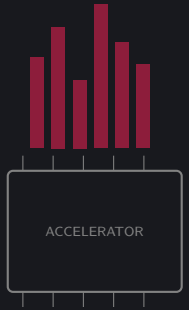
TÉLÉCOM PARIS · SETI MASTER · LECTURE 01

Embedded & Frugal AI

From neural networks to accelerators on silicon



History, foundations, metrics, accelerators and FPGAs

2026–2027 · Session 1 of 8

The course: eight sessions – a lecture then its lab in S1–S4, labs in S5–S8

S1 Wed 9 Dec 2026
Introduction

Lecture: foundations, PyTorch, metrics, accelerators, FPGAs

Lab 1: tagged runs, MLP vs CNN on FashionMNIST, first PTQ XMODEL

S2 Wed 6 Jan 2027
Quantization

Lecture: scale, rounding, calibration, PTQ, QAT

Lab 2: PTQ vs QAT; compile; first KV260 run with XIR/VART

S3 Wed 13 Jan 2027
Pruning and search

Lecture: pruning, recovery, architecture search and OFA

Lab 3: pruning policies on a small CNN, automatic pruning of VGG6-BN; deployed and profiled

S4 Wed 27 Jan 2027
Distillation

Lecture: soft targets, temperature, attention transfer

Lab 4: CE vs KD vs AT students on MNIST, deployed

S5 Wed 3 Feb 2027
Frugal AI on the KV260

Lab 5: standard vs frugal (pruned + KD + INT8), measured, live camera

S6 Wed 10 Feb 2027
HLS: a Conv2D layer

Lab 6: Conv2D versions 00–04, OC_PAR sweep; fuse Conv+ReLU+MaxPool

S7 Wed 24 Feb 2027
MNIST capstone (I)

Lab 7: compact Conv–Pool–FC (6 810 parameters, 31 096 MACs), INT8 export; implement FC and the DATAFLOW top, bit-exact C simulation

S8 Wed 3 Mar 2027
MNIST capstone (II)

Lab 7: parallelism per layer against resources; the accelerator on the KV260, measured

S1–S4: lecture then lab; S5–S8: labs. See the course plan for the calendar. TP06 studies HLS Conv2D; TP07 builds the compact MNIST CNN.

What you will be able to do in March

Phase 1 – the model, with Vitis AI S1–S5

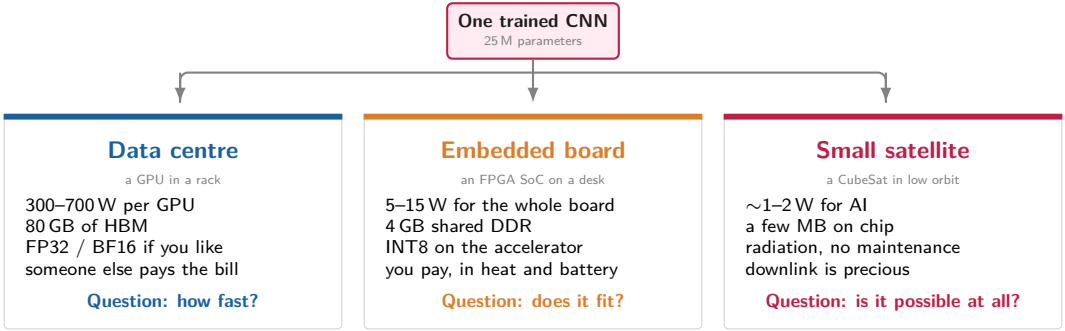
- compute parameters, MACs, activation memory and model size of a CNN;
- train, compare and select models with a reproducible, tagged workflow;
- quantize, prune and distil a model – and say which one a requirement justifies;
- compile for the KV260 DPU, run with XIR/VART, and measure accuracy, FPS and energy;
- diagnose deployment failures: unsupported operators, batch size, preprocessing.

Phase 2 – the hardware, with HLS S6–S8

- turn a convolution loop nest into hardware: line buffers, pipelining, unrolling;
- read HLS reports: initiation interval, latency, DSP, BRAM, LUT;
- fuse layers and stream them with DATAFLOW, AXI interfaces and DMA;
- take a trained CNN to one fused accelerator, bit-exact with its INT8 model, and run it from Python on the KV260;
- choose the parallelism of each layer for a stated goal – and justify it with measured numbers.

The course is not about memorising tools. It is about choosing – and justifying with measurements – the right optimisation for a stated constraint.

One model. Three destinations.



The model does not change. Everything that decides whether it is *usable* lives outside the model.

The dataset: examples, labels, and three splits



The running example of this part: 1 797 handwritten digits, 8×8 pixels, 10 classes (scikit-learn `digits`). Each example is a pair (x_i, y_i): 64 numbers and one label.



Training set
the only data the optimiser ever sees.

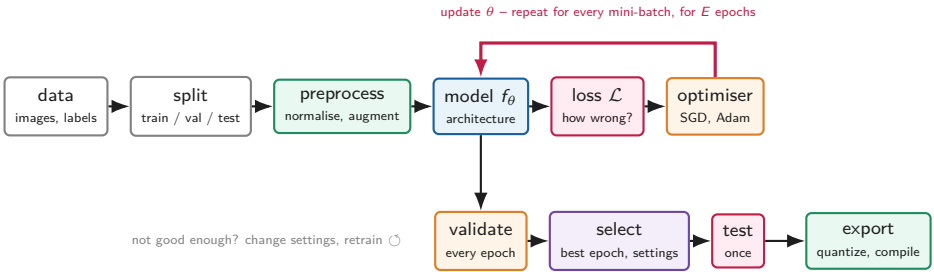
Validation set
used to pick hyper-parameters, epochs, quantization settings, pruning ratio...

Test set
touched once. If you tune on it, your number is optimistic – and your deployment will disappoint.

Preprocessing
normalisation statistics are computed on the training set – and the *same* transform must run on the board.

Every accuracy number in your lab reports must say which split it was measured on.

The workflow of every lab, in one picture



Batch
the samples in one gradient step (e.g. 64)

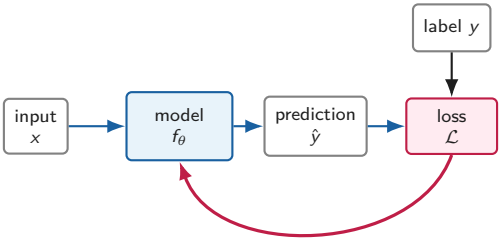
Iteration
one forward + backward + update on one batch

Epoch
one pass over the whole training set

Hyper-parameter
chosen by you, not learned: lr, batch, width. . .

Quantization, pruning and distillation each plug into one box of this picture. Knowing which box is half the lab.

Supervised learning is one loop, repeated



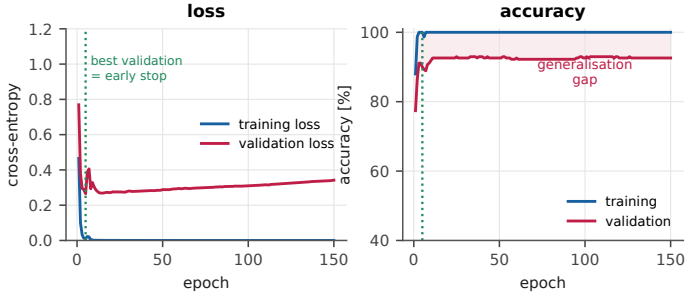

```python
for x, y in loader:
    y_hat = model(x) # forward
    loss = criterion(y_hat, y)
    loss.backward() # gradients
    optimizer.step() # nudge theta
    optimizer.zero_grad()
```

Only the first line survives deployment. The rest is training.

$$\theta^* = \arg\min_\theta \frac{1}{N} \sum_i \mathcal{L}\big(f_\theta(x_i), y_i\big)$$

Repeat a few million times and the numbers settle. From that moment on, only `model(x)` ever runs – billions of times.

Reading a training curve



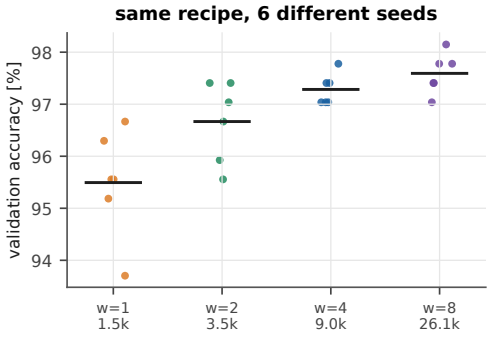
A $64 \to 512 \to 512 \to 10$ network trained on only 150 digits. Training loss goes to zero; training accuracy reaches 100%.

Validation loss is lowest at epoch 5 and then **rises**: the model becomes over-confident on the examples it gets wrong. Validation accuracy stalls at $\approx$93%.

Computed for this lecture (NumPy MLP, Adam, scikit-learn `digits`).

Always plot training *and* validation curves. The gap between them is the first thing to look at.

Is a one-point gain real? Seeds and controlled comparisons



Computed for this lecture: $64 \to 16w \to 16w \to 10$ MLPs on scikit-learn `digits`, 40 epochs, 6 seeds each.

Every dot is the *same* recipe; only the random seed changes (initial weights, batch order).

- width 1 spans **3 points** (93.7–96.7%) on seeds alone;
- width 8 beats width 4 by **0.3 points** on average – for 2.9× the parameters, inside the spread.

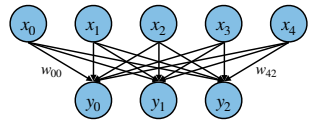
Change one factor at a time
same data split, seed, epochs and preprocessing for both runs.

Repeat when the gap is small
3 seeds or more; report mean ± std.

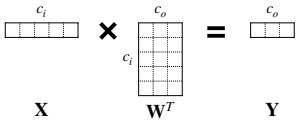
One reference row
compare every variant with the same float source model.

A difference smaller than the seed-to-seed spread is not a result.

Fully-connected layer = matrix–vector product = MACs



$$y_i = \sum_j w_{ij} x_j + b_i$$



Multiply–accumulate (MAC)
$a \leftarrow a + b \cdot c$ – one multiply, one add: the atom of deep learning.

Matrix–vector (FC layer, batch 1)
MACs $= c_i \cdot c_o$ and $c_i \cdot c_o$ weights: every weight is used *once*.

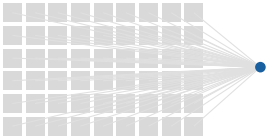
Matrix–matrix (GEMM)
MACs $= m \cdot n \cdot k$ convolution and attention both lower to GEMM.

Figure: MIT HAN Lab 6.5940, Lecture 02.

An AI accelerator is, at its core, a machine for feeding as many MAC units as possible.

Convolution: the first efficiency idea of this lecture

IF EVERY PIXEL GETS ITS OWN WEIGHT



a $224 \times 224 \times 3$ image, one dense layer

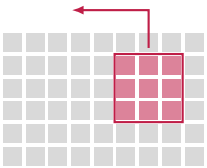
$150\,528 \times 1\,000$

$\approx$ **150 million**

weights – for a single layer

$\div$ *87 000*

IF ONE SMALL FILTER SLIDES EVERYWHERE



the same 3×3 filter at every position

$3 \times 3 \times 3 \times 64$

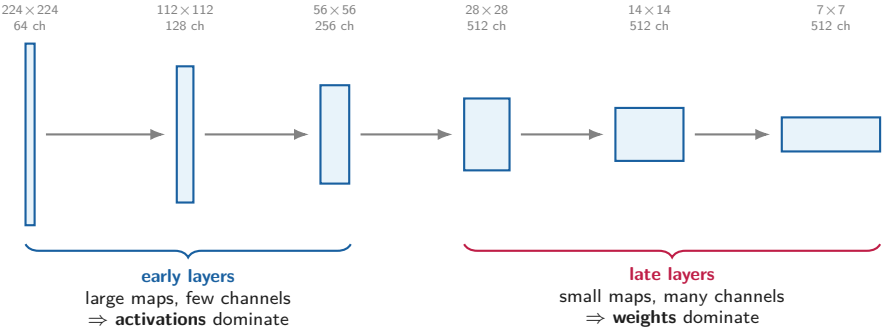
1 728

weights – for 64 feature detectors

$$Y_{o,h,w} = \sum_{c=1}^{C_{in}} \sum_{u,v=1}^{K} W_{o,c,u,v}\, X_{c,\, h+u,\, w+v} + b_o \qquad \text{MACs} = H_o W_o\, C_{in} C_{out} K^2$$

Two assumptions – a feature is local, and it means the same thing everywhere – bought five orders of magnitude. Every later technique is another trade of an assumption for a resource.

The shape of a CNN, and where the cost hides



Maps shrink, channels grow. "Where does the memory go?" has two different answers depending on which end you stand at.

The nine numbers, and what each one really asks

Accuracy
Is the answer good enough *for this task*?

Parameters
How much storage does the model occupy at rest?

MACs / FLOPs
How much arithmetic does one input demand?

Model memory
Do the weights fit in the memory I have?

Activation memory
Does the *largest intermediate* tensor fit?

Memory traffic
How many bytes cross the chip boundary?

Latency
How long from input to answer, once?

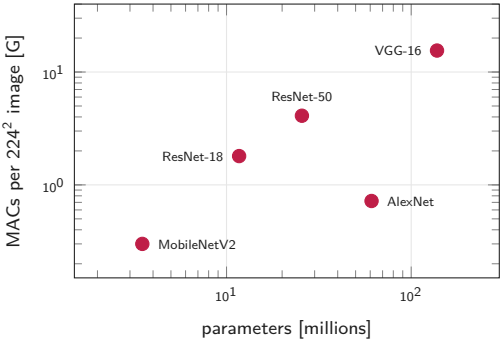
Throughput
How many inputs per second, in steady state?

Energy / inference
How much battery, heat or solar budget per answer?

A tenth, implicit in all of them: hardware utilisation – what fraction of the silicon you paid for is doing useful work.

These are not nine ways of saying the same thing. Improving one routinely makes another worse – which is exactly why they need separate names.

Parameters and operations are not the same number



VGG-16 vs AlexNet
2.3× the parameters, but 21× the arithmetic.

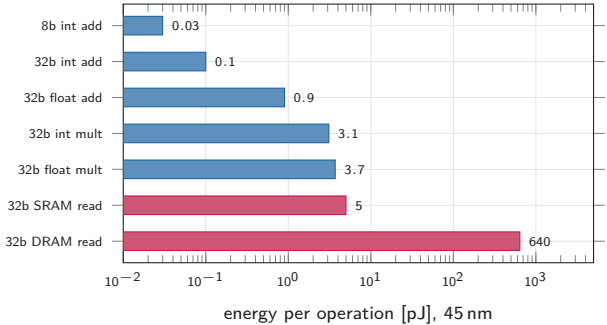
ResNet-50 vs VGG-16
5× fewer parameters *and* 4× less arithmetic – while being more accurate.

Where a parameter sits in the network decides how often it is used: a convolution weight is reused at every pixel, a classifier weight once.

Published parameter and MAC counts of the reference implementations at 224 × 224 input.

A small download is not a fast model.

Moving a number costs far more than computing with it



Fetching one 32-bit word from DRAM costs **640 pJ**. An 8-bit add costs **0.03 pJ**.

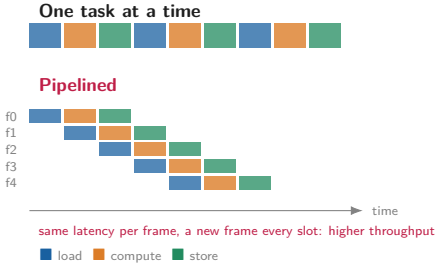
That is a factor of roughly **20 000**. The arithmetic is essentially free; *the commute is the job*.

On-chip SRAM sits in between: $\sim 100\times$ cheaper than DRAM.

Horowitz, *Computing's Energy Problem (and what we can do about it)*, ISSCC 2014. Note the logarithmic axis.

Almost every technique we will study is, underneath, a way to move fewer bytes.

Latency and throughput are different questions



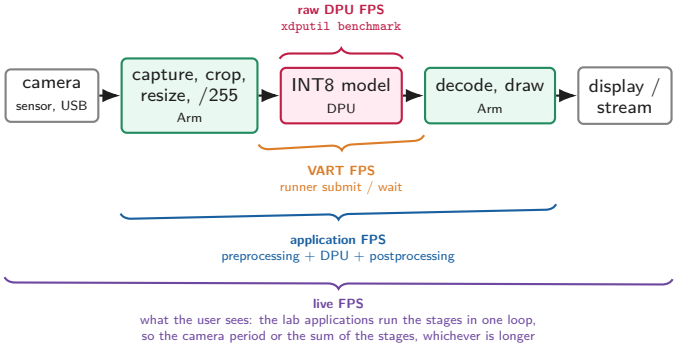
Latency matters for a control loop: a camera, a trigger, a brake.

Throughput matters for a video stream or a satellite pass: a batch of images.

Pipelining overlaps load, compute and store: throughput rises, latency does not fall. Batching does the same – at the price of waiting for the batch.

Scheduling is an efficiency technique: it can multiply throughput without touching the weights or the arithmetic.

Which FPS? Four ways to measure a model on the board



`vaitrace`: where does the time go?
a timeline of DPU runs and runtime calls; it explains a number, it does not replace the accuracy test.

`xmutil xlnx_platformstats -p`: power
sample SOM power (excluding carrier and USB camera) during a controlled workload; energy per input $= \bar{P} \cdot t \,/\, N$.

Always say which FPS: thousands on the DPU, thirty from the camera.

The efficiency toolbox: four families, four different things to edit

Architecture

edits the *shape*

Depthwise-separable convolutions, bottlenecks, smaller inputs, neural architecture search.

Pruning

edits the *count*

Pruning: individual weights (unstructured) or whole channels (structured).

Quantization

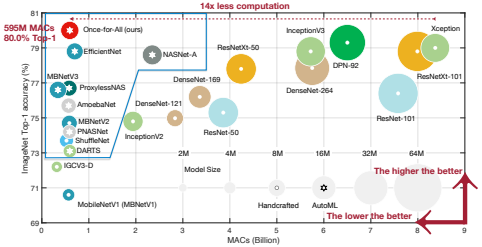
edits the *precision*

Quantization: FP32 → INT8 or below; post-training or quantization-aware.

Distillation

edits the *training signal*

Knowledge distillation: a small student learns from a large teacher's outputs.



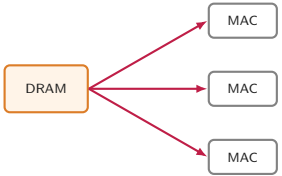
Architecture search moves the whole frontier: Once-for-All reaches 80% top-1 at 595 M MACs – 14× less computation than models of similar accuracy.

A fifth family edits **nothing in the model at all**: compilation, fusion, scheduling and dataflow – Parts 7 and 8.

Figure: MIT HAN Lab 6.5940, Lecture 01; Cai et al., *Once-for-All*, ICLR 2020.

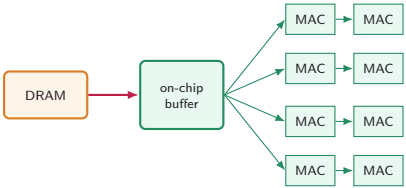
The real trick is reuse, not speed

One value fetched, one multiply



Every operand crosses the chip boundary. At 640 pJ per fetch, the memory bill dominates everything.

One value fetched, reused many times

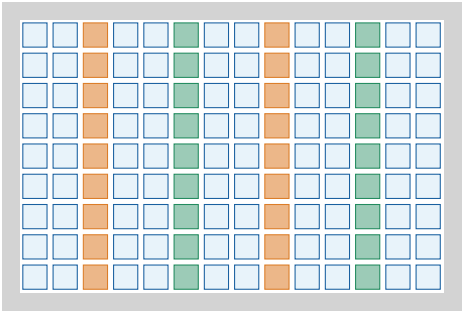


The same weight feeds a row of multipliers; the same activation a column. One fetch, dozens of MACs.

This is the entire idea behind systolic arrays and every neural accelerator since. **Arithmetic intensity is a design parameter**, not a property of the model.

Reuse is how you move a workload rightwards on the roofline: off the bandwidth slope and onto the compute ceiling.

What is inside an FPGA?



LUT + flip-flop
6-input lookup tables: any small logic, control, low-bit MACs; registers for pipelining

DSP slice
27×18 multiplier + 48-bit accumulator: *the* MAC unit (two INT8 MACs with packing)

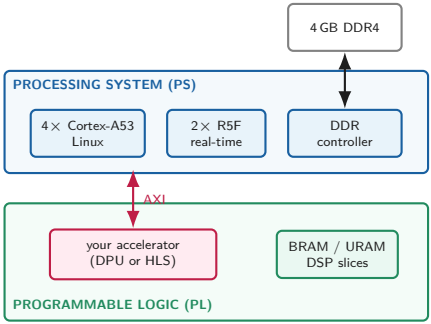
BRAM / URAM
36 Kb / 288 Kb on-chip blocks: tiles, FIFOs, line buffers

Routing
programmable wires connecting everything – flexible, but costly in area and power

AMD UltraScale architecture guides (UG579 DSP, UG573 memory); INT8 packing: AMD WP486.

An FPGA is a sea of small resources: you build the accelerator by deciding how to wire them together.

The platform: a Zynq UltraScale+ MPSoC (Kria KV260)



256K
system logic cells

1 248
DSP slices

26.6 Mb
on-chip SRAM (≈3.3 MB)

7.5 W
typical power

AMD Kria K26 SOM product page / data sheet DS987.

The PS runs Linux and orchestration; the PL hosts the accelerator. The AXI link between them is part of the design.

The overlay you will actually use: the DPU

B4096 the standard Vitis AI DPU configuration

$$\underset{\text{pixels in parallel}}{8} \times \underset{\text{input channels}}{16} \times \underset{\text{output channels}}{16} \times \underset{\text{ops per MAC}}{2} = \underset{\substack{\text{operations per}\\ \text{clock, per core}}}{4\,096}$$

At 300 MHz one B4096 core peaks at ≈1.2 INT8 TOPS. The KV260 hosts one; a ZCU102 up to three.

That is the entire compute budget. Everything else is a question of *keeping it fed*.

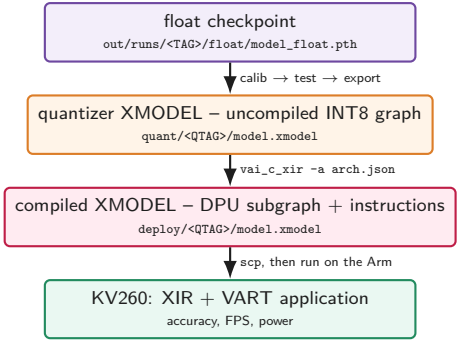
Notice the shape: a layer with 8 channels wastes half of the channel parallelism; one with 24 wastes a quarter (24 of 32 slots in two passes).

Architecture choices should be multiples of 16 – a constraint that never appears in a paper about accuracy.

AMD PG338 (DPUCZDX8G): peak GOPS = pixel × input-channel × output-channel parallelism × 2 × frequency.

The accelerator has a shape. Models that match it run at high utilisation; models that do not, run at half – with identical FLOPs.

Two XMODELs and a batch of one



Why batch 1?
The XMODEL records the shapes it saw. Export after a batch-128 evaluation and the compiler warns "DPU prefers input batch 1, but it is 128" and moves layers to the CPU.

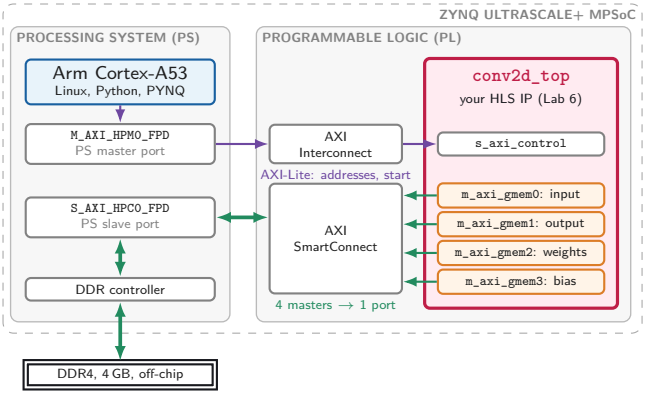
One DPU, one fingerprint
`arch.json` must describe the DPU loaded on your board (B4096). A different fingerprint means a different binary.

Strict DPU mapping
one batch-1 DPU computation partition; only the topology-checked terminal output `fix2float` conversion is allowed on CPU.

Never copy the wrong file
only the *compiled* XMODEL goes to the board.

The compiled XMODEL must match the board's DPU fingerprint and batch-1 input shape.

What you will build in S6–S8: HLS IPs inside the KV260



Lab 6 block design: control through `M_AXI_HPM0_FPD`, data through `S_AXI_HPC0_FPD`.

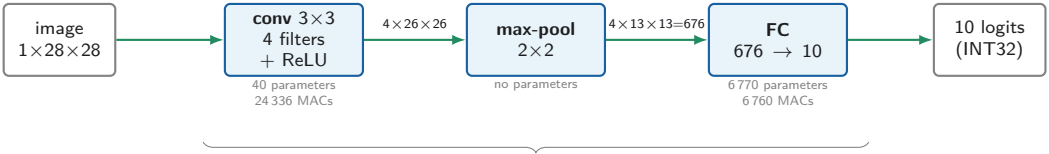
Control path: AXI-Lite
Python (PYNQ) allocates DDR buffers, writes their addresses into the IP's registers and starts it.

Data path: `m_axi`
the IP is itself a bus master: it reads inputs, weights and biases and writes outputs through a high-performance port into the shared DDR.

Inside: streams
line buffers are `hls::stream` FIFOs; the window is a small register array.

S6 (Lab 6): one Conv2D IP that masters DDR itself, so no DMA engine is needed. S7–S8 (Lab 7): your whole CNN as one streaming accelerator, fed by an AXI DMA.

The network you will build in Lab 7



one fused INT8 accelerator on the FPGA: 6 810 parameters, 31 096 MACs per image; images in and logits out through an AXI DMA

From Lab 6

the same line buffers and parallel multipliers, now in every layer, chained as one streaming accelerator.

Bit-exact integers

trained in PyTorch, exported to INT8: the hardware must match the integer reference exactly, not the float model.

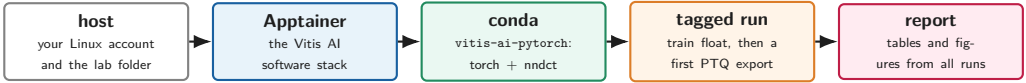
The frugal question

how much parallelism does each layer deserve? The slowest layer sets the image rate.

TP07_cnn_fpga_challenge: specs/compact_mnist.json and tools/cost_model.py; parameters include biases; MACs exclude bias additions.

Lab 7 is Lab 6 for a whole network: the slowest layer sets the pace, and every extra multiplier must earn its place.

Lab 1 in one picture



Script	Role
`source 00_source_env.sh`	host variables and container paths (*sourced*, not run)
`bash 01_host_sanity.sh`	checks the host and the container image – do not continue if it fails
`bash 02_enter_container.sh`	opens a shell inside the container, with the lab folder mounted
`source 03_activate_vai_conda.sh`	activates `vitis-ai-pytorch` and checks `torch`, `pytorch_nndct`
`bash 04_train_ptq.sh --tag ...`	one experiment: `--dataset --arch --width --epochs --seed`
`bash 05_make_report.sh --name N tags`	`out/report/N/`: `summary.csv` and `compare.png`

The host gives you the machine and the files; the container gives the toolchain; conda gives the exact Python environment.

One tag = one experiment

```
out/runs/cnn_w2_e3/
  meta.json                      arguments, sizes
  float/model_float.pth          weights
  float/train_log.csv            curves
  ptq/model_ptq.xmodel           INT8 export
  quant/<QTAG>/                  from Lab 2
  deploy/<QTAG>/                 compiled
  board_results/<QTAG>/          KV260
```

The tag is the identity

two runs that must be compared must not share a tag; an existing tag is refused (`--overwrite` replaces it on purpose).

The report reads folders, not memory

it rebuilds every table from `out/runs/`; if a figure looks wrong, open the run's `meta.json` and `train_log.csv`.

The same structure grows

Labs 2–5 add quantized, compiled and board results under the same tag.

Clean with care

`clean.sh --runs` deletes the experiments the report needs.

You never compare models from memory. You compare them through their run folders.

Before you start the lab

Checklist

1. Sanity first
`01_host_sanity.sh` passes before anything else.

2. Right context
inside the container, with `vitis-ai-pytorch` active.

3. One tag per run
Block A: architectures at a fixed epoch budget. Block B: epoch budgets for one architecture.

4. Report, then interpret
which model is justified from a frugal point of view – and why?

Then, in this order

Final task: CIFAR-10
one small and one large model, tagged runs, a report, and a one-paragraph verdict. Start the CIFAR-10 download during Block A.

Extension, the code you write: CPU latency
add inference time to the report: `no_grad`, warm-up, batch 1, p50/p95 ("Measuring latency properly", for reading).

Keep your run folders: Lab 2 starts from a float checkpoint and the same `out/runs/<TAG>/` structure.

This lab is about discipline, not accuracy: a clean, tagged, reproducible comparison is the foundation of every later lab.