Embedded Artificial Intelligence

SETI Master

Lab 2 — Quantization for Embedded AI

From PyTorch intuition to Vitis AI and the KV260

Author:	Alaa Eddine Mazouz
Academic year:	2026–2027
Version:	10.10
Date:	26 October 2026
Model:	supplied CIFAR-10 CNN (56.95 % top-1)
Target:	AMD Kria KV260, DPU B4096

Télécom Paris

Work in the lab folder `TP02_quantization/`, which contains `scripts/` and `src/`.

Host shell. In each new host terminal, type `bash` and press Enter separately. Wait for its prompt before pasting the commands below; the teaching hosts default to zsh. Commands launched with `bash script.sh` already select Bash. The initial `cd TP02_quantization` assumes the course checkout root.

Raw datasets. Raw data are shared in the course-root `datasets/` folder; a standalone lab uses `$HOME/.cache/c5/datasets` (or `$XDG_CACHE_HOME/c5/datasets`). Set `C5_DATA=/absolute/path` before sourcing `scripts/00_source_env.sh` to override it. Existing datasets stay in place; experiment outputs stay in this lab.

Purpose of this lab

Your first quantization lab and your first deployment on the KV260: make quantization visible with a few PyTorch tensor operations, quantize one CIFAR-10 CNN with Vitis AI (PTQ, fast fine-tuning, QAT), then compile the PTQ and QAT models for the DPU, run them with XIR and VART, and compare them on the board. The concepts (scale, rounding and clipping, calibration, per-channel scales, PTQ and QAT, the DPU contract, XIR and VART, the four measurement levels) are in the L02 slides; this handout gives the commands and what to record.

Plan

Part	Content
A	float model: supplied checkpoint, inspection
B	PyTorch simulations: one tensor, one weight channel, one activation
C	Vitis AI PTQ: inspector, calibration, test, export; calibration experiment; fast fine-tuning (optional)
D	Vitis AI QAT
E	compile, deploy and compare PTQ and QAT on the KV260
F	final task: one controlled experiment

Run a command, then read on or answer the checkpoint while it runs: every result stays in `out/runs/<TAG>/`.

Before starting

Each listing starts with a comment that says where it runs:

- `# host`: a terminal on the lab machine, in the `TP02_quantization` folder;
- `# container`: the Vitis AI shell opened by `02_enter_container.sh`, with PyTorch, the quantizer and the compiler (prompt `(vitis-ai-pytorch)`);
- `# KV260`: a shell on the board (`ssh root@maquette-07`, see Step E3).

Every script runs from any folder, but the commands below assume the lab folder.

TAG, QTAG and the data

- **TAG** names one run, built on one float source model: `out/runs/<TAG>/`. The script that creates the float model refuses an existing TAG; `OVERWRITE=1` deletes it and starts again. Use a new TAG when the source model changes. Every later script needs only `TAG=...`.
- **QTAG** names one quantized variant of that source: `quant/<QTAG>/`, then `deploy/<QTAG>/` and `board_results/<QTAG>/`. Running calibration (or QAT) again with the same QTAG *replaces* that variant: use a new, descriptive QTAG for every new experiment.

- **Data.** CIFAR-10 has 50,000 training and 10,000 test images. Calibration, fast fine-tuning and QAT training use training images; every accuracy is measured on the test set. Nothing is selected from the per-epoch test accuracy printed by training and QAT: it only monitors progress, and the model after the last epoch is kept. In this lab the test accuracy is also used to compare the methods; in a real project that comparison would use a validation split and keep the test set for the final report (Lab 3 does so).
- **Results.** `TAG=${TAG} bash scripts/09_compare_results.sh` rebuilds the result table at any time.

CIFAR-10 (170 MB) is not downloaded again when Lab 1 is in the same folder as this lab, or when the lab machine has the course's shared copy: `00_source_env.sh` links `data/cifar-10-batches-py` to the dataset folder of the course (`datasets/` next to the lab folders, see Lab 1). Otherwise the first script downloads it, once.

The appendix lists every script.

The quantizer of this lab (recap of L02)

Part B uses a symmetric quantizer: for b bits, $q_{\max} = 2^{b-1} - 1$, a representable magnitude α and

$$s = \frac{\alpha}{q_{\max}}, \qquad q = \mathrm{clip}\left(\mathrm{round}\left(\frac{x}{s}\right), -q_{\max}, q_{\max}\right), \qquad \hat{x} = s\,q.$$

With 4 bits and $\alpha = 1$: $s = 1/7 \approx 0.143$, so $x = 0.55$ becomes $q = 4$ and $\hat{x} \approx 0.571$. A value inside the range has a rounding error; a value outside is clipped. The Vitis AI quantizer of Parts C and D uses the same symmetric scale, restricted to a power of two, and the codes -128 to 127 (L02). Every variant starts from the same FP32 checkpoint of the TAG: the result tables start with `float_source` and give every variant as a change from it. QAT adds training, so its change covers the whole workflow, not quantization alone (Part D separates the two).

1 Part A — The float model

```
# host
cd TP02_quantization
source scripts/00_source_env.sh
bash scripts/02_enter_container.sh
```

`02_enter_container.sh` runs the `apptainer exec` command of Lab 1 on the image `$VAI_SIF`, chosen as in Lab 1 by `scripts/course_env.sh` (the same file in every lab). The container shell activates `vitis-ai-pytorch` and prints `[TP02] Container ready: conda env vitis-ai-pytorch, torch <version>`. In another shell inside the container, activate it with:

```
# container
source /opt/vitis_ai/conda/etc/profile.d/conda.sh && conda activate vitis-ai-pytorch
```

Step A0 — choose the float model

```
# container
export TAG=your_tag
TAG=${TAG} bash scripts/03_use_reference_model.sh
```

This copies the supplied checkpoint (`reference/`) into `out/runs/${TAG}/float/` and measures it on the test set: a two-convolution CIFAR-10 CNN with 67,642 parameters, 56.95 % top-1 (the CNN of Lab 1 on CIFAR-10, trained for 5 epochs).

You may instead train your own source model, each under its own TAG (`--help` lists the choices: `DATASET`, `ARCH=cnn|mlp`, `WIDTH`, `EPOCHS`, `LR`, `BATCH_SIZE`, `SEED`):

```
# container
TAG=${TAG}_cnn_w2 ARCH=cnn WIDTH=2 EPOCHS=8 bash scripts/03_train_float.sh
TAG=${TAG}_mlp ARCH=mlp WIDTH=1 EPOCHS=12 DATASET=mnist bash scripts/03_train_float.sh
```

Both routes write `meta.json`, `float/model_float.pth` and `float/metrics.json`, which every later script reads. The supplied CNN is the safe path to the board: the board application runs one DPU subgraph, so another model reaches the board only if the inspector and the compiler place all of it on the DPU.

Step A1 — inspect before quantizing

```
# container
TAG=${TAG} bash scripts/03b_describe_float_model.sh
```

The report (also in `concepts/model_description/`) gives the input shape and classes, the parameter count and FP32 storage, the `Conv2d`/`Linear` layers with their output channels (for Step B2), and the activation outputs (for Step B3).

Checkpoint A

Compare a wider CNN or an MLP (from Lab 1 or trained above) with the supplied CNN. Which has fewer parameters? Which structure maps more naturally to a convolution accelerator? Write a prediction before running the inspector in Part C.

2 Part B — Quantization with PyTorch tensor operations

These activities use ordinary PyTorch operations on the float model of Part A. They do not use Vitis AI and produce nothing deployable; Vitis AI automates the same choices in Part C. For each one: predict, run one command, read the table, then change exactly one parameter.

Step B1 — one small tensor

```
# container
TAG=${TAG} BITS=4 RANGE_ABS=1.0 bash scripts/03c_quantize_numbers.sh
TAG=${TAG} BITS=4 RANGE_ABS=2.0 bash scripts/03c_quantize_numbers.sh
TAG=${TAG} BITS=8 RANGE_ABS=1.0 bash scripts/03c_quantize_numbers.sh   # optional
```

Nine values from -1.8 to 1.8 follow $x \to q = \mathrm{clip}(\mathrm{round}(x/s)) \to \hat{x} = sq \to e = x - \hat{x}$; the table gives each step, whether the value was clipped, and the MSE and SQNR. INT4 only makes the levels visible; the DPU runs INT8.

Guided observation B1

Before the second command, predict what happens to the clipping and to the step s. Why can a wider range remove clipping while making small values less precise? What does the 8-bit run change, and what does it not change?

Step B2 — one real weight channel

```
# container
TAG=${TAG} LAYER=conv2 CHANNEL=0 BITS=8 GRANULARITY=per_tensor \
  bash scripts/03d_quantize_weights.sh
TAG=${TAG} LAYER=conv2 CHANNEL=0 BITS=8 GRANULARITY=per_channel \
  bash scripts/03d_quantize_weights.sh
```

The program prints the range of every output channel, the MSE and SQNR over the whole weight tensor, and the codes of a few weights of the selected channel. It also saves `channel_ranges.png`.

Guided observation B2

Are all channels equally large? Which granularity gives the lower error, and why? Set `CHANNEL` to the smallest-range channel and check your explanation. Repeat on `fc1`: is the gain larger, and why?

Step B3 — one activation layer

```
# container
TAG=${TAG} LAYER=conv1 PROFILE=representative CALIB_SAMPLES=128 BITS=8 \
  bash scripts/03e_calibrate_activations.sh
TAG=${TAG} LAYER=conv1 PROFILE=dark CALIB_SAMPLES=128 BITS=8 \
  bash scripts/03e_calibrate_activations.sh
```

The range is estimated on 128 training images (representative, or darkened by `DARK_SCALE`=0.25), then applied to the activations of 128 normal test images: the program reports the estimated range, the clipped fraction, the MSE and the SQNR. The model, evaluation images, bit width and sample count do not change.

Guided observation B3

Compare range, clipping, MSE and SQNR. Why does the same INT8 format perform differently when only the calibration images change? Repeat on a deeper layer (`conv2`, `fc1`): is the effect stronger or weaker?

3 Part C — Vitis AI PTQ

From now on the quantizer is `vai_q_pytorch` (Python package `pytorch_nndct`); its VAIQ/NNDCT start-up message is expected in Parts C and D. The reusable program is `src/vitis_quantize.py`, built like AMD's `resnet18_quant.py` example. Part C creates the variant `QTAG=ptq_ref`.

Step C0 — check the Vitis AI APIs

```
# container
python3 -c "from pytorch_nndct.apis import torch_quantizer, Inspector; print('OK')"
```

Step C1 — inspect the model for the DPU

```
# container
TAG=${TAG} SOURCE=base bash scripts/04a_inspect_dpu.sh
cat out/runs/${TAG}/inspection/base/inspection_report.txt
```

```
example = torch.zeros(1, C, H, W)
inspector = Inspector("DPUCZDX8G_ISA1_B4096")
inspector.inspect(model, (example,), device=torch.device("cpu"))
```

The inspector predicts which operations this DPU can run; it measures no accuracy and does not replace compilation. A fully connected layer may appear as a convolution-like XIR operation: that does not mean it is unknown.

Checkpoint C1

Does the inspector report a complete DPU path for the supplied CNN? If you trained an MLP, compare its report. Record the first direct incompatibility, if any, not its consequences.

Step C2 — the `torch_quantizer` call

```
quantizer = torch_quantizer(
    quant_mode=quant_mode,       # "calib" or "test"
    module=float_model,
    input_args=(example,),       # shape template: (1, C, H, W)
    output_dir=quantize_dir,
    target="DPUCZDX8G_ISA1_B4096",
)
quant_model = quantizer.quant_model
```

`calib` mode collects the ranges while calibration images pass through; `test` mode rebuilds the quantized model from them, to evaluate or export it. `input_args` is a batch-1 signature: calibration and evaluation may use larger batches, only the export must run one batch-1 input. `target` selects the defaults of this DPU (a strategy file, `QUANT_CONFIG`, replaces them).

Step C3 — phase 1: calibration

```
# container
TAG=${TAG} QTAG=ptq_ref CALIB_SAMPLES=512 CALIB_BATCH_SIZE=32 \
  CALIB_PROFILE=representative bash scripts/04b_ptq_calibrate.sh
find out/runs/${TAG}/quant/ptq_ref/quantize_result -maxdepth 2 -type f | sort
```

```
for images, _ in calibration_loader:
    quant_model(images)          # labels are not used
quantizer.export_quant_config()
```

The 512 training images only pass forward; the weights do not change. The folder now holds the quantization information (`Quant_info.json`) and generated model code; every phase writes its log to `out/runs/${TAG}/logs/`.

Checkpoint C2

Calibration prints no accuracy. Why would an accuracy measured in calibration mode not be the final INT8 accuracy? What does calibration use from the images, what does it ignore, and which artifact must be kept for the next phase?

Step C4 — phase 2: test mode on the host

```
# container
TAG=${TAG} QTAG=ptq_ref TEST_BATCH_SIZE=128 TEST_SAMPLES=0 \
  bash scripts/04c_ptq_evaluate.sh
TAG=${TAG} ACCURACY_TOLERANCE=0.5 bash scripts/09_compare_results.sh
```

Test mode reads the configuration of phase 1 (no recalibration) and evaluates `quant_model` on the whole test set. The table accepts a variant when $A_{\text{float}} - A_{\text{quantized}} \leq 0.5$ point (`ACCURACY_TOLERANCE`).

Checkpoint C3

Record the float accuracy, the INT8 host accuracy and the loss in points. If PTQ meets the requirement, say explicitly that a more expensive method is not yet justified.

Step C5 — read the quantizer's decisions

```
# container
TAG=${TAG} QTAG=ptq_ref bash scripts/04e_show_quant_params.sh
TAG=${TAG} QTAG=ptq_ref FILTER=conv1 bash scripts/04e_show_quant_params.sh
```

For each tensor Vitis AI records a bit width and a fixed-point position f: step $s = 2^{-f}$, range about $\pm 127 \cdot 2^{-f}$ (power-of-two scales, L02).

Checkpoint C4

For one weight tensor and one activation, record bit width, fixed-point position, step and range. Which receives the finer step? Compare with your manual scales of Part B. Why can a very fine step be dangerous when outliers exceed the range?

Step C6 — phase 3: batch-1 XMODEL export

```
# container
TAG=${TAG} QTAG=ptq_ref bash scripts/04d_ptq_export_xmodel.sh
```

A new process rebuilds the test-mode quantizer, runs exactly one forward pass on one real $(1, C, H, W)$ image, calls `quantizer.export_xmodel()` and checks the XMODEL input with XIR. It must print `[DEPLOY] preflight : PASS - exported graph input batch is 1`; the XMODEL is `quant/ptq_ref/model.xmodel`.

The batch size is part of the exported graph

An XMODEL exported after a batch-128 evaluation keeps batch 128: the compiler then puts `conv2d-fix` and `pool-fix` on the CPU (`input batch to be 1, but it's 128 now`), and the later warnings are consequences. This is why the export is a separate command, which refuses any setting other than `--batch_size 1 --subset_len 1`.

Step C7 — the shortcut for later labs

```
# container
TAG=${TAG} QTAG=ptq_ref CALIB_SAMPLES=512 CALIB_PROFILE=representative \
  bash scripts/04_ptq_export.sh
```

It runs `04b`, `04c` and `04d` in this order, with the same variables. Keep `src/vitis_quantize.py` as the template to quantize the pruned and distilled models of the next labs.

Step C8 — one controlled calibration experiment

```
# container
TAG=${TAG} QTAG=ptq_mismatch CALIB_SAMPLES=512 CALIB_PROFILE=dark MISMATCH_SCALE=0.25 \
  bash scripts/04_ptq_export.sh
TAG=${TAG} bash scripts/09_compare_results.sh
```

Only the calibration images change (darkened by a factor 0.25); the source weights, bit width and test set do not. Any difference between `ptq_ref` and `ptq_mismatch` is therefore due to calibration representativeness.

Checkpoint C5

Link the result to Step B3: how does a range estimated on dark images create clipping on normal images? Name the numerical assumption that fails, rather than saying that "the data were bad".

Fast fine-tuning (optional recovery between PTQ and QAT)

```
# container
TAG=${TAG} QTAG=ptq_fast_ft CALIB_SAMPLES=512 CALIB_PROFILE=representative \
  FAST_FINETUNE=1 FT_SAMPLES=512 bash scripts/04_ptq_export.sh
```

Fast fine-tuning adjusts the quantized weights layer by layer on a few unlabelled training images (`quantizer.fast_finetune`, then `load_ft_param` in test and export).

Checkpoint C6

Compare `ptq_fast_ft` with `ptq_ref`. A negligible gain is a result: representative PTQ already handled this model, and the extra step was not needed.

4 Part D — Vitis AI QAT

QAT uses another Vitis AI API, `QatProcessor`, around an ordinary PyTorch training loop (L02): the QAT copy of the model (`src/tp02_models.py`, `enable_quant=True`) marks the quantized region with `QuantStub` and `DeQuantStub`; `processor.trainable_model()` is trained with the rounding and clipping in its forward pass; `to_deployable()` converts it, and the export again runs one batch-1 input. Read `src/qat_export.py`. The whole run (2 epochs, learning rate 10^{-4}), variant `qat_base`:

```
# container
TAG=${TAG} QTAG=qat_base SOURCE=base QAT_EPOCHS=2 QAT_LR=0.0001 \
  bash scripts/04_qat_export.sh
TAG=${TAG} bash scripts/09_compare_results.sh
```

A positive change means the final workflow is more accurate than the original checkpoint. To ask whether the gain comes mostly from the extra training, add `FLOAT_CONTROL=1`: the script also trains a float copy for the same epochs and prints a separate *QAT training check* table (it is not a second starting point).

Checkpoint D — choose rather than rank

1. Does representative PTQ meet the tolerance? Does fast fine-tuning improve it materially?
2. Does QAT beat the original float checkpoint? By how many points does it change accuracy relative to `float_source`?
3. Which method would you deploy under the tolerance, and why?

A valid conclusion: "QAT gives the best host INT8 accuracy, but representative PTQ already meets the requirement and is the cheaper engineering choice."

For the later labs keep the sequence: inspector → representative PTQ → host test → batch-1 export → compile and check the partition. Escalate to fast fine-tuning or QAT only when the requirement is not met.

5 Part E — Compile, deploy and compare PTQ and QAT on the KV260

Quantizer XMODEL and compiled XMODEL

The Python application runs on the Arm cores of the KV260 and the DPU subgraph on the **DPUCZDX8G B4096** in the programmable logic. The quantizer XMODEL, in `quant/<QTAG>/`, is not yet specialized for a DPU; the compiler turns it into the XMODEL of `deploy/<QTAG>/`, for one DPU configuration:

```
vai_c_xir -x out/runs/${TAG}/quant/${QTAG}/model.xmodel -a arch.json \
          -o out/runs/${TAG}/deploy/${QTAG} -n model
```

`arch.json` holds the fingerprint of the DPU of the course boards, as printed by `xdputil query` (`0x101000056010407`); VART refuses a model compiled for another one. The board application of this lab runs one DPU subgraph, so all neural-network computation must map to one DPU partition.

A terminal, shape-preserving `fix2float` from a DPU output to a graph output is permitted; the runner dequantizes DPU logits itself. Never copy the quantizer XMODEL to the board.

Step E1 — compile the PTQ/QAT pair

```
# container
export PTQ_QTAG=ptq_ref QAT_QTAG=qat_base
TAG=${TAG} PTQ_QTAG=${PTQ_QTAG} QAT_QTAG=${QAT_QTAG} \
  bash scripts/05_compile_ptq_qat_pair.sh
```

For each variant, `05_compile_kv260.sh` checks the fingerprint and the batch-1 input, runs `vai_c_xir`, names the first direct CPU assignment (`compiler_diagnosis.json`) and requires one DPU subgraph, batch 1 and no CPU computation operations (`partition_report.txt`).

Checkpoint E1

For `ptq_ref` and `qat_base`, record the compiled input shape and the numbers of DPU and CPU operations. Both must show batch 1 and no CPU computation before the board. Record any permitted output conversion separately.

Step E2 — one shared labelled bundle

```
# container
TAG=${TAG} bash scripts/06_prepare_eval_bundle.sh
```

It writes the bundle (`uint8` images, labels, float logits) and its `bundle_meta.json` in `board_bundle/`: the whole test set by default, so host and board compare the same images. Both models use this bundle; the comparison checks its SHA-256.

Step E3 — your board, by hand first

Each KV260 of the room answers on the network to a name of the form `maquette-07`. Use the name of the board you are given: the `KV260-xx` sticker on a board is an inventory number, not its network name. The commands below use `maquette-07`; replace it by the name of yours. In Labs 2–5 the board runs the PetaLinux image with the Vitis AI runtime, and the account is `root`. Reach it by hand once:

```
# host
ping -c 2 maquette-07
ssh root@maquette-07              # first time: answer yes (records the board key)
# KV260
hostname
xdputil query | head -20       # the DPU of this board and its fingerprint
exit
# host: one folder and one file, by hand
ssh root@maquette-07 mkdir -p /home/root/c5/TP02
scp -O arch.json root@maquette-07:/home/root/c5/TP02/
ssh root@maquette-07 ls -l /home/root/c5/TP02
```

The course PetaLinux image has no SFTP server: `scp -O` selects the legacy SCP protocol. On an older host client that does not recognize `-O`, omit that option (legacy SCP is already its default). The later course copy wrappers choose the compatible protocol automatically. `ssh user@board command` runs one command on the board, and without a command it opens a shell (`exit` closes it). `scp files user@board:folder/` copies files to the board, `scp user@board:path .` copies them back. Every board folder of the course is under `/home/root/c5/` (`/home/ubuntu/c5/` on the Ubuntu image of Lab 6 and Lab 7). Optional: `ssh-copy-id root@maquette-07` installs your SSH key on the board (create one first with `ssh-keygen` if `~/.ssh/` has none), and the board stops asking for the password.

If the name does not answer: the serial console

The micro-USB port of the board is a serial console (and a JTAG port): it works without any network. Connect it to the lab machine and open the console with `tio`:

```
# host
ls /dev/serial/by-id/          # 4 interfaces; the console ends with -if01-port0
tio /dev/serial/by-id/*-if01-port0    # usually /dev/ttyUSB1, 115200 baud
# KV260 console: press Enter for the login prompt, log in as root
ip -4 addr show eth0           # the address follows "inet", e.g. 192.0.2.57
```

Leave `tio` with `ctrl-t` then `q`. The console also shows the boot messages: start `tio` before powering the board to see them, it waits for the device and reconnects by itself. `Permission denied` on the device means that your account is not in the `dialout` group (it must be added to it); no `inet` line on `eth0` means no network (cable, switch). Then use the address everywhere instead of the name: `ssh root@192.0.2.57`, and below `export BOARD=192.0.2.57` (with the address that you read).

Step E3 (continued) — check the board and copy the pair with the scripts

From now on, the scripts reach the board through one variable, `BOARD`, exported once per terminal. The container shell inherits it if you export it on the host before `02_enter_container.sh`; otherwise export it in the container shell too.

```
# container (or host)
export BOARD=maquette-07          # the name of your board, or its IP address
bash scripts/07_check_board.sh
TAG=${TAG} PTQ_QTAG=${PTQ_QTAG} QAT_QTAG=${QAT_QTAG} \
  bash scripts/07_copy_ptq_qat_pair.sh
```

The scripts run the `ssh` and `scp` commands above as `root` (`BOARD_USER` chooses another account, and an account written in `BOARD`, as in `root@maquette-07`, takes precedence), share one connection for 10 minutes, so the password is asked once, and stop with an explanation if `BOARD` is missing or malformed (one word: a name, an address or `user@address`; a name that the machine cannot resolve only gives a warning: `ssh` decides); `scripts/course_env.sh`, the same file in every lab of the course, holds this code. `07_check_board.sh` checks the network, SSH, the XIR/VART imports, `xdputil` and the DPU firmware, and changes nothing. The copy creates `/home/root/c5/TP02/<TAG>/ptq_ref/` and `.../qat_base/`, each with its compiled XMODEL and metadata, the same bundle, the board programs and `setup_env.sh`.

Step E4 — XIR and VART, once, with the PTQ model

```
# container (or host): TAG and PTQ_QTAG exist only in this shell, so the folder is expanded
    before ssh
ssh -t root@${BOARD#*@} "cd /home/root/c5/TP02/${TAG}/${PTQ_QTAG} && exec bash -l"
# KV260
source setup_env.sh
less vart_walkthrough_fpga.py
python3 vart_walkthrough_fpga.py --xmodel model.xmodel --bundle eval_bundle.npz \
  --index 0 --repeat 20
exit
```

`${BOARD#*@}` is `BOARD` without a `user@` part, so that this command and the ones of the next labs work whether `BOARD` holds a name, an address or `user@address`. The program follows the XIR/VART sequence of L02 (deserialize, find the DPU subgraph, create a runner, `execute_async`, `wait`); the input is converted as on the host ($x/255$, NHWC if needed) and to INT8 codes with the input fixed-point position. Record the DPU subgraph, the tensor shapes, the fixed-point positions, the expected and predicted classes. The QAT XMODEL runs with the same application.

Step E5 — evaluate PTQ and QAT

```
# container (the final table needs PyTorch)
TAG=${TAG} PTQ_QTAG=${PTQ_QTAG} QAT_QTAG=${QAT_QTAG} WARMUP=20 LIMIT=0 \
  bash scripts/07_evaluate_ptq_qat_pair.sh
```

For each variant it runs `vart_run_fpga.py` on the whole bundle after 20 warm-up inferences, fetches the results into `board_results/<QTAG>/`, checks the bundle checksum and prints one table (again with `TAG=${TAG} bash scripts/09_compare_ptq_qat_board.sh`).

Step E6 — the paired table

FP32 source (not deployed), then PTQ and QAT: host accuracy, KV260 accuracy, change from FP32, board–host gap, system FPS. Accuracy is the comparison of this lab; FPS is a consistency check (same architecture).

Checkpoint E2 — the deployment decision

1. Are PTQ and QAT evaluated on the same number of images and the same bundle checksum?
2. Does each KV260 accuracy agree with its host INT8 accuracy? What could explain a small gap?
3. By how many points does QAT gain or lose against PTQ on the KV260?
4. If PTQ already meets the tolerance, is the extra QAT training justified?

Choose QAT only when its measured recovery is useful, not because it is the more advanced method.

Step E7 — four measurement levels

The four levels of L02: application and VART FPS (`vart_run_fpga.py`), raw DPU throughput (`xdputil benchmark`), DPU time per run (`vaitrace`), SOM power (`xmutil xlnx_platformstats -p`). Run on the board, in a variant folder, after `source setup_env.sh`:

```
# KV260
python3 vart_run_fpga.py --help
python3 vart_run_fpga.py --xmodel model.xmodel --bundle eval_bundle.npz \
  --bundle_meta bundle_meta.json --detailed_report
xdputil benchmark model.xmodel -i -1 1    # -i -1: whole graph; last number: threads
xdputil benchmark model.xmodel -i -1 4
python3 vai_trace_fpga.py --workload_seconds 5 --trace_time 15 \
  --output_dir results/vaitrace -- --xmodel model.xmodel --model_info model_info.json \
  --bundle eval_bundle.npz --bundle_meta bundle_meta.json
python3 profile_energy_fpga.py --measure_idle --output_dir results \
  --summary_json power.json --raw_log power_raw.csv \
  --cmd "python3 vart_run_fpga.py --xmodel model.xmodel --model_info model_info.json \
         --bundle eval_bundle.npz --bundle_meta bundle_meta.json --quiet"
```

`vai_trace_fpga.py` runs the application under `vaitrace` for a fixed duration; the column `HW_RT` of its summary is the DPU time per run. `profile_energy_fpga.py` samples SOM power while it repeats the command, measures idle power first (`--measure_idle`) and reports total and idle-subtracted energy per counted input. The sensor excludes the carrier and USB camera; the dynamic result is not isolated DPU energy. Preserve sample counts, units and the measured interval. Compare variants with identical commands and board conditions.

Checkpoint E3 — secondary measurements

For each deployed model, record host and KV260 accuracy, VART and system FPS, `xdputil` FPS with 1 and 4 threads, the DPU time from the trace, and the energy per input if measured. Why

can PTQ and QAT differ in accuracy but not in throughput? What explains the gap between raw DPU FPS and application FPS for such a small network?

Troubleshooting

Symptom	First checks
SSH or SCP fails	`getent hosts ${BOARD#*@}`: an unknown name needs the IP address (serial console, Step E3); Ethernet cable; account `root`
`REMOTE HOST IDENTIFICATION HAS CHANGED`	the board was re-imaged (new key under the same name): `ssh-keygen -R ${BOARD#*@}`
`import xir, vart` fails	`source setup_env.sh`; the board image must provide the Vitis AI runtime (VART)
compiler puts layers on the CPU and reports batch 128	export again with `04d` (batch 1); `compiler_diagnosis.json` names the batch as the cause, the constant and reshape warnings follow from it
CPU operations without a batch warning	read the first direct assignment and the inspector report: unsupported operation, shape or DPU parameter. Do not use the DPU-only application until CPU computation is absent; only the checked terminal output conversion is permitted
no DPU subgraph found	`partition_report.txt`; check that the copied file is the compiler output
firmware/XCLBIN error	`XLNX_VART_FIRMWARE` and the B4096 platform file (`07_check_board.sh`)
input shape mismatch	print the XMODEL input shape; the NCHW/NHWC conversion must match it
KV260 accuracy far from host	/255 preprocessing, layout, fixed-point conversion, bundle and model checksums
raw DPU FPS high, application FPS lower	preprocessing, buffer preparation, Python and runtime overhead, output handling

The deployment pattern of the next labs stays the same: quantized graph → compilation for the target → XIR graph → VART runner → measurement of the application.

6 Part F — Final task

Final task — one controlled experiment

Choose one quantization question that the guided runs did not answer: amount of calibration data, calibration class bias, calibration seed, number of fast-fine-tuning images, QAT epoch budget, ...

`Variables`	Method	What changes
`CALIB_SAMPLES, CALIB_SEED`	PTQ	amount or selection of calibration images
`CALIB_PROFILE (representative, class0, dark), MISMATCH_SCALE`	PTQ	representativeness of calibration images
`FAST_FINETUNE, FT_SAMPLES`	PTQ recovery	layer-wise unlabelled weight adjustment
`QAT_EPOCHS, QAT_LR, SEED`	QAT	training budget and optimization

1. State a falsifiable prediction before running anything.
2. Create a new, descriptive QTAG and change exactly one factor relative to a named baseline.
3. Record the unchanged controls: source checkpoint, test set, preprocessing, all other settings.

4. Run the export script and rebuild `summary/results.csv`. The variables are listed by `--help`: `04b_ptq_calibrate.sh`, `04_ptq_export.sh`, `04_qat_export.sh`.
5. Decide which method you would deploy under the 0.5-point tolerance and explain the effort–accuracy trade-off.

Report the command, the new `metrics.json`, `summary/results.csv`, and five to eight sentences with the prediction, controlled variable, result, recommendation and one limitation.

Optional extensions

- `03_train_float.sh`: other architectures, widths and budgets, each under its own TAG, through the whole flow.
- `03c`, `03d`, `03e` with another bit width, channel or layer.
- `03b_make_stress_model.sh`: a functionally equivalent model with strongly unequal channel ranges (per-channel rescaling compensated in the next layer), to stress per-tensor quantization.
- Strategy files with `04_ptq_export.sh`: `QUANT_CONFIG=configs/dpu_corrections.json` or `configs/dpu_no_corrections.json` turns cross-layer equalization and bias correction on or off (`configs/README.md`); re-run the inspector and the compiler.
- `src/tp02_xmodel_viz.py`: Graphviz drawing of the CPU/DPU partition of a compiled XMODEL, `python3 src/tp02_xmodel_viz.py --xmodel <compiled xmodel>` (`--render_png` needs `dot`).
- `TAG=${TAG} bash scripts/04_run_guided_matrix.sh`: the four guided variants in one command.
- `07_deploy_and_evaluate.sh`: copy, run and fetch one variant (for later labs).
- `docs/vitis_ai_api_quick_reference.md`: the PTQ, fast fine-tuning and QAT calls on one page.

References: Vitis AI 3.5 quantization workflow https://xilinx.github.io/Vitis-AI/3.5/html/docs/workflow-model-development.html; `torch_quantizer` https://docs.amd.com/r/3.5-English/ug1414-vitis-ai/class-torch_quantizer; quantization strategy and DPU restrictions https://docs.amd.com/r/3.0-English/ug1414-vitis-ai/Configuration-of-Quantization-Strategy; hardware-aware strategy https://docs.amd.com/r/3.5-English/ug1414-vitis-ai/Hardware-Aware-Quantization-Strategy; QAT https://docs.amd.com/r/3.5-English/ug1414-vitis-ai/vai_q_pytorch-QAT; MPSoC/KV260 quick start, with the batch-1 export https://xilinx.github.io/Vitis-AI/3.5/html/docs/quickstart/mpsoc.html; `resnet18_quant.py` https://github.com/Xilinx/Vitis-AI/blob/v3.5/src/vai_quantizer/vai_q_pytorch/example/resnet18_quant.py.

A Folders and scripts

Folder	Content
`arch.json`	fingerprint of the course KV260 DPU (compiler target)
`configs/`	optional Vitis AI strategy files
`data/`	CIFAR-10 (downloaded on first use)
`reference/`	supplied float checkpoint and its metrics
`src/, scripts/`	programs and the commands that run them
`docs/`	one-page Vitis AI API reference
`out/runs/<TAG>/`	`float/`, `concepts/`, `inspection/`, `quant/<QTAG>/`, `deploy/<QTAG>/`, `board_bundle/`, `board_results/<QTAG>/`, `logs/`, `summary/`

Script	Role
`03_use_reference_model.sh`	copy and measure the supplied checkpoint under the TAG
`03_train_float.sh`	train an MLP or CNN source model under the TAG
`03b_describe_float_model.sh`	layers, shapes, parameters, selectable weights and activations
`03b_make_stress_model.sh`	equivalent model with unequal channel ranges (extension)
`03c / 03d / 03e`	Part B: one tensor, one weight channel, one activation layer
`04a_inspect_dpu.sh`	Vitis AI inspector for the DPU
`04b / 04c / 04d`	PTQ calibration, host test, batch-1 export
`04e_show_quant_params.sh`	bit width, fixed-point position, step and range of each tensor
`04_ptq_export.sh`, `04_qat_export.sh`	PTQ shortcut (04b, 04c, 04d); QAT and its export
`04_run_guided_matrix.sh`	`ptq_ref`, `ptq_mismatch`, `ptq_fast_ft`, `qat_base` in one command
`05_compile_kv260.sh`, `05_compile_ptq_qat_pair.sh`	compile one variant, or the PTQ/QAT pair, with checks
`06_prepare_eval_bundle.sh`	board evaluation bundle
`07_check_board.sh`	SSH, XIR/VART, `xdputil`, firmware
`07_copy_to_board.sh`, `07_copy_ptq_qat_pair.sh`	copy one variant, or the pair, under the remote TAG
`07_evaluate_ptq_qat_pair.sh`, `07_deploy_and_evaluate.sh`	run the pair, or copy, run and fetch one variant
`08_collect_board_results.sh`	fetch the board results of one variant
`09_compare_results.sh`, `09_compare_ptq_qat_board.sh`	result table of every variant; paired KV260 table