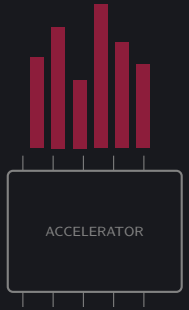
TÉLÉCOM PARIS · SETI MASTER · LECTURE 01

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

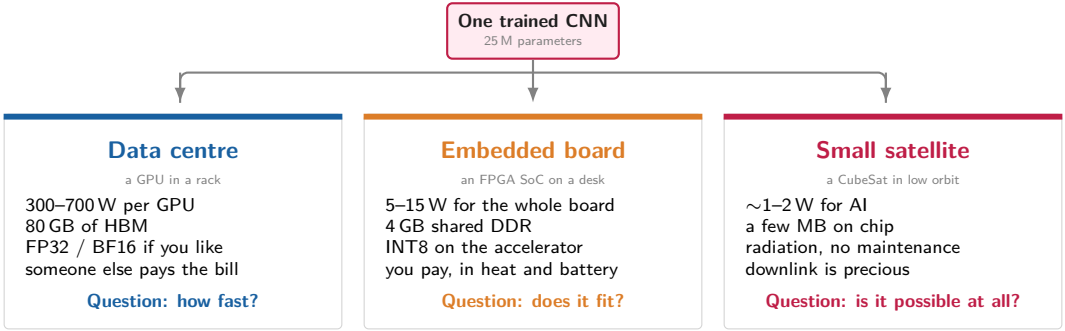
- compute parameters, MACs, activation memory and model size of a CNN;
- train, compare and select models with a reproducible, tagged workflow;
- quantize, prune and distil a model – and say which one a requirement justifies;
- compile for the KV260 DPU, run with XIR/VART, and measure accuracy, FPS and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

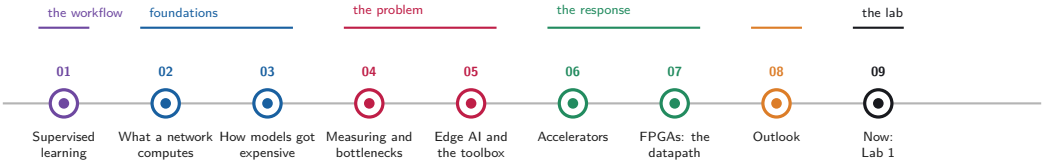
THE THREAD

How much useful intelligence can we obtain from a limited budget of computation, memory and energy?

This question comes back three times today. Each time we will be able to answer it more precisely.

By the end of the course, it is the question your lab reports answer.

Where we are going



Today is a map, not a manual – then Lab 1. The slides after the sources go further, for reading.

PART 1

Supervised learning: the workflow behind every lab

Data, splits, losses, gradient descent, and how to read a training curve – everything the labs assume you already know.

1

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.

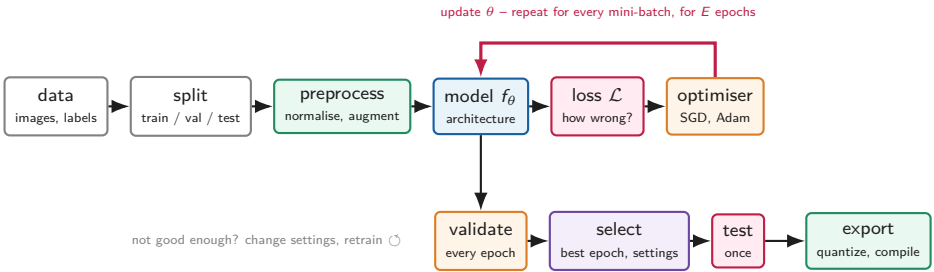
Validation set
used to pick hyper-parameters, epochs, quantization settings, pruning ratio...

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)

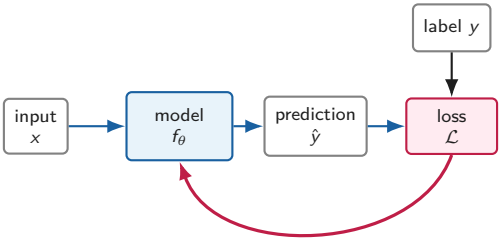
Iteration
one forward + backward + update on one batch

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



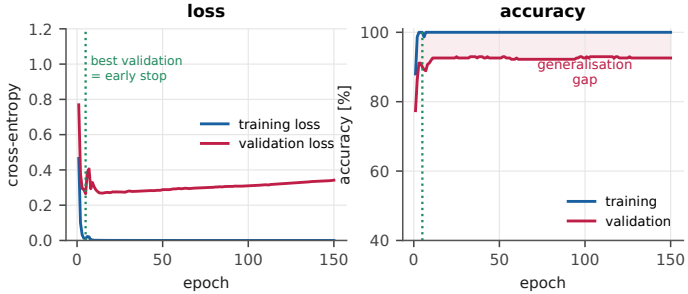

```
for x, y in loader:
    y_hat = model(x) # forward
    loss = criterion(y_hat, y)
    loss.backward() # gradients
    optimizer.step() # nudge theta
    optimizer.zero_grad()
```

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



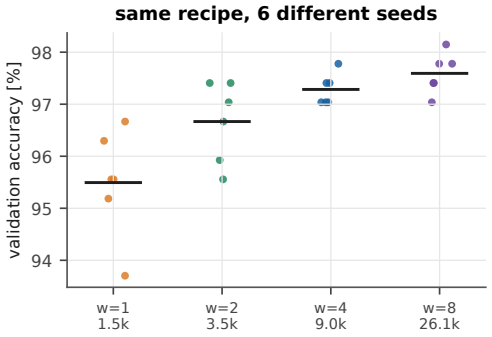
A $64 \to 512 \to 512 \to 10$ network trained on only 150 digits. Training loss goes to zero; training accuracy reaches 100%.

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.

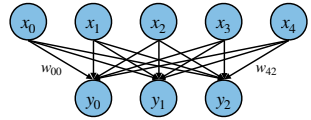
PART 2

What a neural network computes

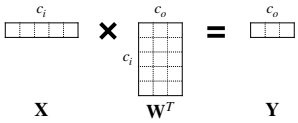
Layers, parameters, feature maps – and the single operation underneath all of them.

2

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*.

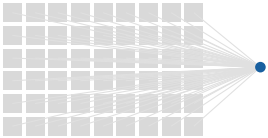
Matrix–matrix (GEMM)
MACs $= m \cdot n \cdot k$ convolution and attention both lower to GEMM.

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

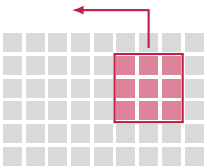
$150\,528 \times 1\,000$

$\approx$ **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

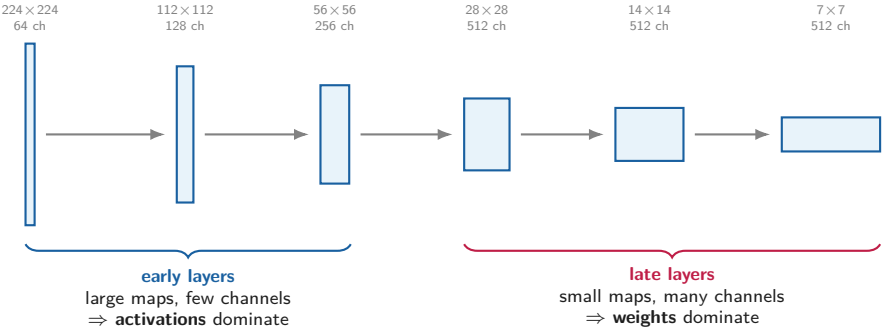
weights – for 64 feature detectors

$$Y_{o,h,w} = \sum_{c=1}^{C_{in}} \sum_{u,v=1}^{K} W_{o,c,u,v}\, X_{c,\, h+u,\, w+v} + b_o$$

$$\text{MACs} = H_o W_o\, C_{in} C_{out} K^2$$

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.

PART 3

How models got so expensive

The one curve that refused to follow the others.

3

High accuracy comes at the cost of high computation

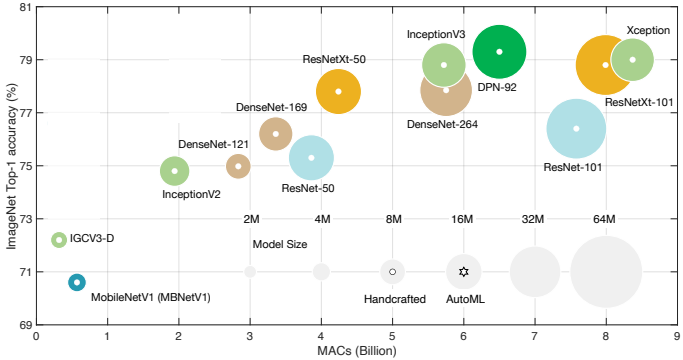


Figure: MIT HAN Lab 6.5940, Lecture 01; data from Deng et al., Proc. IEEE 2020.

For embedded AI, the interesting point is the Pareto frontier, not the single most accurate model.

Two exponentials, running at different speeds

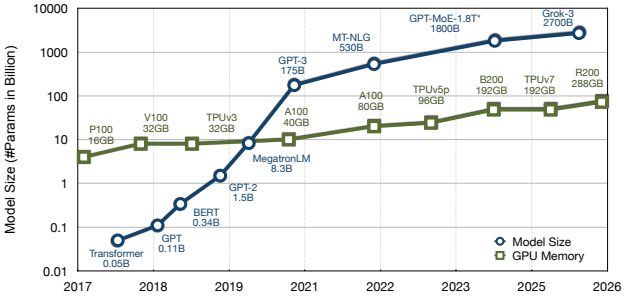


Figure: MIT HAN Lab 6.5940, Lecture 02.

What models ask for

parameters: $\sim 4\times$ every 2 years (0.05 B $\rightarrow$ $>$1 000 B)

What hardware delivers

GPU memory: $\sim 2\times$ every 2 years (16 GB $\rightarrow$ 288 GB)

Same vertical scale, very different slopes: demand leaves supply behind.

Hardware scaling alone cannot keep up: we must reduce what models demand.

THE THREAD – FIRST PASS

How much useful intelligence can we obtain from a limited budget of computation, memory and energy?

Not by scaling. Supply and demand diverge, so "buy a bigger machine" has stopped being a strategy. What remains is the only other lever: spend the budget *better*.

Which raises an awkward question: we have spent this whole story optimising a single number. Is accuracy even the right thing to be spending on?

PART 4

Measuring: accuracy is one axis of many

4

The nine numbers, where memory goes, and what the usual bottleneck is. After this part, you will never again read “our model is 2% better” the same way.

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?

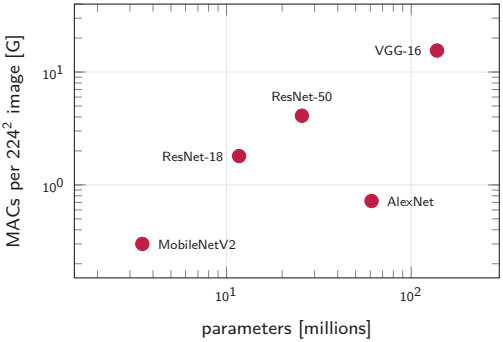
Throughput
How many inputs per second, in steady state?

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.

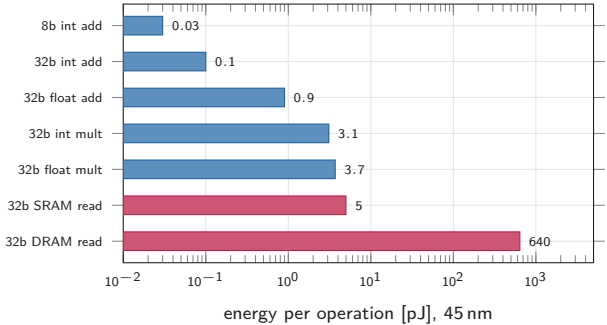
ResNet-50 vs VGG-16
5× fewer parameters *and* 4× less arithmetic – while being more accurate.

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**.

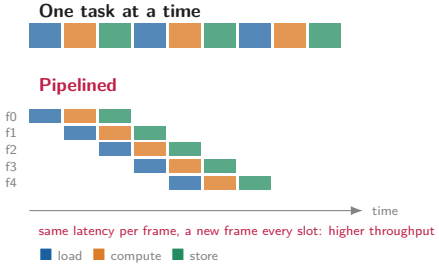
That is a factor of roughly **20 000**. The arithmetic is essentially free; *the commute is the job*.

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



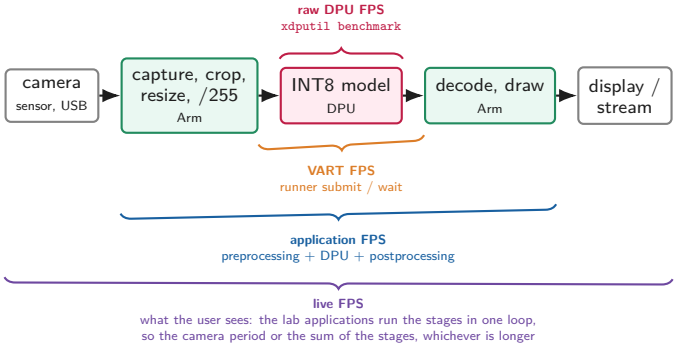
Latency matters for a control loop: a camera, a trigger, a brake.

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.

PART 5

Edge and frugal AI

Why moving the computation to the sensor changes which of the nine numbers you are allowed to care about – and the toolbox that responds: quantization, pruning, distillation, NAS.

5

Frugal AI: performance that meets the need, at the lowest resource cost.

Notice what the definition does *not* say. It does not say "small". It does not say "fast". It does not set any absolute threshold.

Frugality is defined **relative to a stated requirement**. Without a requirement, the word is meaningless.

So the first frugal act is not compression. It is **writing down what "good enough" means** – ≥ 25 FPS, 30 dB PSNR, under 5 W – and then refusing to pay for more than that.

A model that beats its target by 8 dB has not won. It has overspent.

After the *Frugal AI Guide*, ENS Challenge Data (2025), and the French *Référentiel général pour l'IA frugale* (AFNOR).

Every lab in this course begins by fixing a target and ends by asking what it cost to reach it.
Measure, understand, reduce.

Four levels to act on, in order of leverage

01 Frame the problem
Do you need a neural network at all? A smaller task, a coarser output, a lower frame rate, a classical baseline?

02 Design the model
Architecture, width, depth, input resolution, distillation. This is where most of the honest gains live.

03 Change the numbers
Quantization, pruning, mixed precision. Cheap to apply, bounded in what it can give.

04 Map to hardware
Compilation, operator fusion, tiling, dataflow, memory layout. The model does not change at all.

The course is taught bottom-up, from 04 to 01, because that is the order in which the ideas become teachable. Real projects should be read top-down.

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

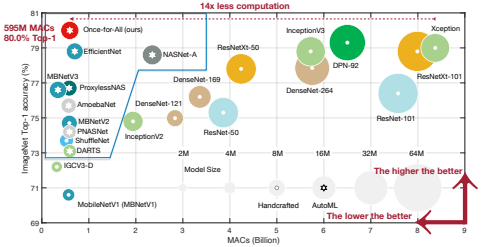
edits the *precision*

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.

Three ways frugality goes wrong – and how to measure honestly

The rebound effect

Make inference ten times cheaper and usage may grow a hundredfold. Efficiency gains can be erased – or reversed – at system level.

Optimising an idle machine

Compression only pays where the model is the bottleneck. Pruning a network that already fits and runs in time is wasted work.

The cost of optimising

Every sweep, search and retraining burns energy too. A 5% gain that costs 100 GPU-hours must be amortised.

increasing fidelity, increasing effort

Estimate it
count parameters and MACs from the graph (`torchinfo`, Green Algorithms)

Instrument it
read energy counters during a run (`CodeCarbon`, RAPL, board power monitors)

Put a probe on it
current-sense the power rail of the board: the only number that settles an argument

Estimate early to reject designs. Measure late to make claims. Never swap the two.

THE THREAD – SECOND PASS

How much useful intelligence can we obtain from a limited budget of computation, memory and energy?

The budget has **three separate currencies** – operations, bytes moved, and time – and they do not convert into one another at a fixed rate.

So far the answer has been "spend less". The rest of the lecture is about the hardware that decides *how much* each currency costs – and how to build that hardware ourselves.

PART 6

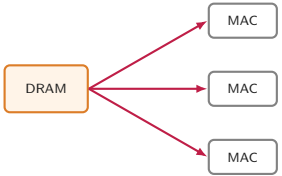
Accelerators: from CPUs to silicon

Why general-purpose processors lose, and when it is worth making a chip.

6

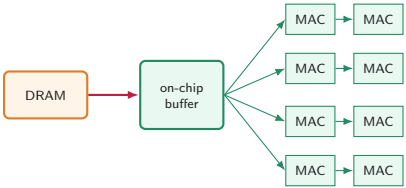
The real trick is reuse, not speed

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

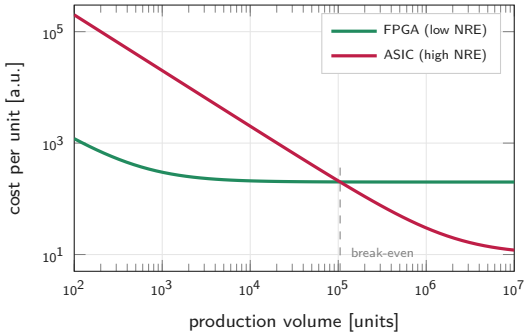


The same weight feeds a row of multipliers; the same activation a column. One fetch, dozens of MACs.

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.

ASIC, FPGA or GPU? Volume, flexibility and time



Illustrative numbers only (NRE = non-recurring engineering: masks, design, verification).

GPU
mature software, flexible, power-hungry

FPGA
reconfigurable; custom precision and I/O; deterministic latency; lower clock

ASIC
best TOPS/W and unit cost at volume; months to years to design; fixed once taped out

FPGAs also prototype ASICs and follow fast-moving models.

Choose an ASIC when the workload is stable and the volume is high; choose an FPGA when either is uncertain.

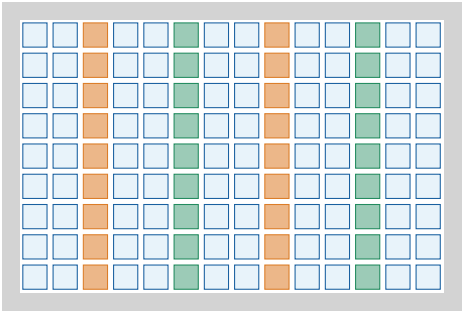
PART 7

FPGAs: building the datapath

7

What is inside an FPGA, the board of the labs, and the accelerator you will use first: the DPU.

What is inside an FPGA?



CLB DSP BRAM/URAM I/O

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)

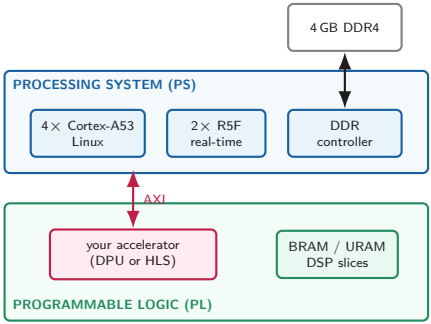
BRAM / URAM
36 Kb / 288 Kb on-chip blocks: tiles, FIFOs, line buffers

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*.

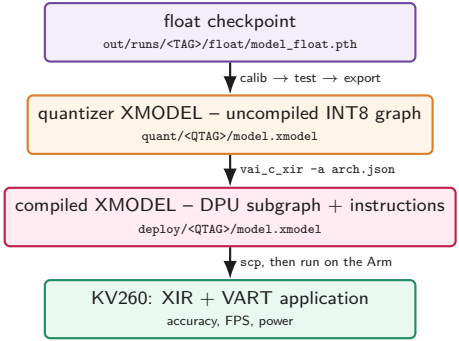
Notice the shape: a layer with 8 channels wastes half of the channel parallelism; one with 24 wastes a quarter (24 of 32 slots in two passes).

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

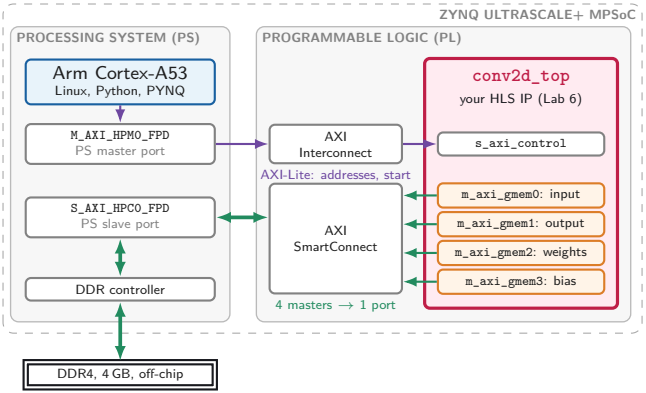
one batch-1 DPU computation partition; only the topology-checked terminal output `fix2float` conversion is allowed on CPU.

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`.

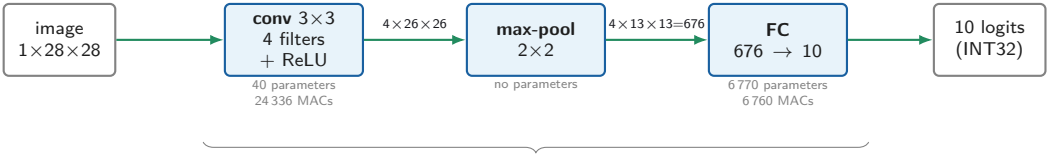
Control path: AXI-Lite
Python (PYNQ) allocates DDR buffers, writes their addresses into the IP's registers and starts it.

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



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.

PART 8

Outlook

What every lab report answers, and where the rest of the course goes.

8

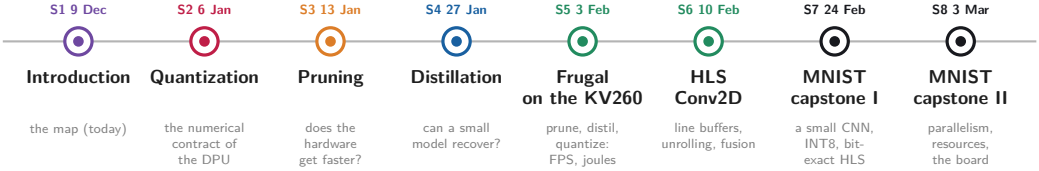
What every lab report has to answer

What was the target?	Stated *before* the experiment: frame rate, quality floor, power ceiling.
What did it cost?	Parameters, MACs, peak activation memory, model size on disk.
What did it actually do?	Measured latency, throughput and resource utilisation, end-to-end.
What did it burn?	Joules per inference, measured on the board – not inferred from FLOPs.
What did you give up?	Accuracy / PSNR against the float baseline – reported even when it is embarrassing.

Every number comes from a tagged run folder, every comparison uses the same float source and the same evaluation bundle, and every board number names its tool: VART application, `xdputil`, `vaitrace` or `platformstats`.

Five questions, every lab, one habit: never report a gain without reporting what paid for it.

Where the rest of the course goes



Exam: Wednesday 10 March 2027, 08:30–10:30, room 3B01.

We start by making the *model* fit the machine and end by making the *machine* fit the model. Every lab ends with numbers measured on a real board.

THE THREAD, ONE LAST TIME

How much useful intelligence can we obtain from a limited budget of computation, memory and energy?

You now have the vocabulary to answer it: **nine numbers**, **four levels of intervention**, and **one stack** running from a Python class down to a memory controller.

You get more intelligence per joule by making the model fit the machine: fewer operations, in a shape the array can use, at a precision the arithmetic can afford, arranged so that data crosses the chip boundary as rarely as possible.

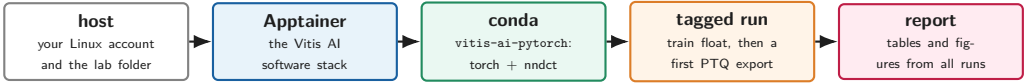
PART 9

Now: Lab 1

The workflow every later lab reuses: a container, a conda environment, tagged runs – and a first frugal comparison.

9

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

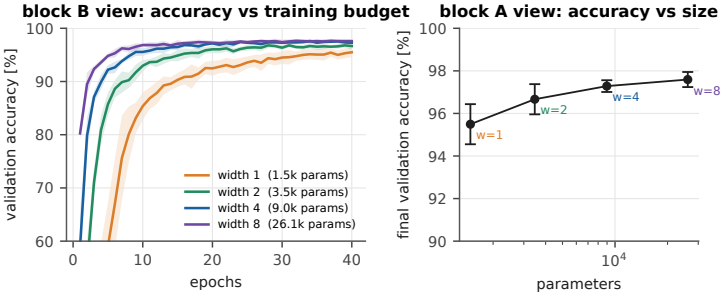
Labs 2–5 add quantized, compiled and board results under the same tag.

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.

What your Lab 1 report will show – and how to read it



Block B (one architecture, several epoch budgets): small models need more epochs; the widest one saturates after about 10. Where does *your* validation curve flatten – and does it start to overfit?

Block A (several sizes, same budget): each doubling of width buys less. Here width 8 costs 2.9× the parameters of width 4 for +0.3 points. Is the larger model justified?

A preview on the scikit-learn `digits` toy dataset (MLP only, mean ± std over 6 seeds). Your job: check whether FashionMNIST, and a CNN, behave the same.

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.

Sources and further reading

Courses and guides

- MIT HAN Lab, 6.5940 *TinyML and Efficient Deep Learning Computing*: https://hanlab.mit.edu/courses/2026-fall-65940
- ENS Challenge Data, *Frugal AI Guide* (2025): https://challengedata.ens.fr/static/Guide_IA_Frugale_EN.pdf
- V. Sze et al., *Efficient Processing of Deep Neural Networks*, Proc. IEEE 2017.

Accelerators

- H.T. Kung, *Why Systolic Architectures?*, IEEE Computer 1982.
- N. Jouppi et al., *In-Datacenter Performance Analysis of a TPU*, ISCA 2017.
- Y.-H. Chen, J. Emer, V. Sze, *Eyeriss*, ISCA 2016; S. Han et al., *EIE*, ISCA 2016.
- M. Horowitz, *Computing's Energy Problem*, ISSCC 2014.
- S. Williams et al., *Roofline*, CACM 2009.

FPGAs

- C. Zhang et al., *Optimizing FPGA-based Accelerator Design for Deep CNNs*, FPGA 2015.
- Y. Umuroglu et al., *FINN*, FPGA 2017; J. Duarte et al., *hls4ml*, JINST 2018.
- AMD: Vitis AI (UG1414), DPU (PG338), Vitis HLS (UG1399), Kria K26 (DS987); Arm AMBA AXI4 (IHI 0022).

Case studies

- A. Mazouz et al., *Lightweight FPGA Deployment of Learned Image Compression with Knowledge Distillation and Hybrid Quantization*, arXiv:2503.04832, 2025.
- A. Mazouz, C. P. Bridges, *Automated CNN Back-Propagation Pipeline Generation for FPGA Online Training*, J. Real-Time Image Processing, 2021.
- G. Giuffrida et al., *CloudScout*, IEEE TGRS 2020.

FOR READING

Supervised learning, in more detail

Examples, losses, gradient descent, capacity and overfitting.

The same machine, behind very different things

See
classify, detect, segment, track

Hear
transcribe, denoise, wake-word

Read & write
translate, summarise, answer

Generate
images, audio, video, code

Compress
learned codecs for image and video

Control
robots, drones, vehicles

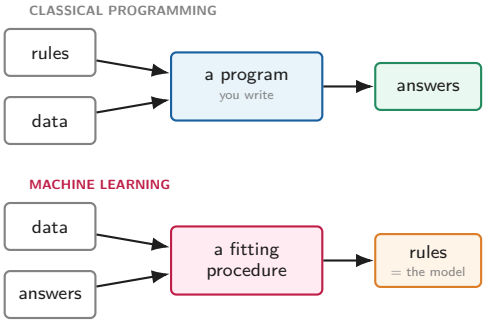
Predict
forecasting, anomaly detection

Decide
triage what is worth keeping or sending

Underneath all eight: **a stack of tensor operations whose numbers were fitted to data**. The differences live in the data and in the shape of the stack – not in the kind of machine.

The last two – compress and decide – follow us all hour. Both are useless unless they run *where the data is born*.

Machine learning inverts the direction of programming



Those "rules" are just **numbers**. A vision model holds tens of millions of them; a language model, hundreds of billions. Nobody chose them individually – they are the *output* of an optimisation.

This is why efficiency is hard: **you cannot read a model and see which parts matter.** You have to measure.

A trained model is a large table of numbers plus a fixed recipe for using them. Every technique in this course edits one or the other.

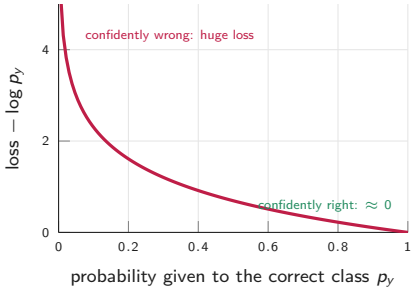
The datasets and models you will meet

Lab	Data	Input tensor	INT8 bytes	Model
1	FashionMNIST, 10 classes	$1 \times 28 \times 28$	784	MLP or small CNN, width $\times 1, \times 2$
2–3	CIFAR-10, 10 classes	$3 \times 32 \times 32$	3 072	small two-convolution CNN; VGG6-BN (Lab 3)
4	MNIST, 10 classes	$1 \times 28 \times 28$	784	teacher CNN $\rightarrow$ LeNet student
5	rock–paper–scissors, 3 classes	$3 \times 128 \times 128$	49 152	five Conv–BN–ReLU stages
5	a camera frame	$3 \times 480 \times 640$	921 600	cropped and resized before the DPU
6	one Conv2D layer	$3 \times 128 \times 128 \rightarrow 64 \times 126 \times 126$	49 152 $\rightarrow$ 1 016 064	Conv2D + ReLU + MaxPool in HLS
7	MNIST, 10 classes	$1 \times 28 \times 28$	784	compact Conv3$\times$3 (1$\rightarrow$4), pool, FC676$\rightarrow$10

From MNIST to one camera frame, the input grows by a factor of about **1 200**. In Lab 6 a single layer turns a 48 KB input into a 1 MB output: the activations, not the 1 728 weights, decide where the data can live.

Write down the tensor shape first: every cost – MACs, bytes, cycles, joules – follows from it.

Loss functions: what "wrong" means



Classification: cross-entropy
$p = \mathrm{softmax}(z)$, $\mathcal{L} = -\log p_y$ – the network outputs scores z (logits), softmax turns them into probabilities.

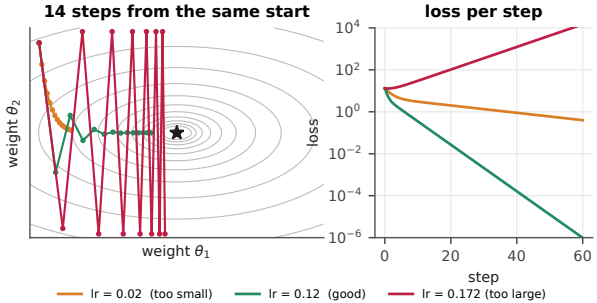
Regression: mean squared error
$\mathcal{L} = \frac{1}{N}\sum_i (\hat{y}_i - y_i)^2$ – depth, pose, a pixel value.

Compression: rate + distortion
$\mathcal{L} = R + \lambda D$ – bits spent vs quality lost; λ picks the trade-off.

Distillation (L04)
match a teacher's probabilities instead of (or as well as) the label.

The loss is the only thing the optimiser understands. If it is not in the loss, the network will not learn it.

Gradient descent, and why the learning rate matters



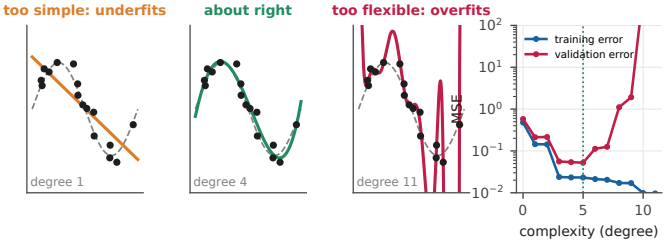
$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}$$

η is the learning rate. The gradient is estimated on a mini-batch: **stochastic** gradient descent.

too small: correct, but slow. **well chosen**: fast and stable. **too large**: overshoots along the steep direction and diverges. Adam adapts the step per parameter and is the usual default in the labs.

Gradient descent on a quadratic loss with one steep and one flat direction (computed for this lecture).

Underfitting, overfitting and model capacity



Sixteen noisy samples of a sine (dashed), fitted with polynomials of growing degree. The training error keeps falling; the **validation error** is U-shaped.

The goal is not to fit the training data – it is to **generalise**. For frugal AI there is a bonus: the best model is often not the biggest one.

Capacity must match the data. Too little: underfit. Too much: memorise the noise – and pay for it in parameters.

Fighting overfitting, and measuring more than accuracy

More data
the best regulariser, when you can get it

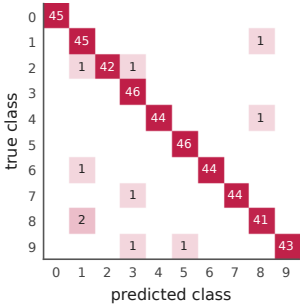
Augmentation
flips, crops, noise: new examples for free

Weight decay
penalise large weights: $\mathcal{L} + \lambda\|\theta\|^2$

Dropout
randomly silence neurons during training

Early stopping
keep the best validation epoch

Smaller model
less capacity – and cheaper to deploy



Confusion matrix, 450 test digits, 97.8% accuracy.
precision $= \frac{TP}{TP+FP}$ recall $= \frac{TP}{TP+FN}$

A single accuracy hides which classes fail. After compression, check *which* ones broke – not just how many.

FOR READING

PyTorch essentials for the labs

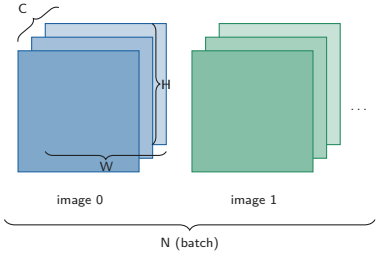
Tensors, autograd, modules, data loaders, the training loop, and the handover to deployment.

Tensors: shape, data type, device

```python
import torch
x = torch.randn(8, 3, 224, 224) # N, C, H, W
x.shape # torch.Size([8, 3, 224, 224])
x.dtype # torch.float32 (4 bytes/value)
x.numel() * x.element_size() / 1e6 # 4.8 MB

x = x.to("cuda") # move to the GPU (if present)
y = x.half() # float16: half the memory
q = torch.quantize_per_tensor(x.cpu(), scale=0.02,
       zero_point=0, dtype=torch.qint8) # 1 byte/value

x[0].mean(), x.permute(0, 2, 3, 1).shape # NHWC view
```



dtype	bytes	used for
float32	4	training, reference
float16 / bf16	2	GPUs
int8	1	edge inference, DPU

Shape $\times$ bytes per element = memory. PyTorch is NCHW; many accelerators (the DPU included) are NHWC.

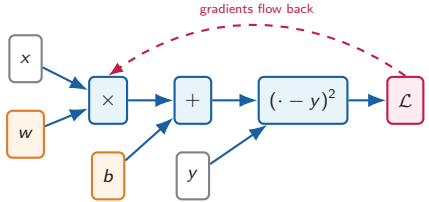
Autograd: the graph is recorded while you compute

```
w = torch.tensor(2.0, requires_grad=True)
b = torch.tensor(0.5, requires_grad=True)
x, y = torch.tensor(3.0), torch.tensor(4.0)

y_hat = w * x + b # forward: graph built
loss = (y_hat - y) ** 2 # = (6.5 - 4)^2 = 6.25
loss.backward() # backward: chain rule

w.grad # dL/dw = 2(y_hat - y) * x = 15.0
b.grad # dL/db = 2(y_hat - y) = 5.0

with torch.no_grad(): # the update itself must
    w -= 0.01 * w.grad # not enter the graph
```



Every operation on a tensor with `requires_grad` is recorded. `backward()` walks the graph in reverse and fills `.grad`.

To do that, the **forward activations must be kept** until the backward pass – the reason training needs far more memory than inference.

You never write derivatives by hand. You do pay for them in memory – which is why training on the edge is hard.

nn.Module: defining a model

```python
import torch.nn as nn

class SmallCNN(nn.Module):
    def __init__(self, n_classes=10):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(1, 16, 3, padding=1), # 16x28x28
            nn.BatchNorm2d(16), nn.ReLU(),
            nn.MaxPool2d(2), # 16x14x14
            nn.Conv2d(16, 32, 3, padding=1), # 32x14x14
            nn.BatchNorm2d(32), nn.ReLU(),
            nn.MaxPool2d(2)) # 32x7x7
        self.classifier = nn.Linear(32*7*7, n_classes)

    def forward(self, x): # x: Nx1x28x28
        x = self.features(x)
        return self.classifier(x.flatten(1)) # logits

model = SmallCNN()
sum(p.numel() for p in model.parameters()) # 20,586
```

`__init__`
declares the layers – this is where the *parameters* live.

`forward`
says how data flows – this is where the *activations* are produced.

Conv–BN–ReLU
the pattern every compiler fuses into one operation at deployment.

Write shapes as comments
you will need them for MACs, memory, and the accelerator's constraints.

Simple, static, standard layers: that is what quantizers and compilers handle.

Data: Dataset, transforms, DataLoader

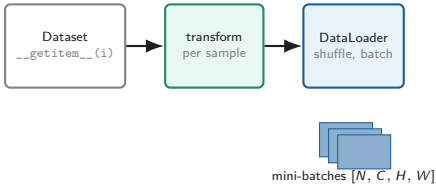
```python
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, Subset

tf_train = transforms.Compose([transforms.RandomHorizontalFlip(),
                               transforms.ToTensor()])
# uint8 -> [0,1]
tf_eval = transforms.ToTensor() # no augmentation

root = "data"
train_all = datasets.FashionMNIST(root, train=True, download=True,
                                  transform=tf_train)
eval_all = datasets.FashionMNIST(root, train=True, transform=tf_eval)
test_set = datasets.FashionMNIST(root, train=False, transform=tf_eval)

g = torch.Generator().manual_seed(0) # reproducible split
idx = torch.randperm(60000, generator=g)
train_set = Subset(train_all, idx[:54000]) # augmented
val_set = Subset(eval_all, idx[54000:]) # NOT augmented

train_loader = DataLoader(train_set, batch_size=64, shuffle=True)
val_loader = DataLoader(val_set, batch_size=256)
x, y = next(iter(train_loader)) # x: [64,1,28,28] y: [64]
```



Augment training data only

splitting one augmented dataset would silently augment the validation set too.

Preprocessing is part of the model

the board application must repeat it exactly (/255, layout, crop) – or accuracy silently collapses.

The training and evaluation loop

```python
device = "cuda" if torch.cuda.is_available() else "cpu"
model = SmallCNN().to(device)
opt = torch.optim.Adam(model.parameters(), lr=1e-3)
loss_fn = nn.CrossEntropyLoss()
def train_one_epoch():
    model.train() # BN / dropout: training mode
    for x, y in train_loader:
        x, y = x.to(device), y.to(device)
        loss = loss_fn(model(x), y) # forward
        opt.zero_grad() # clear old gradients
        loss.backward() # backward
        opt.step() # update
@torch.no_grad() # no graph: less memory
def evaluate(loader):
    model.eval() # BN uses running stats
    hits = 0
    for x, y in loader:
        pred = model(x.to(device)).argmax(1)
        hits += (pred == y.to(device)).sum().item()
    return hits / len(loader.dataset)

for epoch in range(20): # + keep the best epoch
    train_one_epoch(); acc = evaluate(val_loader)
```

The classic bugs

Forgot `model.eval()`
BatchNorm and dropout keep behaving as in training: noisy, wrong accuracy.

Forgot `zero_grad()`
gradients accumulate across batches.

Tensors on two devices
model on GPU, data on CPU: a runtime error.

Evaluating with gradients
no `no_grad()`: memory fills with a graph you never use.

Inspect, save, export: the handover to deployment

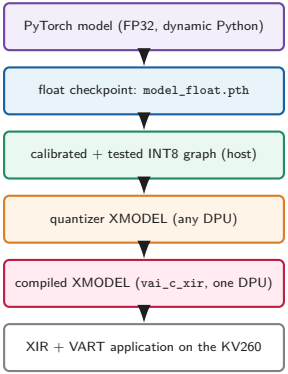
```python
torch.save(model.state_dict(), # the float source
           f"out/runs/{TAG}/float/model_float.pth")

from pytorch_nndct.apis import torch_quantizer
dummy = torch.randn(1, 1, 28, 28) # batch 1, fixed shape

# 1. calibrate: observe activation ranges (labels not needed)
q = torch_quantizer("calib", model, (dummy,), output_dir=out)
for x, _ in calib_loader: # a few hundred images
    q.quant_model(x)
q.export_quant_config()

# 2. test: evaluate the INT8 model on the host
q = torch_quantizer("test", model, (dummy,), output_dir=out)
evaluate(q.quant_model, val_loader)

# 3. export: a separate test pass that sees one batch-1 input
q = torch_quantizer("test", model, (dummy,), output_dir=out)
q.quant_model(dummy) # one forward pass
q.export_xmodel(output_dir=out) # quantizer XMODEL
```



Each arrow removes freedom: dynamic Python becomes a fixed graph, floats become integers, operators become DPU instructions. **Anything that does not survive an arrow must be changed in the model.**

Lab 1 ends at the quantizer XMODEL; from Lab 2 on, you compile and run it.

FOR READING

What a network computes, in more detail

Feature maps, the loop nest the hardware sees, depthwise convolutions, attention, training versus inference.

From neurons to deep networks

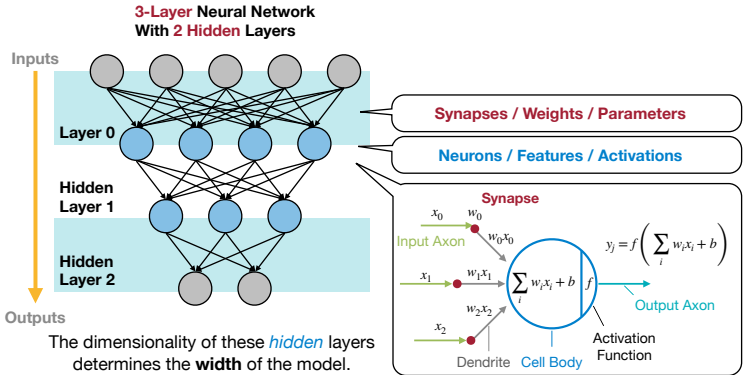
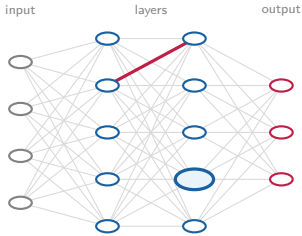


Figure: MIT HAN Lab 6.5940, Lecture 02, https://hanlab.mit.edu/courses/2026-fall-65940

A neuron computes a weighted sum and a nonlinearity:
$y = f\left(\sum_i w_i x_i + b\right)$**. Everything else is organisation.**

Inside the model: parameters versus feature maps



A line is a parameter. One learned number. Count them all and you get *model size* – what you download, store and quantize.

A node holds an activation. It exists only while *this* input is being processed. Stacked per layer, these are the *feature maps*.

Parameters are storage. Feature maps are traffic. Keep the two apart – most deployment surprises come from confusing them.

Model size is what you download once. Feature maps are what you move again and again – for every single input.

What the hardware sees: seven nested loops

```
for (n = 0; n < N; n++) // batch
 for (m = 0; m < C_out; m++) // output channel
  for (y = 0; y < H_out; y++) // output row
   for (x = 0; x < W_out; x++) // output column
    for (c = 0; c < C_in; c++) // input channel
     for (i = 0; i < K; i++) // kernel row
      for (j = 0; j < K; j++) // kernel column
        O[n][m][y][x] += W[m][c][i][j]
                       * I[n][c][y*S+i][x*S+j];
```

Each weight is reused $H_o W_o$ times; each input pixel up to $K^2 C_{out}$ times.

Loop order
which data stays in place?

Tiling
which block fits on chip?

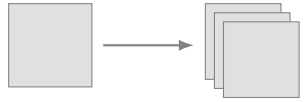
Unrolling
which loops become parallel hardware?

Pipelining
how often does a new iteration start?

The ML view sees features; the hardware view sees loops, buffers and reuse. This course constantly translates between the two.

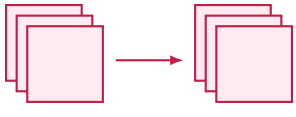
Depthwise-separable convolution: buy the same feature for less

Standard 3×3 convolution



$9 \cdot C^2 \cdot H \cdot W$

Depthwise + pointwise



$9 \cdot C \cdot H \cdot W + C^2 \cdot H \cdot W$

The convolution is factorised into "filter each channel" then "mix the channels". At $C = 256$ the ratio is $\frac{1}{9} + \frac{1}{256}$: about $9\times$ **less arithmetic** for the same output shape – but each byte is reused less, so the layer is often **memory-bound**.

Another assumption traded for a resource: spatial filtering and channel mixing need not happen in the same operation. This is the trick behind every mobile-class backbone.

Transformers: attention is also matrix multiplication

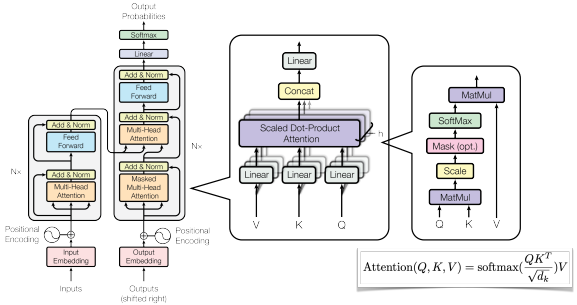


Figure: MIT HAN Lab 6.5940, Lecture 02; Vaswani et al., NeurIPS 2017.

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

$$Y = \text{softmax}\left(QK^\top/\sqrt{d}\right)V$$

- projections and MLP: GEMM;
- attention grows as n^2 with sequence length;
- softmax / LayerNorm need non-MAC hardware;
- global interaction: harder to stream than convolution.

CNNs map naturally to line-buffered streaming; transformers map to tiled GEMM plus careful memory management.

Training and inference are not the same problem

Training	Inference
happens once	happens forever
weeks on a GPU cluster	milliseconds, on the device
one run, amortised over the model's life	once per photo, per frame, per query
large batches – the hardware stays busy	batch of one – the hardware starves
forward *and* backward	forward only
every feature map kept for the backward pass	feature maps thrown away at once
FP32 / BF16	INT8, sometimes INT4
power is an invoice	power is heat, a battery, or a solar panel

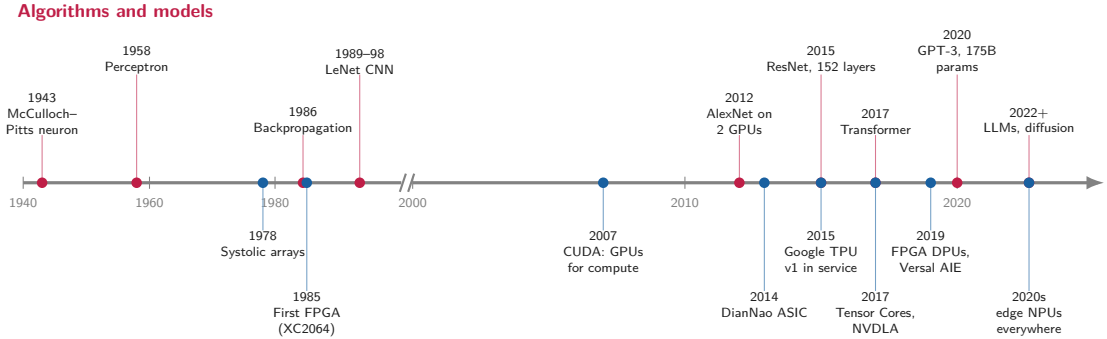
This course lives in the right-hand column. Whenever a paper claims "efficiency", ask first: efficiency of *which* column?

FOR READING

How models got so expensive

A short history of AI told through the curves that matter.

A short history: every AI leap rode on a hardware leap



Kung & Leiserson (1978); LeCun et al. (1989, 1998); Krizhevsky et al., NeurIPS 2012; Chen et al., ASPLOS 2014; Jouppi et al., ISCA 2017; Vaswani et al., NeurIPS 2017.

Neural networks existed for decades; they took off when data met parallel hardware.

Deep learning did not win on ideas alone

Data 2009

Labelled sets went from thousands of images to millions. ImageNet alone: 1.2 M training images, 1 000 classes.

Compute 2007–

The GPU turned a research idea into an overnight experiment. Backpropagation is, in the end, dense linear algebra.

Method 2010–

ReLU, better initialisation, dropout, batch normalisation, residual connections. Deep stacks finally became trainable.

None of the three is an algorithm alone. Two of them are **resources**. That is the quiet fact underneath the whole field – and the reason the resource question had to come back eventually.

A field that grew by spending resources will sooner or later have to learn to spend them well.

2012: the year the curve broke

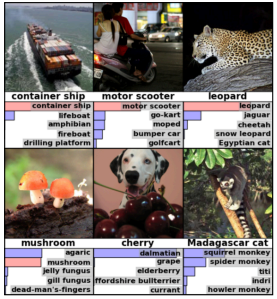


Image Classification

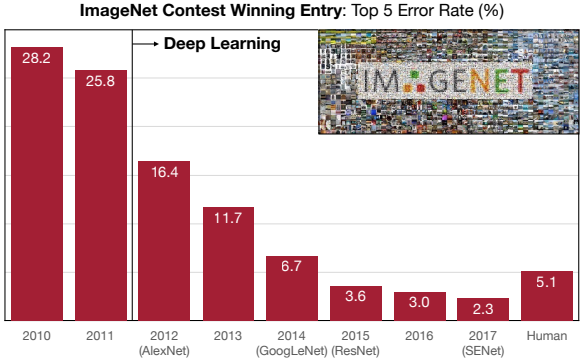
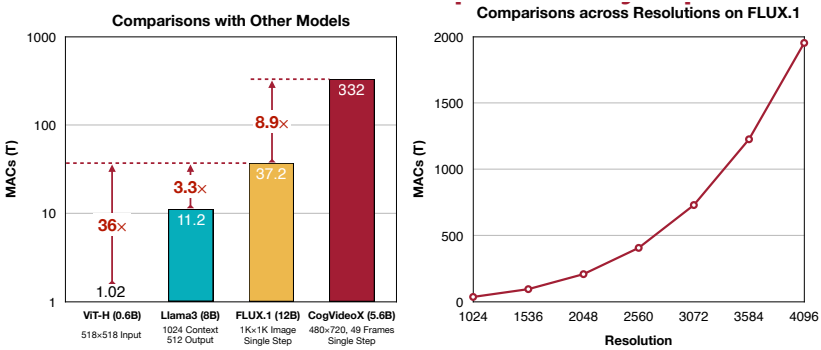


Figure: MIT HAN Lab 6.5940, Lecture 01; ImageNet/ILSVRC winning entries (Russakovsky et al., IJCV 2015).

A convolutional network trained on two consumer GPUs cut the error by a third in one year. The lesson the community drew: when in doubt, make it bigger.

Generative models make the gap sharper



Diffusion models require more computation than other models. The computation scales super-quadratically with resolution.

Figure: MIT HAN Lab 6.5940, Lecture 01.

Compute grows super-quadratically with resolution – a key lesson for every image and video workload on the edge.

The bill arrives

200–400

TWh

projected electricity for AI in data centres by **2030**

France's total consumption in 2024: 449 TWh

20 400

t CO_2e

to train one 123B-parameter model over 18 months

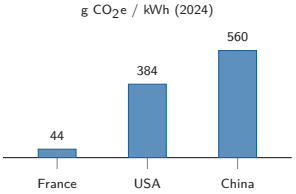
$\approx$ 2 300 French residents for a year

0.42–1.79

Wh

per request to one large deployed model

times $\approx$ one billion requests a day



The same computation emits up to $\sim 10\times$ more depending on the grid. Figures are contested; read them as **orders of magnitude** – exactly what one engineering decision can move.

ENS Challenge Data, *Frugal AI Guide* (2025), compiling IEA, EDF, Mistral AI and peer-reviewed inference studies.

FOR READING

Measuring, in more detail

Memory, weights versus activations, bottlenecks, and how to time a model.

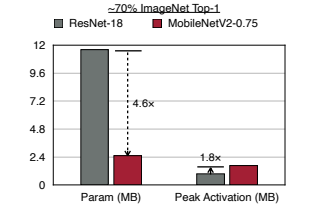
"Memory" means four different things

Weight memory
The parameters, at rest. Fixed for the model's life.
fits, or does not

Peak activation
The largest intermediate tensor alive at once.
the usual show-stopper

Memory traffic
Bytes moved between chip and DRAM per inference.
where the energy goes

Bandwidth
Bytes per second the link can sustain: a hard ceiling.
where the time goes



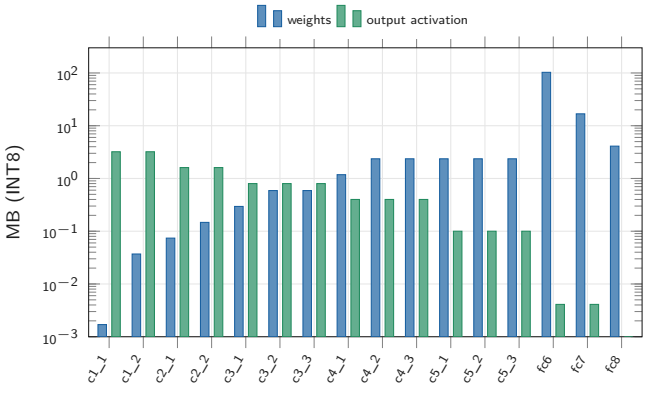
*** All parameters and activations are Integer numbers (8 bits).**

MobileNetV2 has 4.6× fewer parameters than ResNet-18 – but a 1.8× *larger* peak activation.

One early feature map: $64 \times 224 \times 224 = 3.2\,\text{MB}$ in INT8 – about the *entire* on-chip SRAM of a mid-range FPGA SoC.

Figure: MIT HAN Lab 6.5940, Lecture 02; MCUNet (Lin et al., NeurIPS 2020).

Weights or activations? It depends on the layer



VGG-16, 224×224 input, INT8, per layer.

Early convolutions
3.2 MB feature maps, KB of weights: **activations** decide what fits.

Late convolutions
2.4 MB of weights, 0.1 MB maps: **weights** dominate.

Fully-connected
103 MB of weights, each used once per image: pure **bandwidth**.

Computed from the VGG-16 layer shapes (Simonyan & Zisserman, ICLR 2015). Total: 138 MB of weights, 13.6 MB of activations.

There is no single "memory footprint". An accelerator must handle both regimes – or the model must be designed to avoid one.

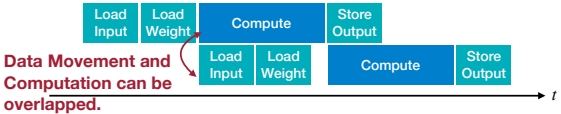
So what is the usual bottleneck?

Layer type	Ops / byte	Usually limited by	What helps
Early conv (large maps)	high	on-chip memory for activations	line buffers, spatial tiling, streaming, fusion
Late conv (many channels)	high	weight storage / reload	channel tiling, weight-stationary buffers, pruning
Depthwise / pointwise	low	DRAM bandwidth, PE utilisation	fusing dw+pw, keeping maps on chip
FC / attention, batch 1	≈2	DRAM bandwidth for weights	quantization, pruning, batching, weight compression
Unsupported op	–	CPU fallback + two DRAM round trips	replace the op or build a custom core

On an edge FPGA the usual answer is **data movement**: DDR bandwidth shared with the CPU, and on-chip SRAM measured in MB. DSPs usually run out only *after* the memory system is fixed.

Profile per layer, not per network: the bottleneck moves as you go deeper.

Latency is set by the slower of compute and data movement



$$Latency \approx \max\left(T_{computation}, T_{memory}\right)$$

$$T_{\text{computation}} \approx \frac{\text{Number of Operations in Neural Network Model}}{\text{Number of Operations that Processor can Process Per Second}}$$

NN Specification

Hardware Specification

$$T_{\text{memory}} \approx T_{\text{data movement of activations}} + T_{\text{data movement of weights}}$$

$$T_{\text{data movement of weights}} \approx \frac{\text{Neural Network Model Size}}{\text{Memory Bandwidth of Processor}}$$

NN Specification

Hardware Specification

$$T_{\text{data movement of activations}} \approx \frac{\text{Input Activation Size + Output Activation Size}}{\text{Memory Bandwidth of Processor}}$$

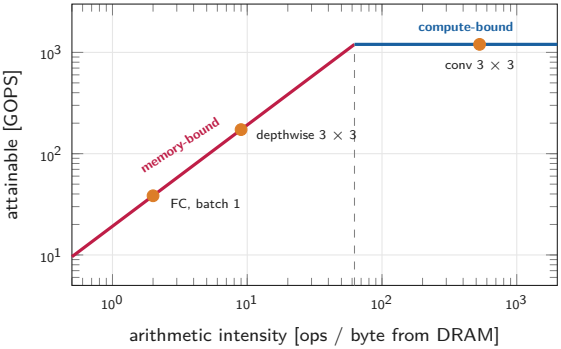
NN Specification

Hardware Specification

Figure: MIT HAN Lab 6.5940, Lecture 02.

Every latency term is a ratio: what the network needs divided by what the hardware provides.

Two ways to be slow – and they need opposite cures



$$P = \min\left(P_{peak},\ BW \times I\right)$$

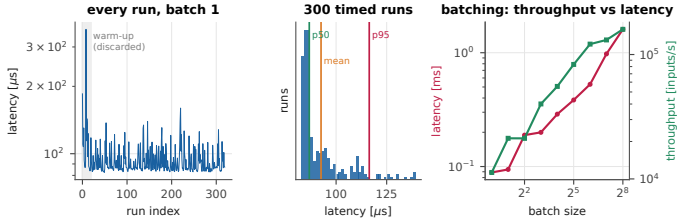
Illustrative edge accelerator: 1.2 INT8 TOPS, 19.2 GB/s DRAM $\Rightarrow$ ridge at $\approx$62 ops/byte.

- **left of the ridge**: fewer FLOPs do nothing – move fewer bytes;
- **right of the ridge**: fewer FLOPs help – until you cross back.

Williams, Waterman & Patterson, *Roofline*, CACM 2009. Operating points: back-of-the-envelope INT8 estimates.

Before optimising arithmetic, compute the arithmetic intensity: it tells you which wall you will hit.

Measuring latency properly




```
model.eval(); torch.set_num_threads(1)
x = torch.randn(1, 1, 28, 28) # batch 1
with torch.no_grad():
    for _ in range(20): model(x) # warm-up
    t = []
    for _ in range(100):
        t0 = time.perf_counter(); model(x)
        t.append(time.perf_counter() - t0)
p50, p95 = np.percentile(t, [50, 95])
```

- **warm-up**: the first call took 185 μs, the median 87 μs – caches, allocation, lazy initialisation;
- **tail**: p95 is 116 μs; a single run can land anywhere in the distribution;
- **batching**: 256 inputs at once gives 14$\times$ more inputs per second, but each answer waits 18$\times$ longer;
- **report**: p50, p95, mean, number of runs, threads, CPU model.

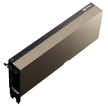
Measured for this lecture: a $784 \to 512 \to 512 \to 10$ float32 MLP on one CPU core (NumPy).

FOR READING

Edge AI and the toolbox, in more detail

Why the edge, the ladder of devices, space, and one picture per compression family (L02–L04 go further).

Edge devices have a huge gap to cloud processors







	Cloud AI	Mobile AI	Tiny AI
Memory (Activation)	80GB	4GB	320kB
Storage (Weights)	~TB/PB	256GB	1MB

Figure: MIT HAN Lab 6.5940, Lecture 01.

From cloud to microcontroller: $\sim$250 000$\times$ less activation memory. The same model cannot simply be copied down.

Six reasons the computation moves to the sensor

Latency
A round trip is tens to hundreds of ms, and jittery. A control loop cannot plan around that.

Bandwidth
Sensors now out-produce every link attached to them. Sending everything stopped being possible.

Privacy
Data that never leaves the device cannot leak from a server – increasingly a legal requirement.

Reliability
Tunnels, factories, orbit, open sea. The system must still work when the network does not.

Energy
Transmitting a megabyte often costs more than processing it locally. Radio is expensive.

Cost
No per-inference cloud bill, no egress charges – and hardware you buy once.

Only two of these six are about speed. The other four are about things a faster server cannot fix.

“The edge” is a ladder, not a place

	memory	power	
Microcontroller	256 KB SRAM	< 0.1 W	always on, INT8 at best, no OS
Embedded FPGA SoC	2–4 GB DDR	5–15 W	this course: Kria KV260, ZCU102
Mobile SoC	6–16 GB	2–8 W	NPU behind a vendor runtime
Edge GPU module	8–64 GB	15–60 W	a small data centre with a fan
Server GPU	80–288 GB HBM	400–1000 W	mains power, liquid cooling



Going down the ladder breaks things: floating point gets slow or disappears, dynamic shapes and virtual memory vanish, the operator set shrinks, retraining on the device becomes a research problem.

Four orders of magnitude of power separate the top and bottom rungs.
“Runs on the edge” means nothing until you say which rung.

Space: the edge at its most extreme

Φ-sat-1 ESA, launched Sept. 2020

A 6U CubeSat with a hyperspectral camera and a CNN (CloudScout) that decides *in orbit* whether an image is too cloudy to be worth sending home.

92%	**325 ms**	**∼2 W**
accuracy	per inference	on a Myriad 2 VPU

Giuffrida et al., IEEE TGRS 2020 (CloudScout); Giuffrida et al., Remote Sensing 2021 (Φ-sat-1).

Power
a solar panel and a battery, shared with everything else

Thermal
no air: heat leaves only by radiation

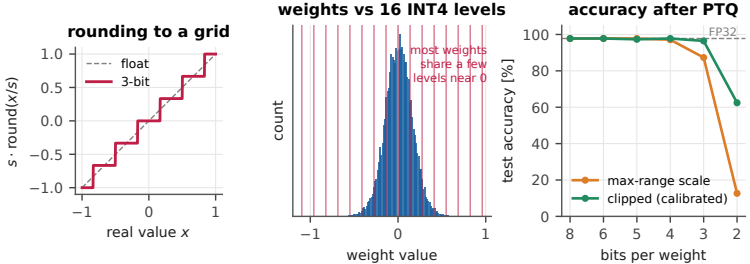
Radiation
bit flips in memory and logic; parts must be qualified

Downlink
a few minutes per ground-station pass – every byte counts

FPGAs add one more option: they can be **reconfigured in orbit** – new model, same hardware.

Every constraint in this course exists in a harsher form in orbit. If a technique survives there, it survives anywhere.

Quantization in one picture



Left: a real value is replaced by the nearest point on a grid of 2^b levels: $x \approx s \cdot q$, q an integer.

Middle: trained weights are bell-shaped. With 4 bits most of them share the few levels near zero; the outer levels are nearly empty.

Right: INT8 is free, INT4 costs almost nothing *if the range is calibrated*, INT2 breaks. Clipping outliers matters more than bits.

Computed for this lecture: 3-layer MLP on scikit-learn `digits`, per-tensor symmetric weight quantization, no retraining.

Quantization trades resolution for bytes; calibration and QAT decide where the grid goes.

PTQ or QAT: who adapts to the rounding?

FP32: 32 bits

INT8: 8 bits



Storage	$\div 4$
Memory traffic	$\div 4$
Multiplier cost	$\approx \div 16$

multiplier area and energy grow roughly with the square of the bit width

$$x \approx s\,(q - z), \qquad q \in \{-128, \ldots, 127\}$$

PTQ post-training

Calibrate on a few hundred images, pick scales, convert. No retraining. Minutes of work – sometimes a visible accuracy drop.

QAT quantization-aware

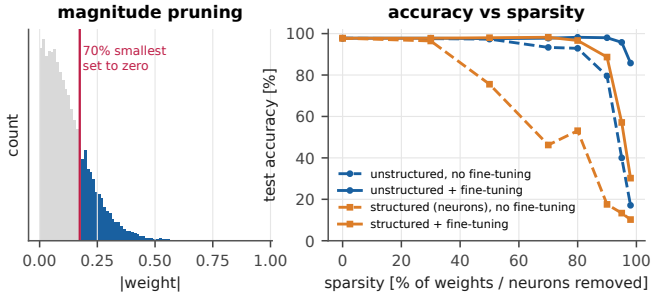
Simulate rounding inside the training loop so the network learns to tolerate it. Hours of work; usually recovers most of the loss.

Same bit width, same hardware – the difference is whether the network was allowed to *adapt* to the rounding.

Nagel et al., *A White Paper on Neural Network Quantization*, 2021; ENS Challenge Data, *Frugal AI Guide* (2025).

Quantization is not a conversion step at the end. Done properly it is part of training.

Pruning in one picture



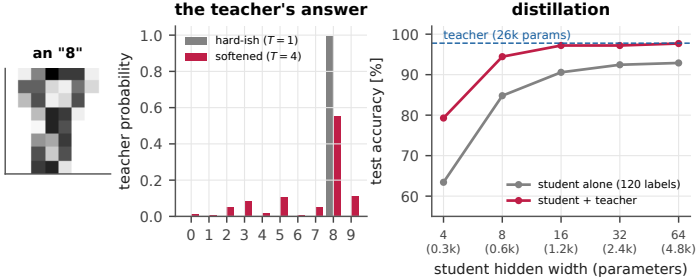
Unstructured (blue): zero the smallest individual weights. With a few epochs of fine-tuning, 90% of the weights can go at no loss – but the matrices keep their shape, so a dense accelerator is *not* faster.

Structured (orange): remove whole neurons or channels. It hurts more without fine-tuning, but with it 70% of the neurons go at no loss – and the layers really are smaller: fewer MACs on *any* hardware.

Computed for this lecture: 3-layer MLP on scikit-learn `digits`; magnitude (L1) criteria; 15 epochs of fine-tuning.

Fine-tuning makes pruning work; structure makes it pay on hardware.

Knowledge distillation in one picture



Middle: at $T = 1$ the teacher says "8" with 99.6% confidence. Softened with $T = 4$ it also says "a bit like 9, 5 or 3" – information a label cannot carry.

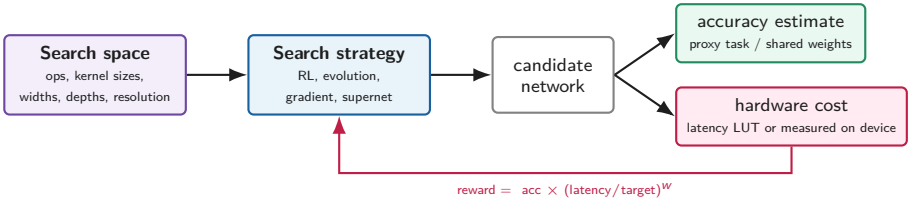
Right: small students trained on 120 labelled digits (grey) versus the same students taught by the teacher on the unlabelled training images (red). An 8-neuron student goes from 85% to 94%.

$$\mathcal{L} = w_{KD}\, T^2\, \mathrm{KL}\left(p_t^T \| p_s^T\right) + w_{CE}\, \mathcal{L}_{CE}$$

Computed for this lecture: teacher 64-128-128-10 MLP; students 64-w-10; $T = 4$, $w_{KD} = 0.9$, $w_{CE} = 0.1$; mean of 5 seeds.

A teacher can help a small student: check it against a fair baseline.

Neural architecture search: let the hardware vote



MnasNet 2019
latency measured on a real phone inside the reward

ProxylessNAS 2019
search directly on the target task and hardware

FBNet / DARTS 2019
differentiable search: GPU-days, not thousands

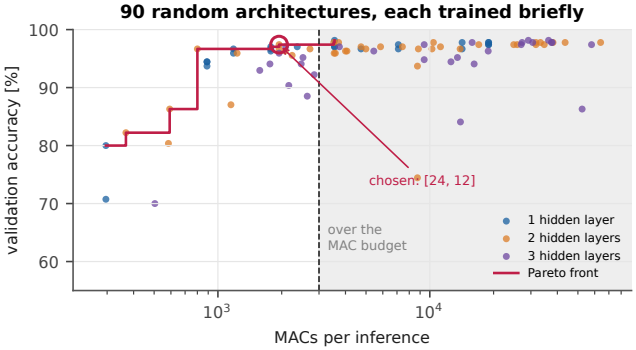
Once-for-All 2020
train one supernet, specialise sub-nets per device

For our FPGA, the **search space itself** should encode the accelerator's contract: supported operators only, channel counts in multiples of 16, static shapes.

Tan et al., CVPR 2019; Cai et al., ICLR 2019 and ICLR 2020; Wu et al., CVPR 2019; Liu et al., ICLR 2019.

NAS puts the hardware in the loop – and the search must pay for itself.

NAS in one picture: search, then pick under the budget



Each dot is one architecture: 1–3 hidden layers, widths from 4 to 256, trained for 25 epochs.

The **Pareto front** is the set of models no other model beats on both axes. Everything below it is wasted design.

With a 3 000-MAC budget, the search picks a $64 \to 24 \to 12 \to 10$ network: 97.4% at 1 944 MACs, versus 98.1% for the best model at 3 552 MACs – and far above many models $10\times$ larger.

Computed for this lecture: random search, scikit-learn `digits`, validation accuracy. Real NAS replaces random sampling with a smarter strategy and MACs with measured latency.

NAS turns "which architecture?" into a measured trade-off. The budget line comes from the hardware.

FOR READING

Accelerators, in more detail

From Python to a memory controller, systolic arrays, dataflows, tiling, a decade of AI chips.

The descent: from Python to a memory controller

YOU WRITE THIS

each step throws information away

Model	`class Net(nn.Module)` – Python objects and autograd; convenient, unexecutable by hardware
Operators	`conv2d`, `relu`, `add` – a finite vocabulary of tensor functions
Computation graph	operators plus the tensors between them; shapes fixed, execution order explicit
Quantized graph	every tensor gets a scale and a zero point; floating point disappears
Compiled instructions	fusion, scheduling, tiling, buffer allocation: a binary for one specific engine
Accelerator	MAC arrays, on-chip buffers, a memory controller – it executes instructions, not layers
Memory system	DRAM, bus, bandwidth: the floor everything rests on, and usually what limits you

THIS IS WHAT RUNS

Problems surface far below where you wrote the code, as a number you did not expect.

Why a CPU is the wrong shape for this

CPU

A few very clever cores. To perform one multiply-accumulate it fetches and decodes an instruction, checks dependencies, predicts a branch, and touches a cache hierarchy.

the overhead dwarfs the arithmetic

GPU

Thousands of simple lanes running the same instruction on different data. Far better – but still a general machine, still fetching instructions, still built for throughput over latency.

great in a rack, hungry at the edge

Accelerator

A fixed array of MAC units with operands routed directly between them. The dataflow *is* the hardware. One instruction can drive thousands of multiplications.

no flexibility, enormous efficiency

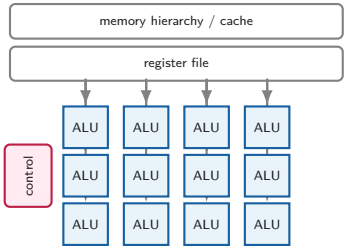
Instruction overhead on a CPU costs tens of pJ – against 0.03 pJ for the 8-bit add it wraps. The trade is always the same: **generality for efficiency**.

Horowitz, ISSCC 2014; Hameed et al., *Understanding Sources of Inefficiency in General-Purpose Chips*, ISCA 2010.

You are not buying more arithmetic. You are buying the removal of everything that was not arithmetic.

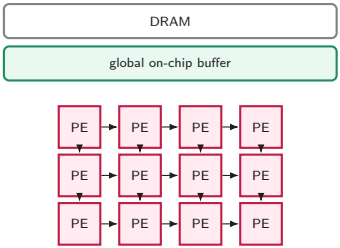
Two ways to organise parallel hardware

Temporal architecture (CPU SIMD, GPU)



centralised control; ALUs do not talk to each other – all data goes through the memory hierarchy.

Spatial architecture (TPU, NPU, FPGA)



processing elements with local registers pass data directly to neighbours: reuse happens inside the array.

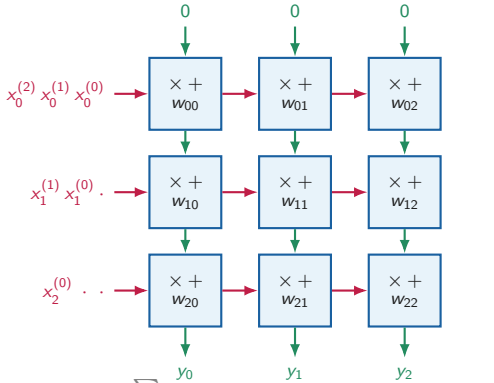
Sze, Chen, Yang & Emer, *Efficient Processing of Deep Neural Networks: A Tutorial and Survey*, Proc. IEEE 2017.

Spatial architectures turn data reuse into wires between neighbours instead of trips to memory.

Core idea (Kung, 1978–82)

- like a heart (*systole*), data is pumped from PE to PE every clock;
- each value read from memory is used by many PEs;
- nearest-neighbour wires only: scales, clocks fast.

Weight-stationary GEMM
weights w_{ij} preloaded; activations flow right (skewed one cycle per row); partial sums flow down.

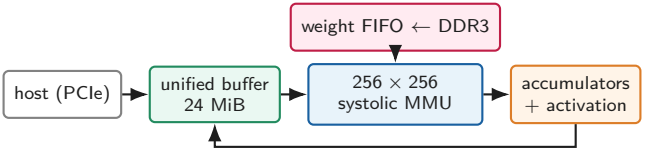


Kung & Leiserson, *Systolic Arrays (for VLSI)*, 1978; Kung, *Why Systolic Architectures?*, IEEE Computer 1982.

An $N \times N$ array does N^2 MACs per cycle while reading only $2N$ values.

Case in point: Google TPU v1 (in service 2015, published 2017)

65 536	92	28	30–80×
INT8 MACs	TOPS peak	MiB on chip	TOPS/W
a 256 × 256 systolic array	at 700 MHz	software-managed buffers	vs CPUs and GPUs of its time



Inference only, driven by the host. Many workloads were **memory-bound**: with GPU-class GDDR5, the authors estimate ∼3× more performance.

Jouppi et al., *In-Datacenter Performance Analysis of a Tensor Processing Unit*, ISCA 2017 (arXiv:1704.04760).

Even a 92-TOPS ASIC is limited by memory bandwidth – the roofline applies to everyone.

Dataflows: what stays still inside the array?

Dataflow	Idea	Minimises	Examples
Weight stationary	each PE keeps a weight; inputs and partial sums move	weight reads	TPU, NVDLA
Output stationary	each PE accumulates one output pixel	partial-sum traffic	ShiDianNao
Input stationary	each PE keeps an input value	input reads	SCNN (sparse)
Row stationary	each PE keeps one row of filter and input	overall energy	Eyeriss
No local reuse	large global buffer, no PE storage	PE area	DianNao

Chen, Emer & Sze, *Eyeriss*, ISCA 2016; Sze et al., Proc. IEEE 2017; Du et al., ISCA 2015 (ShiDianNao).

A dataflow is a loop order baked into hardware: it decides which operand is reused locally.

Tiling: making a big layer fit in a small buffer

```
// tile sizes chosen so W, I, O tiles fit on chip
for (mt = 0; mt < C_out; mt += Tm) // DDR -> BRAM
 for (yt = 0; yt < H_out; yt += Ty)
  for (ct = 0; ct < C_in; ct += Tc) {
    load_tile(W, I); // burst reads
    for (m = mt; m < mt+Tm; m++) // UNROLL: Tm PEs
     for (c = ct; c < ct+Tc; c++) // UNROLL: Tc lanes
      for (y..) for (x..) for (i..) for (j..)
        O[m][y][x] += W[m][c][i][j]*I[c][y+i][x+j];
 }
 store_tile(O); // burst write
```

Outer loops: temporal

a sequence of tiles streamed through DDR

Inner loops: spatial

unrolled into $T_m \times T_c$ MAC units

Tile sizes

trade on-chip memory for DDR traffic

peak = PEs × frequency;
achieved = peak × utilisation.

Zhang et al., *Optimizing FPGA-based Accelerator Design for Deep CNNs*, FPGA 2015.

Tiling is how a finite accelerator executes an arbitrarily large layer – and it is where the roofline is won or lost.

A decade of AI chips: each exploits one more kind of structure

Chip	Year	Key idea	Exploits
DianNao	2014	dedicated NN datapath with on-chip buffers	regular MACs
Eyeriss	2016	row-stationary spatial array, skips zeros	reuse + zeros
EIE	2016	compressed sparse layers served from on-chip SRAM	sparsity
TPU v1	2017	256×256 INT8 systolic array	GEMM + INT8
NVDLA	2017	open-source configurable inference engine	conv pipeline
Apple ANE, mobile NPUs	2017+	low-power INT8/FP16 engines inside phone SoCs	edge inference
LLM-era engines	2020+	2:4 sparsity, FP8/INT4, attention units	sparsity, low bits

Chen et al., ASPLOS 2014; Chen et al., ISCA 2016; Han et al., ISCA 2016; Jouppi et al., ISCA 2017; NVIDIA NVDLA (2017).

Algorithm and hardware progress are coupled: compression research (pruning, quantization) directly shaped accelerator design.

FOR READING

FPGAs and HLS, in more detail

Design flows, AXI and DMA, pipelining and unrolling, streaming and fusion, resources – background for Lab 6.

Why FPGAs for frugal AI – and what they cost you

Strengths

- **any precision**: INT8, INT4, binary, mixed;
- **deterministic latency**: no OS jitter in the datapath;
- **streaming I/O**: sensor → network → output without DRAM;
- **custom operators** a vendor NPU does not support;
- **reconfigurable** as models evolve – even in orbit.

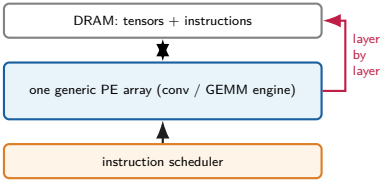
Costs

- lower clock (200–500 MHz vs GHz);
- programmable routing costs area and power vs an ASIC;
- long compile times (synthesis, place & route);
- hardware design skills required;
- limited on-chip memory (a few MB).

FPGAs win when the problem needs custom precision, custom dataflow or tight I/O – not raw peak TOPS.

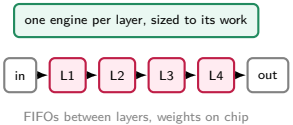
Two approaches to building an FPGA accelerator

A. Overlay: a processor-like engine



compile the network into instructions; the same bitstream runs many models. **e.g. AMD DPU / Vitis AI.**

B. Dataflow: the network is the circuit



all layers work concurrently on a stream; minimal DRAM traffic, but the network must fit on chip. **e.g. FINN, hls4ml, custom HLS.**

Overlay: flexible, any model size, DRAM-bound. Dataflow: lowest latency and traffic, but only what fits.

What the accelerator demands in return

A fixed operator menu
If a layer is not in the compiler's vocabulary, it is cut out of the graph and run on the CPU instead. One unsupported activation can cost more than pruning ever gave you.

Static shapes
Buffers are sized at compile time. Variable input sizes and data-dependent branches must be designed away or padded.

Integer arithmetic
Weights and activations arrive as INT8 with per-tensor scales. Anything that needs more range must be handled explicitly.

A memory layout
Tensors are stored in the order the engine reads them. A transpose you did not ask for is a real cost.

The Vitis AI path: PyTorch → quantizer (PTQ / QAT to INT8) → compiler (XMODEL for one DPU) → VART runtime on the Arm cores.

"Supported" is a contract: it decides whether your speed-up is real.

Design flows: from hand-written RTL to model compilers

Flow	You write	Trade-off
RTL	Verilog / VHDL	full control; slowest to write and verify
HLS	C++ plus pragmas (Vitis HLS)	fast design-space exploration; still requires hardware thinking
Generators	a trained model → HLS / RTL (hls4ml, FINN, MATLAB HDL Coder)	automatic; limited operator set and model size
Vendor IP	a trained model → DPU instructions (Vitis AI)	fastest path; fixed architecture and operator menu

Every flow ends the same way: simulation, co-simulation, synthesis, place & route, timing closure – and a measurement on the board.

The course moves from using an accelerator (the DPU) to understanding how to build accelerator datapaths (HLS).

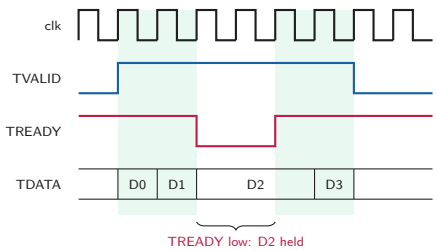
Protocols: how blocks talk – the AXI4 family

AXI4 (memory-mapped)
bursts to and from DRAM; five channels (AW, W, B, AR, R); up to 256 beats per burst

AXI4-Lite
control and status registers: start, done, sizes, addresses

AXI4-Stream
point-to-point data flow, no addresses: TDATA, TVALID, TREADY, TLAST

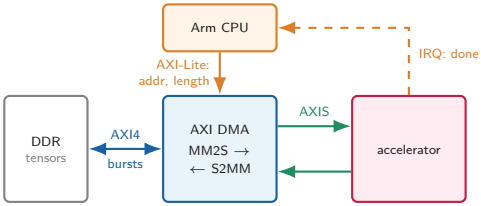
AXI4-Stream valid/ready handshake



A beat transfers only when **TVALID and TREADY** are both high (green).

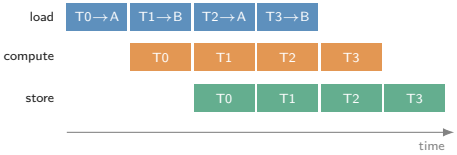
Memory-mapped AXI moves tensors; AXI-Stream moves a schedule – back-pressure keeps a pipeline correct without global control.

DMA: moving tensors without the CPU



- the CPU writes **descriptors**; the DMA issues the bursts;
- long contiguous bursts are efficient; strided ones are not;
- one 128-bit port at 300 MHz: at most 4.8 GB/s;
- caches must be flushed / invalidated around each transfer.

Double buffering (ping-pong)

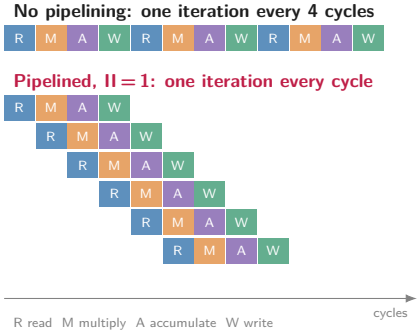


The DMA fills buffer B while the PEs read buffer A, then they swap. Transfers hide behind compute – **if** compute is the longer of the two.

Cost: every on-chip buffer is allocated twice.

DMA plus double buffering turns "load, compute, store" into a pipeline – at the price of on-chip memory.

Pipelining: the initiation interval (II)



$$\text{cycles} \approx (N-1)\cdot \text{II} + \text{depth}$$

For large N only the II matters: $N = 10^6$ iterations at 300 MHz take 3.3 ms at $\text{II} = 1$ and 6.7 ms at $\text{II} = 2$.

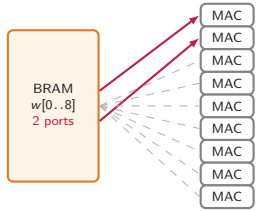
What forces $\text{II} > 1$:

- **memory ports**: a BRAM has 2; four reads per cycle need four ports;
- **loop-carried dependencies**: `acc +=` ... with a multi-cycle adder;
- **shared resources**: one DSP used twice per iteration.

Pipelining multiplies throughput without adding MAC units. The HLS report's II is the first number to check.

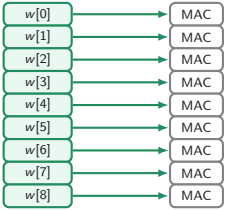
Unrolling needs memory ports: array partitioning

Unrolled $\times 9$, one BRAM



9 MACs, 2 reads per cycle $\Rightarrow$ the tool raises II to 5. Most of the DSPs sit idle.

Unrolled $\times 9$, array partitioned

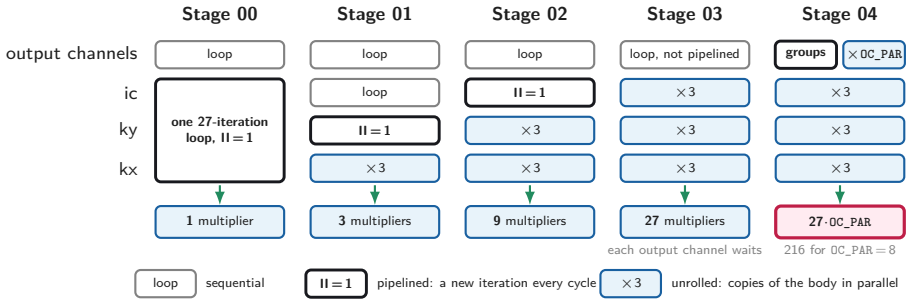


Each weight in its own register (`complete`) or bank (`cyclic`/`block`): 9 reads per cycle, II = 1.

Bandwidth demand = unroll factor $\times$ bit width $\times$ f_{clk}. On an FPGA, the **number of BRAMs is often set by the ports you need, not by the bits you store** – and partitioning into registers costs LUTs and flip-flops.

Parallel compute needs parallel memory. Every UNROLL must come with a matching ARRAY_PARTITION.

From loops to multipliers: the unrolling ladder of S6



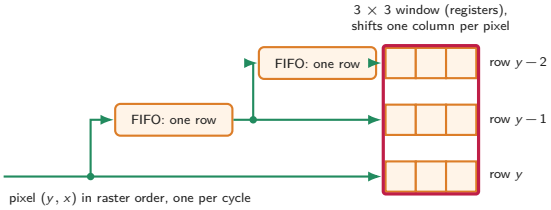
Lab 6 sources, Stages 00–04: one 3×3 convolution over 3 input channels, 27 MACs per output value. Stage 04 pipelines the loop over groups of `OC_PAR` output channels and unrolls the lanes (shown with `OC_PAR` $= 8$).

The layer and its MAC count never change. Each version unrolls one more loop of the same nest: more multipliers work in parallel, so fewer cycles are needed per frame.

With $\mathrm{II} = 1$, cycles per frame $\approx$ MACs per frame / MACs per cycle, and FPS $= f_{clk}$/cycles per frame. The lab question: the *smallest* parallelism that meets 60 FPS.

Parallelism is bought with DSPs, and every unrolled multiplier needs its operands in the same cycle.

Streaming convolution: line buffers and sliding windows



one cascade per input channel: $K-1=2$ FIFOs, each delaying one row of W pixels; the layer keeps $C_{in}(K-1)\,W$ pixels on chip, not the frame

Lab 6, Stages 01–04: `hls::stream<data_t> linebuf[IN_CH][K-1]`, declared with depth $2W$ (slack; the delay is one row).

- pixels arrive in raster order, one per clock; the image never exists in full on chip;
- each input channel keeps $K-1$ line FIFOs of one row each:

$$\text{FIFO words} = C_{in}\,(K-1)\,W$$

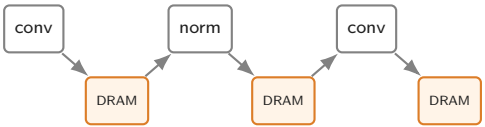
- each new pixel shifts the window by one and completes one output pixel;
- the output is itself a stream: the next layer can start immediately.

For a 1920×1080 frame and $K=3$: 2 lines instead of 1 080 – a **540**× smaller buffer.

A line buffer turns a whole-image operation into a stream: memory drops from the full frame to a few rows.

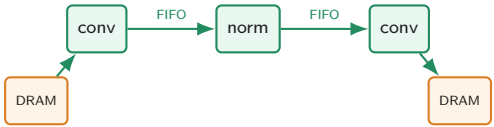
Layer at a time vs streaming dataflow – and why fusion matters

Layer at a time what a general accelerator does



every intermediate feature map is written to DRAM and read back:
for a 720p tensor that is megabytes, per layer, per frame

Streaming dataflow what HLS lets you build



results pass directly from one stage to the next
through small on-chip buffers: DRAM touched once in, once out

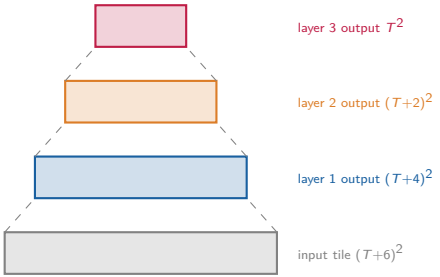
Operator fusion: $X \xrightarrow{\text{Conv+BN+ReLU}} C$ instead of materialising A and B. For a $64 \times 224 \times 224$ INT8 map, that avoids $\approx$12.8 MB of traffic.

BatchNorm folding (free at inference):

$$W' = \frac{\gamma}{\sqrt{\sigma^2+\epsilon}}\, W, \quad b' = \frac{\gamma\,(b-\mu)}{\sqrt{\sigma^2+\epsilon}} + \beta$$

Fusion and streaming do not reduce arithmetic at all. They reduce *traffic* – which is where the energy and the time went.

Fusing several layers: pyramids and halos



three fused 3×3 layers: each output tile needs a halo of $K-1$ extra rows and columns per layer.

- only the input tile is read from DRAM and only the last output is written back;
- intermediate tiles live in on-chip buffers;
- neighbouring pyramids **overlap**: either recompute the halo, or cache it on chip;
- larger tiles $\Rightarrow$ less overlap, but more on-chip memory.

Fusing the first five convolutional layers of a VGG network cut off-chip transfers per image by **95%** (77 MB $\rightarrow$ 3.6 MB) in the original fused-layer accelerator.

Alwani, Chen, Ferdman & Milder, *Fused-Layer CNN Accelerators*, MICRO 2016.

Multi-layer fusion trades on-chip memory (or recomputation) for DRAM traffic – the same currency exchange as tiling, one level up.

HLS: describing concurrency, storage and communication in C++

```
void conv_bn_relu(stream<pix_t>& in, stream<pix_t>& out) {
#pragma HLS INTERFACE axis port=in
#pragma HLS INTERFACE axis port=out
#pragma HLS DATAFLOW
  stream<win_t> win;
#pragma HLS STREAM variable=win depth=4
  line_buffer(in, win); // (K-1) rows in BRAM
  mac_bn_relu(win, out); // fused Conv+BN+ReLU
}
void mac_bn_relu(stream<win_t>& win, stream<pix_t>& out) {
  for (int p = 0; p < H*W; p++) {
#pragma HLS PIPELINE II=1
    win_t w = win.read();
    acc_t acc = 0;
    for (int k = 0; k < K*K; k++) {
#pragma HLS UNROLL
      acc += w.px[k] * Wf[k]; // BN folded
    }
    out.write(relu(acc + bf));
  }
}
```

Directive	Hardware meaning
`PIPELINE II=1`	a new iteration every clock
`UNROLL`	K^2 parallel MACs (DSPs)
`ARRAY_PARTITION`	more memory ports
`DATAFLOW`	functions run concurrently
`STREAM`	a FIFO channel
`INTERFACE axis`	an AXI4-Stream port

At the labs' 100 MHz with $\mathrm{II} = 1$: one pixel per cycle $\Rightarrow$ a 128×128 input in 0.16 ms, a 1920×1080 frame in $\approx$21 ms (48 FPS).

HLS is not "C++ but faster": it describes concurrency, storage and communication.

The resource budget: what constrains an FPGA accelerator

DSP slices → MACs / cycle

KV260: 1 248 DSPs × 2 INT8 MACs × 300 MHz ≈ **1.5 TOPS** ceiling. LUT-based MACs add more for low bit widths.

BRAM / URAM → buffers

26.6 Mb total for tiles, FIFOs, line buffers and double buffering – and ports, not bits, often decide the count.

DDR → bytes / second

≈19.2 GB/s peak, **shared with the CPU**; achieved is lower for short or strided bursts.

LUT / FF → control

address generators, adders, partitioned arrays, AXI logic. High utilisation causes routing congestion.

Timing → clock frequency

long paths and congestion lower f_{clk}. Above ∼70–80% utilisation, closure gets hard.

Power and thermal

7.5 W typical for the whole module. Toggle rate, clock and I/O all count.

$$t_{layer} \approx \max\left(\frac{\text{MACs}}{\#\text{PE} \cdot f_{clk} \cdot u}, \frac{\text{bytes moved}}{BW}\right) \qquad u = \text{utilisation}$$

Designing an accelerator is balancing these six budgets so that no single one idles the others.

Sizing the on-chip memory: a worked example

Layer: 3×3 conv, $256 \to 256$ channels, 56×56. Tiles $T_m = T_c = 32$ channels, 14×14 pixels.

Buffer	Size	Bytes
weights	$T_m T_c K^2$	9 216
inputs	$T_c(T+K-1)^2$	8 192
outputs	$T_m T^2 \times 4$ (INT32 acc.)	25 088
total		42.5 KB
$\times 2$	double buffering	**85 KB**

$\approx$19 BRAM36 by capacity – but 32×32 parallel reads need far more *banks*.

DRAM traffic with these tiles

weights, reloaded per spatial tile	9.4 MB
inputs, reloaded per output-channel tile	8.4 MB
outputs, written once	0.8 MB
total vs. **2.3 MB** minimum	**18.6 MB**

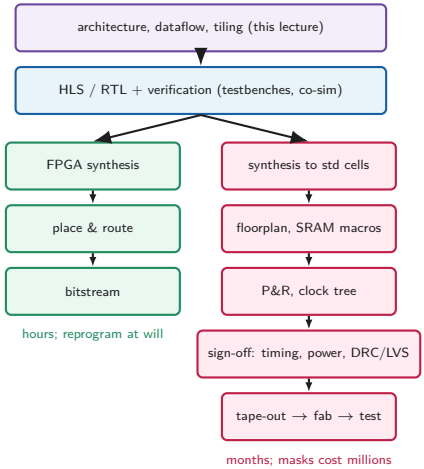
6.0 ms
compute: 1.85 GMAC on 1 024 MACs @ 300 MHz

0.97 ms
memory: 18.6 MB at 19.2 GB/s

Compute-bound: double buffering hides the transfers. Quadruple the PEs (1.5 ms) and memory is close behind – larger tiles become necessary.

Tile sizes set three numbers at once: BRAM usage, DRAM traffic, and which of compute or memory is the bottleneck.

From FPGA to ASIC: same architecture, different silicon



The FPGA → ASIC gap

area	21–40×
critical-path delay	3–4×
dynamic power	∼12×

Kuon & Rose, ACM FPGA 2006 (smaller with hard DSP/RAM blocks).

- BRAM → SRAM macros of exactly the size needed;
- DSP → MACs of exactly the width needed (INT4, INT2);
- verification dominates: a bug is permanent;
- FPGA first, as the prototype;
- to learn: SkyWater 130 nm PDK, OpenROAD, Tiny Tapeout.

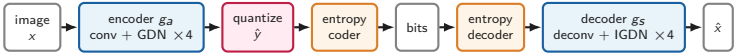
Dataflow, tiling and buffering carry over; being wrong just costs more.

FOR READING

Case studies

Learned image compression on an FPGA SoC, an FPGA accelerator built for space, and on-device learning.

Case 1 – Learned image compression on an FPGA SoC



Huge feature maps

Input and output are full HD or 4K images; intermediate tensors dwarf the weights.

An unsupported operator

GDN normalisation is not in the DPU's menu: it needs a custom hardware core or a replacement.

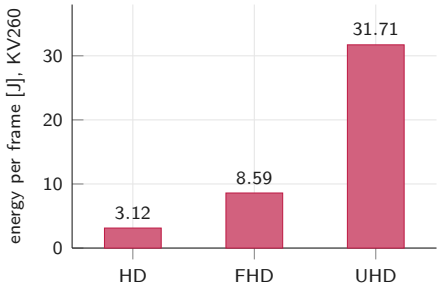
Quality is a curve

Measured as rate–distortion (bits per pixel vs PSNR), not as a single accuracy number.

Mazouz et al., arXiv:2503.04832 and arXiv:2503.04867 (2025); model family after Ballé et al., ICLR 2018.

LIC is a stress test for everything in this lecture: activations, operator support, quantization and streaming.

Case 1 – Same weights, nine times the work



The model file is byte-for-byte identical in all three runs. Only the **input tensor** changed – and with it every feature map inside the network.

HD → UHD: 9× the pixels, 10× the energy.

What made it fit:

- a distilled, narrower **student** model;
- **QAT** to INT8, with the sensitive layers kept at higher precision;
- several DPU cores working on image **patches** in parallel, plus a custom core for GDN.

Internal KV260 measurements; Mazouz et al., arXiv:2503.04832 (2025).

Resolution is not a cosmetic parameter: it is a multiplier on memory, traffic and joules.

Case 2 – An FPGA accelerator built for space: learning in orbit

Why train on board?

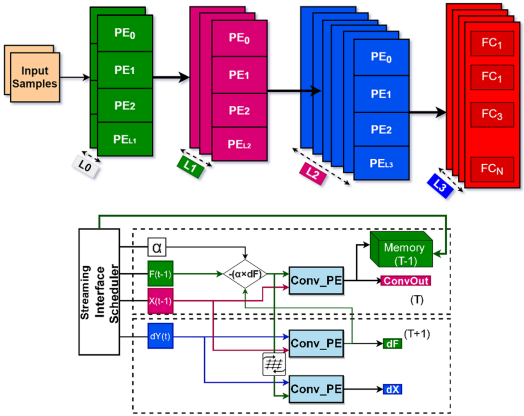
Close-proximity operations (docking, inspection) meet targets and lighting never seen on the ground; the data cannot all be downlinked.

Why an FPGA?

Streaming pipelines at low power, custom precision, and reconfiguration in orbit.

The design

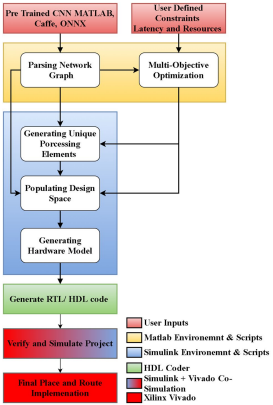
Forward *and* backward passes as a layer-pipelined streaming architecture, generated automatically from a trained CNN.

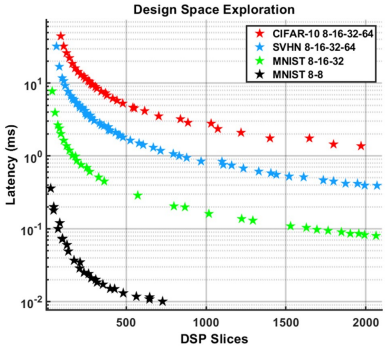


Mazouz & Bridges, *Multi-Sensory CNN Models for Close Proximity Satellite Operations*, IEEE Aerospace 2019; *Automated CNN Back-Propagation Pipeline Generation for FPGA Online Training*, J. Real-Time Image Processing 18:2583–2599, 2021.

Training is the hardest edge workload of all: every feature map must be kept for the backward pass.

Case 2 – From a trained CNN to RTL, and the price of parallelism





Left: the network graph and user constraints (latency, resources) drive a generator that emits processing elements and HDL.

Right: latency vs DSP slices for generated designs – more parallelism buys lower latency, with diminishing returns.

Mazouz & Bridges, J. Real-Time Image Processing, 2021.

Design-space exploration = finding the knee of the curve that meets the latency budget with the fewest resources.

Beyond this course: on-device learning

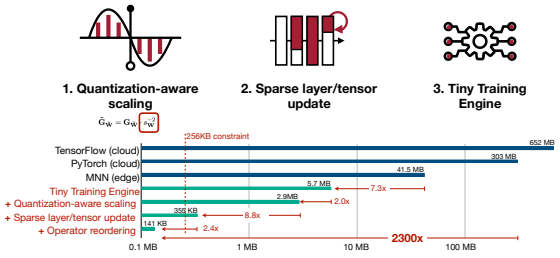


Figure: MIT HAN Lab 6.5940, Lecture 01; Lin et al., *On-Device Training Under 256KB Memory*, NeurIPS 2022.

Why

adapt to drift and to each user, keep data private, keep learning when there is no link (orbit, remote sensors).

Why it is hard

the backward pass keeps every activation, gradients need range, the optimizer adds state: memory grows several-fold.

How it is made to fit

update only some layers or biases, sparse updates, quantized training, compile the backward graph ahead of time.

We will not cover training on the device in this course – but every technique we do cover is a prerequisite for it.