_n = g_n(\mathbf{W}_{n,n-1}\mathbf{x}_{n-1} + \mathbf{b}_n)$$

↗ activation function
precomputed and stored in BRAMs

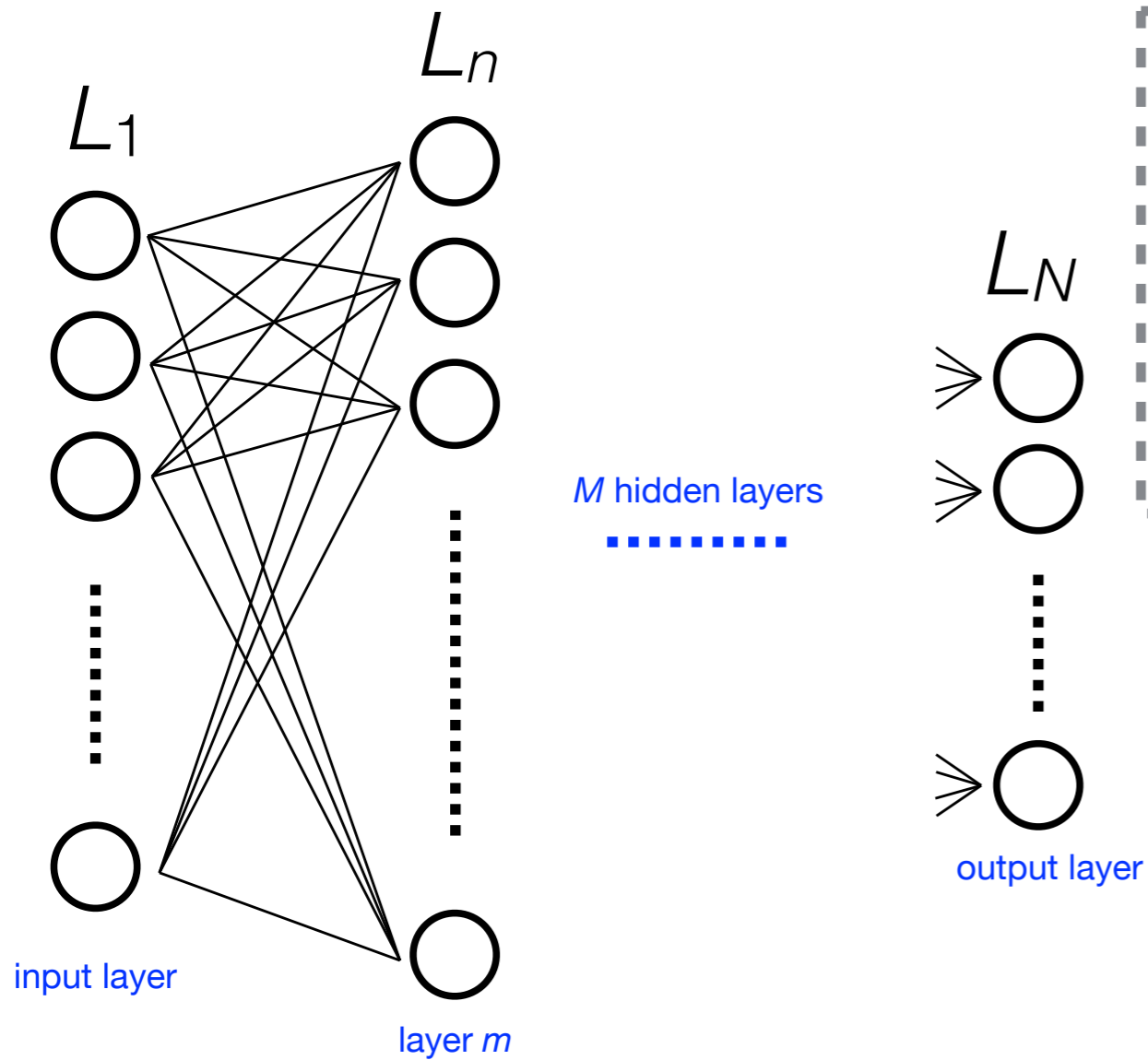
↑ multiplication
DSPs

↖ addition
logic cells



$$N_{\text{multiplications}} = \sum_{n=2}^N L_{n-1} \times L_n$$

Neural network inference

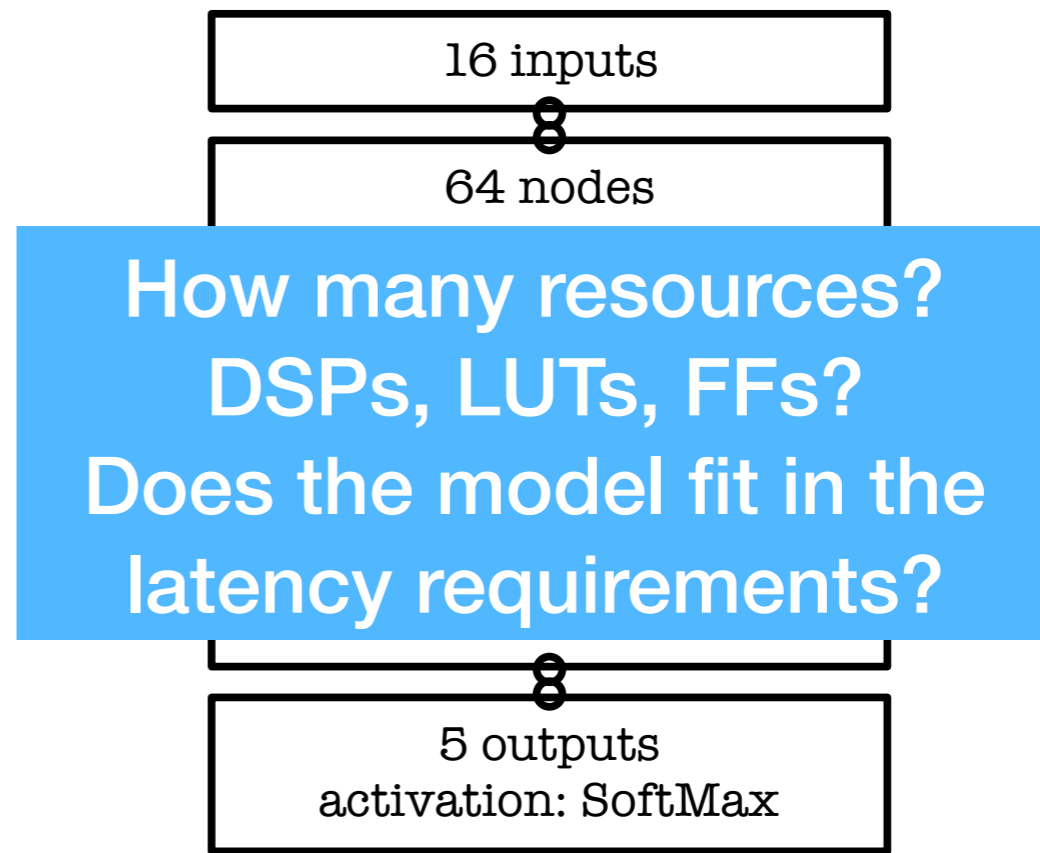


$$\mathbf{x}_n = g_n(\mathbf{W}_{n,n-1}\mathbf{x}_{n-1} + \mathbf{b}_n)$$

activation function
precomputed and stored in BRAMs

multiplication
DSPs

addition
logic cells



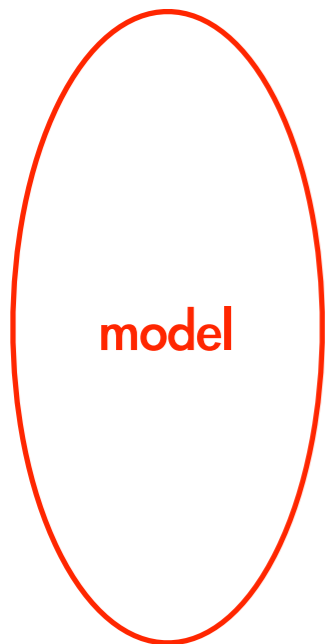
$$N_{\text{multiplications}} = \sum_{n=2}^N L_{n-1} \times L_n$$

Today you are going to implement a NN on FPGA with this package:

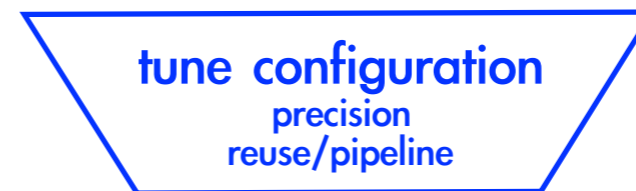
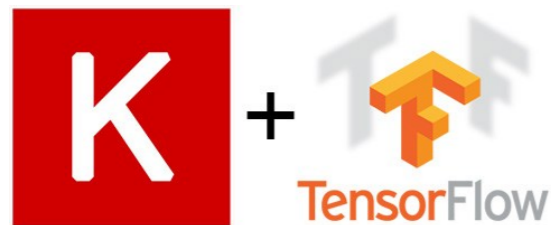
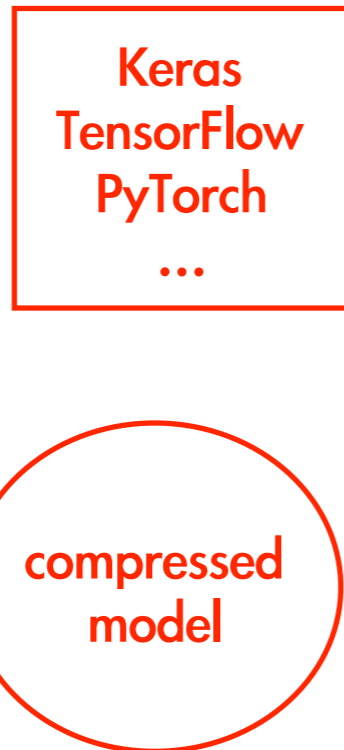
high level synthesis for machine learning

<https://arxiv.org/abs/1804.06913>

PYTORCH



Usual ML software workflow



<https://hls-fpga-machine-learning.github.io/hls4ml/>

Efficient NN design for FPGAs

FPGAs provide huge flexibility

Performance depends on how well you take advantage of this

Constraints:

Input bandwidth
FPGA resources
Latency

Today you will learn how to optimize your project through:

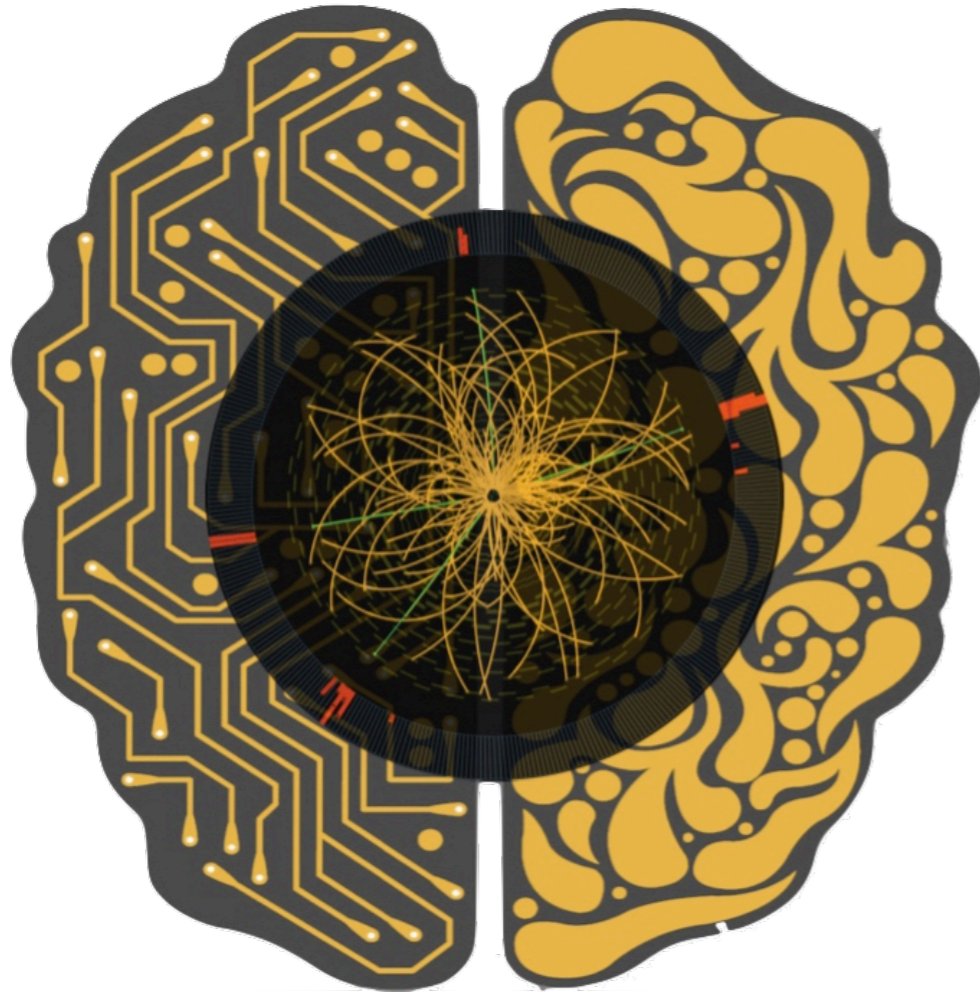
- **compression:** reduce number of synapses or neurons
- **quantization:** reduces the precision of the calculations (inputs, weights, biases)
- **parallelization:** tune how much to parallelize to make the inference faster/slower versus FPGA resources

NN TRAINING

FPGA PROJECT
DESIGNING

Today's hls4ml hands on

- First part:
 - take confidence with the package, its functionalities and design synthesis by running with one of the provided trained NN
 - learn how to read out an estimate of FPGA resources and latency for a NN after synthesis
 - learn how to optimize the design with quantization and parallelization
- Second part:
 - learn how to run the design on Amazon Web Services FPGAs with SDAccel
 - timing and resources studies after running on real FPGA
- Third part:
 - learn how to do model compression and its effect on the FPGA resources/latency



hls4ml Tutorial

Part I. Introduction - Hands On

Efficient NN design: quantization

ap_fixed<width,integer>

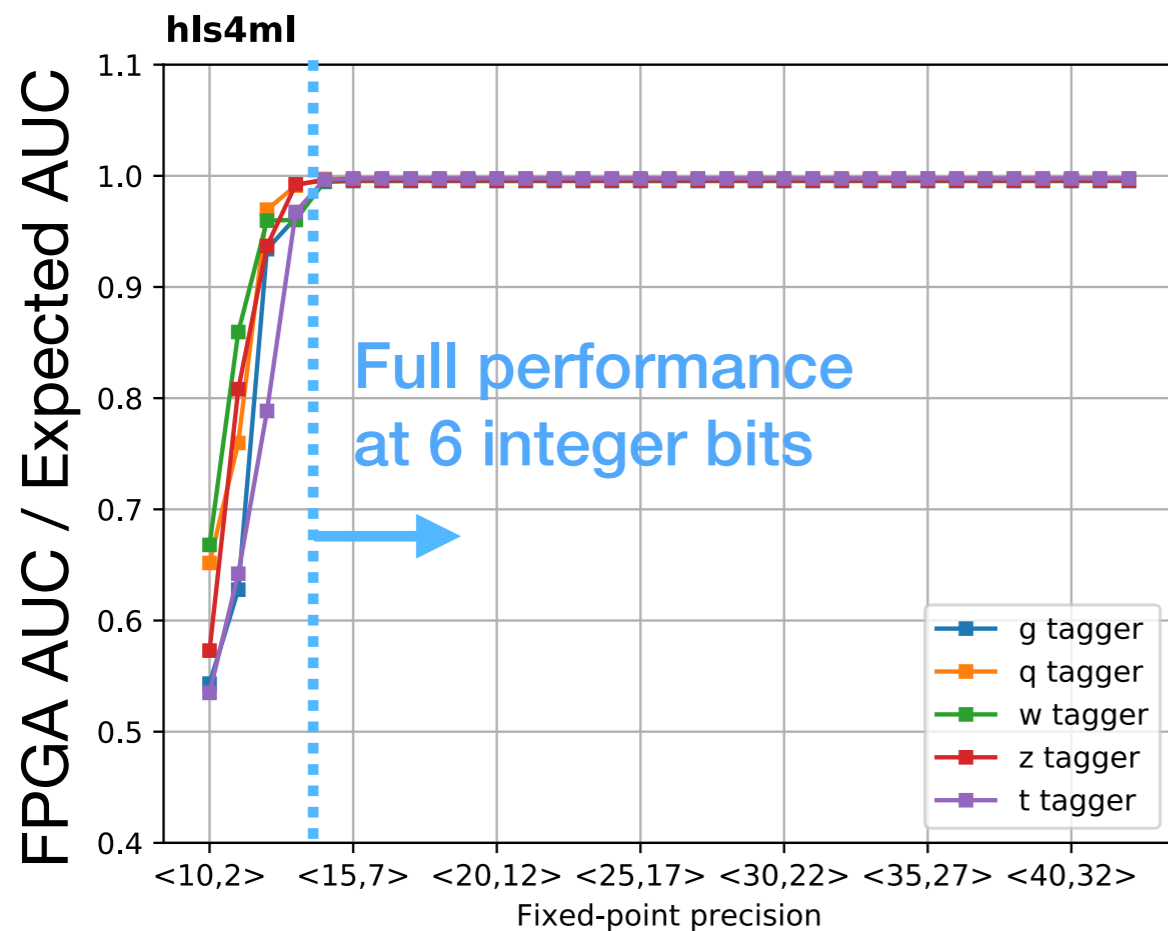
0101.1011101010



- Quantify the performance of the classifier with the AUC
- Expected AUC = AUC achieved by 32-bit floating point inference of the neural network

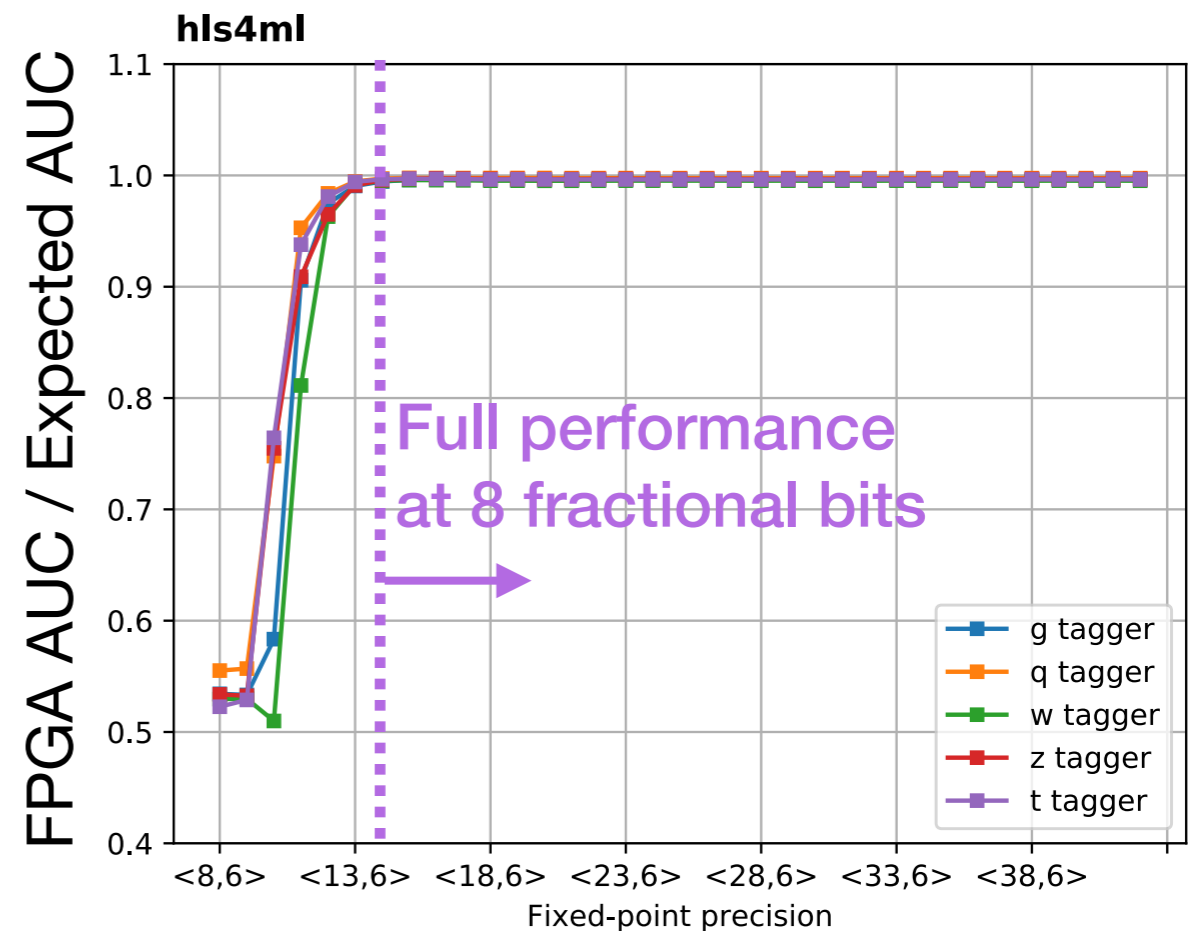
Scan integer bits

Fractional bits fixed to 8



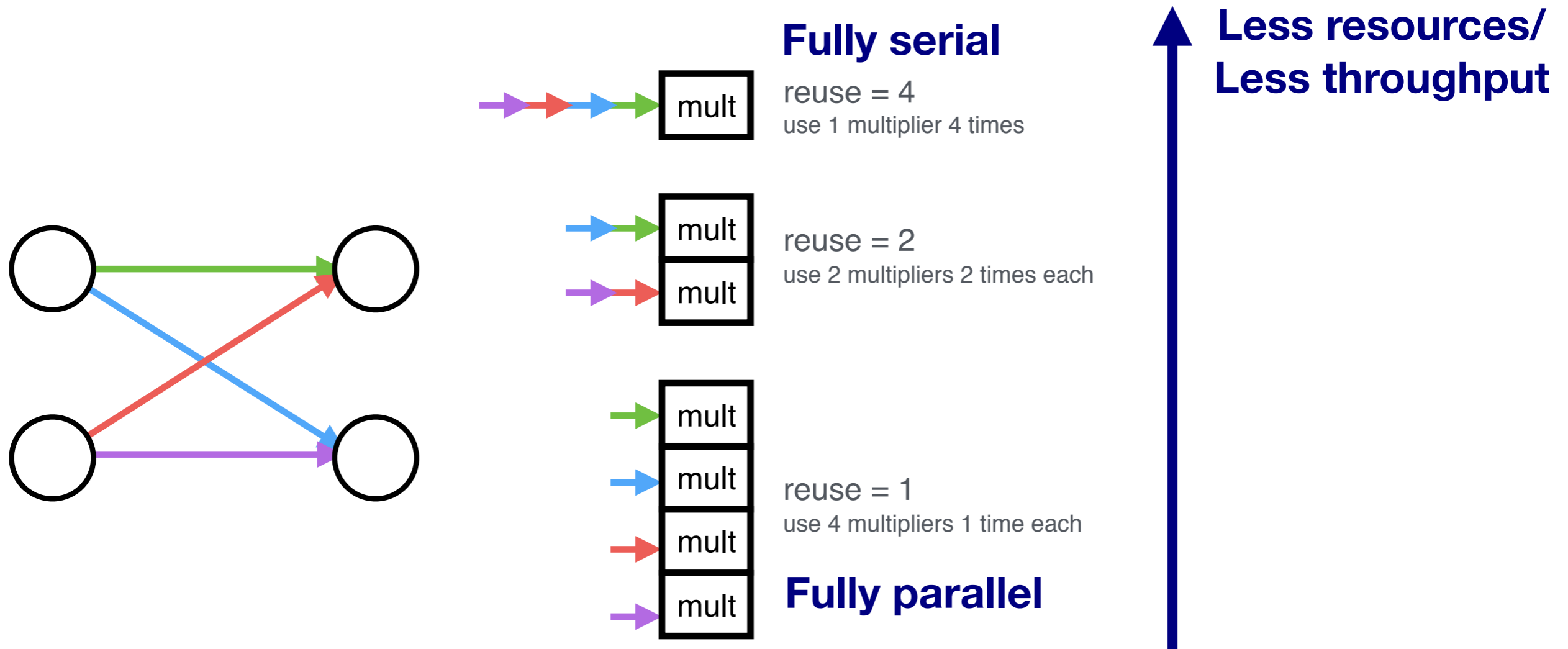
Scan fractional bits

Integer bits fixed to 6



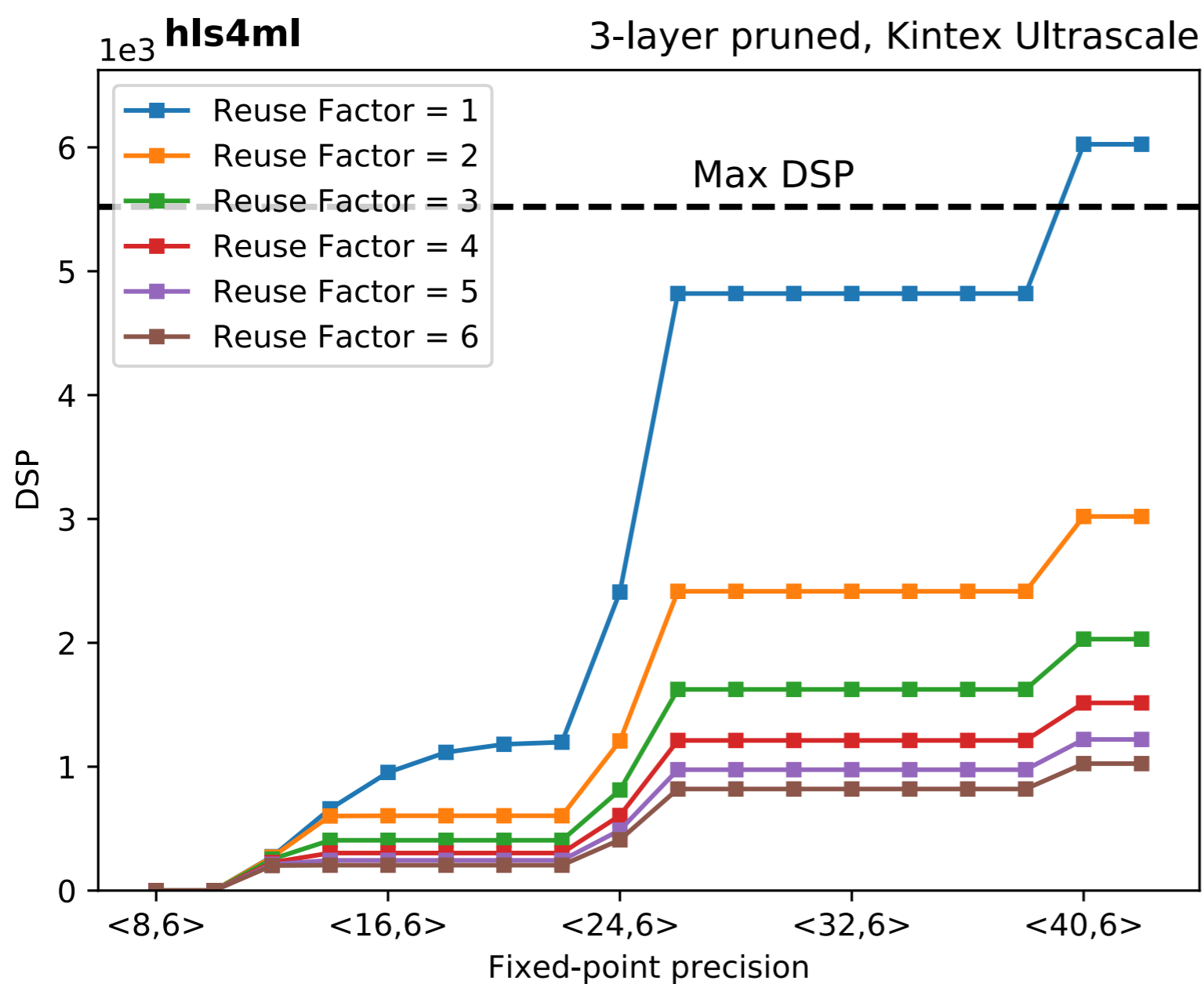
Efficient NN design: parallelization

- Trade-off between latency and FPGA resource usage determined by the parallelization of the calculations in each layer
- Configure the “reuse factor” = number of times a multiplier is used to do a computation



Reuse factor: how much to parallelize operations in a hidden layer

Parallelization: DSP usage



Fully parallel
Each mult. used 1x

Each mult. used 2x

Each mult. used 3x

⋮

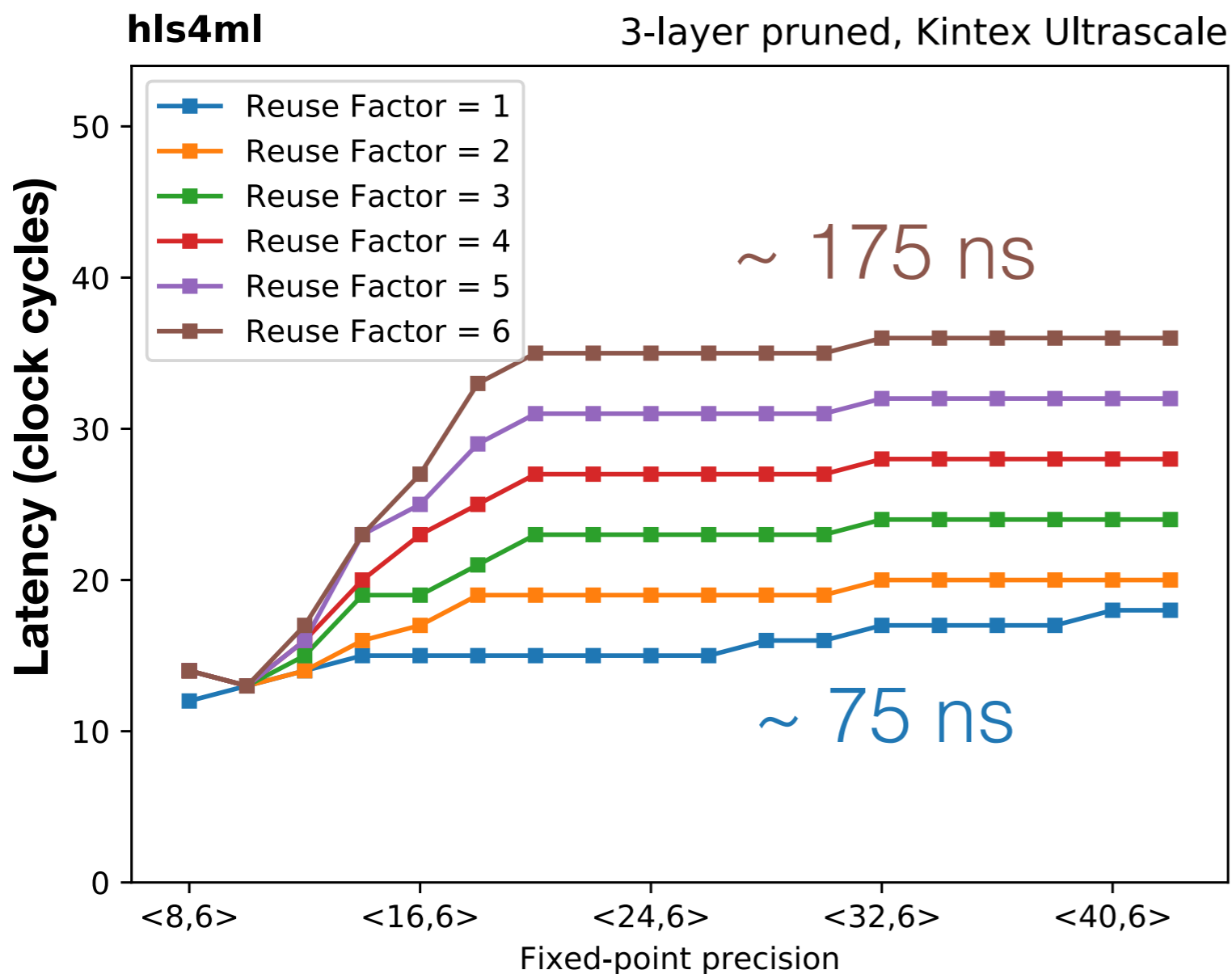
More resources

Longer latency

Parallelization: Timing

Latency of layer m

$$L_m = L_{\text{mult}} + (R - 1) \times II_{\text{mult}} + L_{\text{activ}}$$



Longer latency

Each mult. used 6x

⋮

Each mult. used 3x

⋮

Fully parallel

Each mult. used 1x



More resources

The config.yml file

- The model to translate
 - Some test vectors for simulation (check precision)
 - Output directory / name
 - Target FPGA, clock speed
 - Model data precision and parallelisation
 - More fine grained data precision and parallelisation
 - Per-layer, or per-layer type
- ```
KerasJson: keras/KERAS_3layer.json
KerasH5: keras/KERAS_3layer_weights.h5
#InputData: keras/KERAS_3layer_input_features.dat
#OutputPredictions: keras/KERAS_3layer_predictions.dat
OutputDir: my-hls-test
ProjectName: myproject
XilinxPart: xcku115-flvb2104-2-i
ClockPeriod: 5

IOType: io_parallel # options: io_serial/io_parallel
HLSConfig:
 Model:
 Precision: ap_fixed<16,6>
 ReuseFactor: 1
LayerType:
Dense:
ReuseFactor: 2
Strategy: Resource
Compression: True
```

# Hands On - Setup

- [https://github.com/holzman/course\\_material/blob/fml/part0\\_setup.md](https://github.com/holzman/course_material/blob/fml/part0_setup.md)
- Download the ssh key from: [https://www.dropbox.com/s/yd5yiov0onva6qt/fastml\\_rsa?dl=0](https://www.dropbox.com/s/yd5yiov0onva6qt/fastml_rsa?dl=0)
- Find your IP from: <https://tinyurl.com/hls4ml-demo>
- Then in a terminal:
 

```
chmod 600 fastml_rsa
```

```
ssh -i fastml_rsa <your assigned IP>
```

(password)
- You are now connected to a VM with the Xilinx Vivado HLS suite installed

```

 ____ _____
_ \|								
	_							
_ \|								
	_							
 ____ _____

```

```

AMI Version: 1.6.0
Readme: /home/centos/src/README.md
GUI Setup Steps: /home/centos/src/GUI_README.md
GUI Setup script: /home/centos/src/scripts/setup_gui.sh
AMI Release Notes: /home/centos/src/RELEASE_NOTES.md
Xilinx Tools: /opt/Xilinx/
Developer Support: https://github.com/aws/aws-fpga/blob/master/README.md#developer-support
Centos Common code: /srv/git/centos-git-common
[centos@ip-172-31-10-12 ~]$

```

# Exercise

- Follow the step-by-step instructions at:
- [https://github.com/FPGA4HEP/course\\_material/blob/fml/part1\\_hls4ml\\_intro.md](https://github.com/FPGA4HEP/course_material/blob/fml/part1_hls4ml_intro.md)
- For the final part “Change precision of calculations and reuse factor”:
- Everybody pick a **Precision** and **Reuse Factor** from the spreadsheet
  - Put your name in the column, pick one that isn't already assigned
- [https://docs.google.com/spreadsheets/d/1xrFf3\\_-6G10wmYnZ8zuDM3SfCfUMe0\\_KOB8mW6S-E2E/edit#gid=0](https://docs.google.com/spreadsheets/d/1xrFf3_-6G10wmYnZ8zuDM3SfCfUMe0_KOB8mW6S-E2E/edit#gid=0)
- Put your results in the spreadsheet!
  - Plots are generated on the ‘Plots’ sheet


# Other Examples

- The FPGA workflow can take a long time, so here are some results from pre-compiled models...




# Large MLP

- 'Strategy: Resource' for larger networks and higher reuse factor
- Uses a slightly different HLS implementation of the dense layer to compile faster and better for large layers
- We use a different partitioning on the first layer for the best partitioning of arrays



```
KerasJson: keras/MNIST_model.json
KerasH5: keras/MNIST_model_weights.h5
#InputData: keras/MNIST_model_input_features.dat
#OutputPredictions: keras/MNIST_model_predictions.dat
OutputDir: my-hls-test
ProjectName: myproject
XilinxPart: xcku115-flvb2104-2-i
ClockPeriod: 5
```

```
IOType: io_parallel # options: io_serial/io_parallel
HLSConfig:
 Model:
 Precision: ap_fixed<16,6>
 ReuseFactor: 128
 Strategy: Resource
 LayerName:
 dense1:
 ReuseFactor: 112
```



A model trained on the MNIST digits classification dataset  
Architecture: 784 x 128 x 128 x 128 x 10  
Model accuracy: ~97%

**Can you calculate the number of DSPs it will use?**

(Don't cheat and look ahead)

# Large MLP

- It takes a while to synthesise, so here's one I made earlier...
- The DSPs should be:  $(784 \times 128) / 112 + (2 \times 128 \times 128 + 128 \times 10) / 128 = 1162$  🙌

```
=====
== Performance Estimates
=====
```

```
+ Timing (ns):
 * Summary:
+-----+-----+-----+-----+
| Clock | Target| Estimated| Uncertainty|
+-----+-----+-----+-----+
| lap_clk | 5.00| 4.375| 0.62|
+-----+-----+-----+-----+
```

```
+ Latency (clock cycles):
 * Summary:
+-----+-----+-----+-----+
| Latency | Interval | Pipeline |
| min | max | min | max | Type |
+-----+-----+-----+-----+
| 518| 522| 128| 128| dataflow |
+-----+-----+-----+-----+
```



|| determined by the largest reuse factor

```
=====
== Utilization Estimates
=====
```

```
* Summary:
+-----+-----+-----+-----+
| Name | BRAM_18K| DSP48E| FF | LUT |
+-----+-----+-----+-----+
| DSP | -| -| -| -|
| Expression | -| -| 0| 3144|
| FIFO | 1394| -| 28998| 46116|
| Instance | 568| 1162| 140203| 166361|
| Memory | -| -| -| -|
| Multiplexer | -| -| -| 7002|
| Register | -| -| 778| -|
+-----+-----+-----+-----+
| Total | 1962| 1162| 169979| 222623|
+-----+-----+-----+-----+
| Available SLR | 2160| 2760| 663360| 331680|
+-----+-----+-----+-----+
| Utilization SLR (%) | 90| 42| 25| 67|
+-----+-----+-----+-----+
| Available | 4320| 5520| 1326720| 663360|
+-----+-----+-----+-----+
| Utilization (%) | 45| 21| 12| 33|
+-----+-----+-----+-----+
```

# Binary & Ternary Neural Networks

- Constrain the weights (and optionally activations) to  $\pm 1$  (binary) or  $\pm 1$  & 0 (ternary)
  - Can use a few LUTs to perform ‘activation \* weight’ products instead of DSPs
- Consider it a form of model compression
- ...but typically need to increase model size to retain performance
  - So in hls4ml this goes hand in hand with `Strategy: Resource`
- There are example jet tagging B/TNNs in the hls4ml repo:
  - `KERAS_3layer_binary_smaller{.json, _weights.h5}` - Binary dense layer & our optimized ‘Batch Normalization + Binary Tanh’ layer (1 bit weights & activations)
  - `KERAS_3layer_binarydense_relu_max{.json, _weights.h5}` - Binary dense layer, batch normalization and ‘clipped ReLU’ activation (~few bits activations)
  - `KERAS_3layer_ternary_small{.json, _weights.h5}` - Ternary dense layer, batch normalization and ternary tanh (2 bit weights & activations)

# Binary MNIST Model

- Now a model trained on the MNIST digits, with the same architecture as the last one, but now with 1 bit weights: 93% accuracy
- Now we use 0 DSPs! The LUTs look a bit high, but note these always go down a lot after the later stages of Xilinx compilation flow (goes down to 14%)

```
=====
== Performance Estimates
=====
```

```
+ Timing (ns):
 * Summary:
+-----+-----+-----+-----+
| Clock | Target| Estimated| Uncertainty|
+-----+-----+-----+-----+
| lap_clk | 5.00 | 4.375 | 0.62 |
+-----+-----+-----+-----+
```

```
+ Latency (clock cycles):
 * Summary:
+-----+-----+-----+-----+
| Latency | Interval | Pipeline |
| min | max | min | max | Type |
+-----+-----+-----+-----+
| 516 | 520 | 128 | 128 | dataflow |
+-----+-----+-----+-----+
```

```
=====
== Utilization Estimates
=====
```

```
* Summary:
+-----+-----+-----+-----+
| Name | BRAM_18K | DSP48E | FF | LUT |
+-----+-----+-----+-----+
| DSP | - | - | - | - |
| Expression | - | - | 0 | 3144 |
| FIFO | 1394 | - | 28998 | 39646 |
| Instance | 59 | 0 | 65662 | 209840 |
| Memory | - | - | - | - |
| Multiplexer | - | - | - | 7002 |
| Register | - | - | 778 | - |
+-----+-----+-----+-----+
| Total | 1453 | 0 | 95438 | 259632 |
+-----+-----+-----+-----+
| Available SLR | 2160 | 2760 | 663360 | 331680 |
+-----+-----+-----+-----+
| Utilization SLR (%) | 67 | 0 | 14 | 78 |
+-----+-----+-----+-----+
| Available | 4320 | 5520 | 1326720 | 663360 |
+-----+-----+-----+-----+
| Utilization (%) | 33 | 0 | 7 | 39 |
+-----+-----+-----+-----+
```