TP07 – MNIST CNN on FPGA

28×28 MNIST | Conv3×3 + ReLU + MaxPool + FC | 7-hour lab

Goal. Use the convolution and pooling ideas from TP06 to build a complete MNIST classifier. Implement the FC layer, connect the HLS stages, check the integer results, then compare two hardware designs. Finish with handwritten-digit capture from a USB camera.

1 The CNN

28×28×1 image → **Conv 3×3 (4 filters)** → 26×26×4 → **ReLU + 2×2 MaxPool** → 13×13×4 → **FC 676→10** → 10 signed INT32 logits

Stage	Parameters	MACs / image	Responsibility
Conv3×3, 1→4	40	24,336	Given, review TP06 techniques
Pool2×2	0	0	Given, inspect raster order
FC 676→10	6,770	6,760	**Implement, test and optimize**
Total	**6,810**	**31,096**	Full CNN

The initial HLS top-level interface is provided: one 32-bit AXI-Stream word per grayscale pixel; 10 signed INT32 output words per image; **TLAST only on the final output word of the entire `numReps` call**, not after each image. The top also exposes `numReps` through AXI-Lite. You must connect its processes. The supplied Tcl and host scripts create the project and (after successful IP export) the Vivado block design. *They do not write your FC computation or make architectural choices for you.*

2 Train and quantize

In TP06 the weights were **synthetic INT8 test values**, a fixed pattern generated by the testbench or the weight header. That was sufficient to test convolution hardware; it did not demonstrate a trained classifier. In TP07 the steps have distinct purposes:

1. **Training (float32):** PyTorch learns convolution and FC weights from real MNIST images using labels. The model accepts grayscale pixel values in $[0, 1]$; the held-out test split is not used for training.
2. **Calibration (training data):** The exporter selects integer fixed-point positions for weights and intermediate activations. The test split is **not** used for calibration.
3. **Quantization:** A learned weight is encoded approximately as $w_q = \mathrm{clip}_{[-128,127]}\big(\mathrm{round}(2^{f_w} w)\big)$. Likewise an activation uses position f_x. Integer convolution/FC multiply `int8` operands and accumulate into `int32`. Bias is aligned as $b_q \ll s_b$; convolution activations are rounded, shifted and saturated to signed INT8. **The last FC produces raw INT32 logits**, not 10 INT8 probabilities.
4. **Hardware export:** `tools/export_cnn.py` writes the *trained and quantized* arrays into `compact_mnist_net.h`, with scaling shifts, and generates exact integer reference outputs. This generated header is **not** a student implementation and should not be edited by hand.

Do not compare the HLS output bit-for-bit with float32 PyTorch. Compare it with the exported *integer* reference. Report both classification accuracy and integer correctness. Accuracy is a statistical metric; bit-exactness is a numerical implementation contract.

3 Build the classifier

3.1 Train and export

Work from the `TP07_cnn_fpga_challenge/` folder. Reuse the same Vitis AI PyTorch container from TP02; do **not** install a new Conda environment and do not try to run Vitis HLS inside the container.

```
# HOST
bash tools/02_enter_container.sh
# CONTAINER (same repository directory is mounted)
bash tools/check_env.sh
bash tools/prepare_mnist.sh
cat data/compact_mnist/export_report.json
python3 tools/quantization_explainer.py
exit
```

`prepare_mnist.sh` trains for five epochs by default, calibrates on 1,000 *training* images, exports signed integer weights/reference vectors, and evaluates 1,000 held-out test images. It stops on failure. To increase epochs, set `TP07_EPOCHS=8` before running it. Model quality and training time depend on the teaching host; the exported accuracy is measured, not a guaranteed number.

Observe and record: float accuracy, INT8 accuracy, quantization gap, parameter count, MAC count and shift values `conv1.s`, `fc1.sb`. The exported classifier uses the model that was trained *in this run*; do not reuse stale headers from other checkpoints.

3.2 Read the HLS code

Locate these files:

- `hls/compact_mnist/compact_mnist_layers.h`: provided Conv + Pool, incomplete FC.
- `hls/compact_mnist/compact_mnist_top.cpp`: AXI interface, FIFO declarations, incomplete connection calls.
- `compact_mnist_net.h`: *generated* parameters and fixed-point information.
- `hls/compact_mnist/tb_compact_mnist.cpp`: independent stage and whole-network checks.

Trace the total number of stream elements: **784 input pixels, 676 convolution pixels (4 signed lanes each), 169 pooled pixels, 10 output logits**, per image. Why should this not become a sequence of intermediate DDR transactions?

Examples. TP06 Stage 05 (`05_conv_relu_pool2x2_unpacked_student`) illustrates Conv+Pool; Stage 06 (`06_conv_relu_pool_stream_dataflow_student`, an optional part of TP06: read it now if you skipped it) illustrates FIFOs and `DATAFLOW`. TP07 has different AXI interfaces: study its given `conv1_stage`, `maxpool1_stage`, `read_stage` and `write_stage` before connecting them.

3.3 FC layer

In `compact_mnist_layers.h`, fill the TODOs of `fc1_stage`. The provided code already consumes 169 pixel words into `feature[p][c]`. For output class o, the accumulator is

$$A_o = (b_{q,o} \ll s_b) + \sum_{p=0}^{168} \sum_{c=0}^{3} x_q[p, c]\, w_q[o,\ c \cdot 169 + p].$$

Flattening. The pooled output has shape $13 \times 13 \times 4$. The FC sees 676 features, with PyTorch (NCHW) index $i = c \cdot 169 + p$. The hardware receives pixel p with four channel lanes in one word; it can access `feature[p][c]` directly without a separate flattening IP or a DDR transfer.

Prediction and softmax. The FC emits ten signed INT32 *logits*. The predicted digit is the index of the largest logit, $\arg\max_o A_o$. **No softmax is needed on the FPGA**: softmax does not change the winning class and would add exponentials and divisions. If probabilities are required, they can be estimated in software after applying the correct output scale; raw integer logits or their gap must not be called probabilities. Do not requantize the final logits to INT8.

FC implementation ideas. There is more than one useful organization:

- *Buffered:* keep the pooled features on chip; compute the dot product for each class in turn.
- *Streaming:* initialize one accumulator per class and update it as each pooled word arrives. This may avoid storing the entire feature map.
- *Tiled:* compute a few classes in parallel. Consider how many weight values must be read per cycle and how the final group is handled.

Whichever approach you choose, include each input feature once per class, align the bias to the accumulator scale, and write ten signed INT32 logits in class order. Check the channel-major weight index before deciding on a loop order. Different loop and buffer organizations are welcome; compare their hardware costs.

Fast functional loop, on HOST:

```
bash tools/test_cpp.sh
```

This compiles the *actual HLS C++ source* against a tiny host-only type shim. It checks conv, pool, FC and then the whole classifier. It is intentionally **not** a substitute for Vitis HLS. While the FC skeleton is incomplete, the FC check should report `FAIL`; fix it before proceeding. The supplied stage checks are debugging aids, not a prescribed implementation. If you merge or rename stages, test the top-level AXI interface instead:

```
bash tools/test_cpp.sh top-only
bash tools/run_hls.sh compact_mnist csim top-only
```

The native test supports only a subset of the Xilinx HLS C++ API; code using another valid HLS stream operation may not compile with that shim. Use real Vitis HLS C simulation in that case. For a valid custom accelerator, numerical output and the external interface matter more than internal function names.

3.4 Connect the layers

In `compact_mnist_top.cpp`, the four internal FIFOs are already declared. Enable `DATAFLOW`. Call `read_stage`, `conv1_stage` and `maxpool1_stage`, followed by `fc1_stage` and `write_stage`. Pass the same `numReps` to every function. TP06 Stage 06 shows the producer–FIFO–consumer pattern; `compact_mnist_io.h` shows the AXI input and output adapters. The final output word must assert TLAST.

Check the stream counts: 784 input words → 676 convolution words → 169 pooled words → 10 logits, for each image. If a stage reads or writes the wrong count, the pipeline can block even when the arithmetic is correct.

```
# HOST: Vitis HLS is installed on the lab machine
bash tools/test_cpp.sh
bash tools/run_hls.sh compact_mnist csim
```

Pass criterion: whole-network logits match the INT8 reference, including consecutive images. The provided stage tests should also pass when you retain the given stage decomposition. If the native check passes but Vitis HLS C simulation fails, inspect the HLS log, not the CNN accuracy.

4 Optimize and test

4.1 Find the bottleneck

Use the analytic estimate, noting that it *does not* establish timing closure or a real initiation interval:

```
python3 tools/cost_model.py compact_mnist
```

The provided convolution is a functional baseline with no explicit pixel-loop pipeline. The cost model assumes one modeled loop iteration per cycle; it can therefore be much more optimistic than this baseline. Increasing `PAR` alone does not guarantee a lower interval. Inspect the convolution schedule, memory ports and achieved II; you may optimize the provided stages as well as your FC.

Questions: At one input word per cycle, what is the minimum interval for 784 pixels? Which stage limits throughput at `PAR=1`? Will accelerating a different stage help?

4.2 Two hardware designs

Use the supplied tool to update configuration macros and matching `ARRAY_PARTITION` factors, then read the changed pragma lines. It supports the provided loop/pragma structure and factors that divide the output count. A different implementation may use other factors and handle a partial final group; in that case edit its HLS directives directly, verify correctness and compare synthesis reports. The **optimization choice and explanation are yours**.

```
# Baseline: compare the measured HLS result with the cost model
bash tools/run_hls.sh compact_mnist synth
python3 tools/parse_hls_report.py compact_mnist
bash tools/snapshot_report.sh baseline

# Example candidate: adjust one or both parallelism factors
python3 tools/set_par.py compact_mnist fc1=2
bash tools/test_cpp.sh
bash tools/run_hls.sh compact_mnist csim
bash tools/run_hls.sh compact_mnist synth
bash tools/snapshot_report.sh optimized
```

Variable `numReps` loops can make overall HLS latency and interval appear as `?`. Record `N/A (variable trip count)` in the table, retain the fixed-count loop rows and compare measured batch timing when available. An unknown interval cannot establish images/s, and a cost-model estimate must remain labelled as an estimate. `fc1=2` is an example configuration, not a promised speedup when convolution is the bottleneck.

The script regenerates the project for each synthesis. Use `bash tools/snapshot_report.sh baseline` (or another short label) immediately after each run to preserve the measured report and its configuration. Changing a directive can change banking, DSP/LUT use, ROM/BRAM mapping, II, fmax and throughput, but not the expected logits.

4.3 FPGA deployment (optional)

On Ubuntu, before HTTPS downloads, check `date -Is` and `timedatectl status`. If the date is incorrect or time is unsynchronized, contact the instructor. Keep certificate verification enabled; offline HLS and FPGA work may continue without NTP.

The Tcl flow creates the Zynq MPSoC, AXI DMA, control/data interconnect, block-design wrapper, bitstream and HWH. Trace the **PS control path, DMA stream paths, clocks and resets**; manual wiring is unnecessary. The board runs the Ubuntu/PYNQ image, as in the board part of TP06: the lab scripts log in as `ubuntu`, and `BOARD` is the network name of your board, as in Lab 2.

```
# HOST: only after IP export has been validated
bash tools/run_hls.sh compact_mnist export
JOBS=4 bash vivado/build_overlay.sh compact_mnist
export BOARD=maquette-07    # the name of your board, or its IP address
bash host/deploy_to_kv260.sh compact_mnist
```

The example limits parallel Vivado jobs on 16 GB teaching PCs. Close other heavy jobs; reduce `JOBS` further if memory is limited. No fallback IP is bundled: if export fails, preserve its log and ask the instructor. Any substitute IP must include its matching weights and reference vectors.

On the KV260, complete the PYNQ preflight and numerical check shown under *Run FPGA inference* before benchmarking or using the camera.

Hardware consistency: the FPGA overlay contains weights compiled from a specific quantized checkpoint. The bitstream, hardware metadata and reference vectors must come from the same build. Record FPGA measurements only after verifying the actual overlay.

5 USB camera

A UVC USB webcam supplies an image to the KV260's ARM processor. The FPGA remains a single fused Conv–Pool–FC accelerator: only the *28×28 quantized input* and ten logits cross the DMA boundary. The camera does not require an additional video-processing IP.

5.1 Capture a digit

Draw a dark digit on plain light paper and position it inside the central square of the camera frame. The camera utility crops the central ROI, converts it to grayscale, selects foreground polarity, isolates the digit, scales it to a 20-pixel box, centers its intensity within a 28×28 image and saves that image for inspection. Unlike a pristine MNIST digit, a photograph may include shadows, off-center writing and background texture.

```
# HOST: copy the small camera utility (works without an FPGA overlay)
export BOARD=maquette-07  # the name of your board, or its IP address
bash host/deploy_camera_only.sh
# KV260: Ubuntu, from /home/ubuntu/c5/TP07
./check_usb_camera.sh --source /dev/video0
# Inspect camera_preview/roi_frame.jpg and mnist28.png
```

To inspect the pictures on the workstation, run `bash host/fetch_camera_preview.sh` on the HOST after the camera check. The files appear in `results/camera_preview/`. The camera check automatically saves its preview files and does not load PYNQ or the FPGA. If the result is blank or clipped, change the paper position or pass `--roi 0.45`; use `--invert yes` for dark ink on light paper if automatic polarity is inadequate. The exported model expects *bright strokes on a dark background*, the same spatial orientation and 28×28 input dimensions as during training.

5.2 Run FPGA inference

Once the matching Vivado bitstream and HWH have been deployed, the same preprocessed 28×28 image is converted to the model's signed INT8 input and passed to the fused network using the existing AXI DMA. No changes to the HLS CNN are necessary.

```
# HOST: deploy the verified overlay, app and reference vectors
bash host/deploy_to_kv260.sh compact_mnist
# KV260
cd /home/ubuntu/c5/TP07
./preflight_pynq.sh compact_mnist
./run_app.sh compact_mnist --mode check
./run_app.sh compact_mnist --mode camera --camera-frames 30 --json
# A different UVC device: --camera-source /dev/video1
# Graphical desktop only: add --camera-preview --camera-frames 0
# From a saved camera frame: --camera-image photo.jpg
```

The default mode is headless and works through SSH. The marked frame, the actual 28×28 grayscale input and a JSON timing report are saved under `compact_mnist/results/`. These can be retrieved using `host/fetch_results_from_kv260.sh`. A graphical preview is optional and requires an accessible desktop display.

Measure and explain: compare camera acquisition time, preprocessing on the ARM cores, DMA plus FPGA time and overall camera throughput. A faster HLS interval may not improve camera FPS if USB acquisition or preprocessing dominates. A classifier's top-two raw-logit gap is *not* a calibrated prediction probability. Try a correctly centered digit and an off-center or poorly lit digit; inspect the 28×28 input before diagnosing the accelerator.

The camera preprocessing utility can be tested before the FPGA overlay exists. Actual FPGA classification requires a bitstream and HWH built from the same learned and quantized weights as the reference. An ROI preview alone is not an FPGA inference result.

6 Results

Submit the modified FC and DATAFLOW HLS sources, passing correctness checks, camera ROI preview (`mnist28.png`), and a one-page comparison. Distinguish *measured* resources/intervals from cost-model predictions; FPS requires a measurement or a transparently stated estimate.

Variant	INT8 pass?	HLS interval	Area (DSP/-LUT/BRAM)	Interpretation
Baseline PAR=1				
Optimized PAR				

Address: (1) how trained weights are quantized, (2) FC flattening order, (3) the throughput bottleneck, (4) area versus parallelism, and (5) how camera preprocessing and I/O affect complete-system performance. Separate measured FPGA data from HLS estimates.

If a command fails, see the troubleshooting notes in `README.md`. Check the exact error before restarting synthesis.