Embedded Artificial Intelligence

SETI Master

Lab 6 — Frugal Embedded AI with Vitis HLS

One Conv2D layer in hardware: parallelism,
the frugal choice and operator fusion

Author: Alaa Eddine Mazouz

Academic year: 2026–2027

Version: 46

Date: 31 October 2026

Télécom Paris

Purpose of the lab

One quantized CNN operator, an int8 2D convolution (Conv2D), synthesized with Vitis HLS for the KV260 at 100 MHz. You see how line buffers reuse data and how unrolling and pipelining trade resources for cycles, and you answer one engineering question:

What is the smallest architecture that reaches the required frame rate?

Then you write HLS yourself: Conv2D, ReLU and MaxPool fused in one accelerator. The ideas (what HLS builds, pipelining, unrolling, line buffers, fusion, DATAFLOW) are summarized in the short HLS/CNN guide of Labs 6 and 7. The reference layer is fixed unless stated otherwise:

$$128 \times 128 \times 3 \xrightarrow{3\times3,\ 64\ \textit{filters}} 126 \times 126 \times 64.$$

	Content	Your work	Planning estimate
Part A — the lab (required, one session)			
A1	workflow; the layer; line buffers (Sections 1–3)	Exercises 1–3	45–60 min
A2	one layer, five architectures (Section 4)	Exercise 4	30–40 min
A3	the frugal choice: the `OC_PAR` sweep (Section 4)	Exercise 5	15–20 min
A4	your code: Conv2D, ReLU and MaxPool fused (Section 5)	TODO A, B; Ex. 6–7	45–60 min
A5	towards Lab 7 (Section 6)	read	5 min
Part B — optional extension			
B1	Stage 06: DATAFLOW and AXI-Stream (Section 7)	TODO 1, 2; Ex. 8–10	45–60 min
B2	Version 04 on the KV260: IP, Vivado, PYNQ (Section 8)	Exercises 11–13	2–3 h

Part A is the lab: the syntheses run while you read and answer, so it fits one session. Part B is for groups that finish early, or for a later visit; B1 is the best preparation for Lab 7, where you connect a whole network with `DATAFLOW`. The estimates are for planning only: tool times depend on the machine.

Part A — The lab

1 Workflow and C simulation

The lab folder holds the five provided versions, your two stages and the tools:

```
TP06_hls_conv2d/
|-- hls/
|   |-- common/                       conv2d_config.h (the layer), tb_conv2d_top.cpp (testbench)
|   |-- 00_naive_serial_1mul/ ... 04_linefifo_oc_parallel_packed/      provided versions
|   |-- 05_conv_relu_pool2x2_unpacked_student/                          your code (Part A)
|   |-- 06_conv_relu_pool_stream_dataflow_student/                      your code (Part B)
|   |-- run_hls_stage.tcl             HLS flow shared by all stages
|   `-- setup_csim_env.sh             linker fix for C simulation (lab machines)
|-- tools/                            lab_env.sh, configure_layer.py, parse_hls_report.py
|-- DSE_sweep/                        the OC_PAR sweep
|-- exported_ip/, host/, kv260/       IP export and the board (Part B)
`-- README.md
```

Each version folder holds `conv2d_top.h`, `conv2d_top.cpp` and `run_hls.tcl`; the top function is always `conv2d_top`. Versions 00–04 keep the same memory-mapped interface while the hardware inside changes; Stage 05 changes the output shape (pooling), Stage 06 the interface (AXI-Stream).

Each listing starts with where it runs: `# host` is a terminal on the lab machine, where Vitis HLS and Vivado are installed (no container in this lab). Go to the lab folder, remember it, and load

the environment (`$VITIS_HLS_BIN`, `$VIVADO_BIN`; use them instead of hard-coded tool paths). Type `bash` and press Enter as a separate action in each new host terminal. Wait for its prompt before pasting the following block:

```
# host (after entering Bash separately)
cd TP06_hls_conv2d
export TP06=$(pwd)
source tools/lab_env.sh
source hls/setup_csim_env.sh    # C-simulation linker fix of the lab machines
```

The teaching login shell is zsh, while these sourced helpers use Bash syntax. Paths below are relative to `$TP06`. One version by hand: go to its folder and run

```
$VITIS_HLS_BIN -f run_hls.tcl      # C simulation, then C synthesis
```

C simulation answers *does this C++ compute the intended integer convolution?* The testbench `hls/common/tb_conv2d_top.cpp` uses deterministic int8 inputs, weights and biases, computes an integer golden reference and compares every output value:

```
PASS: conv2d_top matches integer golden reference.
```

A variable `numReps` loop in Stage 06 can make an overall latency or interval appear as `?`; record it as unavailable, then inspect fixed-count inner-loop rows or measure a known batch. Do not replace an unknown with a cost-model prediction. C synthesis answers *how much hardware and how many cycles does it take?* The report is `hls/<stage>/prj_<stage>/solution1/syn/report/conv2d_top_csynth.rpt`. The same switches work in every stage:

```
RUN_CSIM=0     # no C simulation
RUN_CSYNTH=0   # C simulation only (Stages 05 and 06 do this by default; RUN_CSYNTH=1 adds the synthesis)
RUN_COSIM=1    # C/RTL co-simulation      (Part B only)
RUN_EXPORT=1   # export a Vivado IP       (Part B only)
```

Without `source hls/setup_csim_env.sh` (above), the C simulation fails at link time on the lab machines (`ld: cannot find crt1.o`): run it once in every new shell, from `$TP06`.

Start the five syntheses now, then read

The five versions take a while to synthesize. Open a second terminal, load the environment as above, and run them one after the other while you work on Sections 2–4 in the first terminal:

```
# host (second terminal, after entering Bash separately)
cd TP06_hls_conv2d && export TP06=$(pwd)
source tools/lab_env.sh && source hls/setup_csim_env.sh
for v in 00_naive_serial_1mul 01_linefifo_row_kmul 02_linefifo_kernel_kxky_parallel \
         03_linefifo_dot_parallel 04_linefifo_oc_parallel_packed; do
  (cd $TP06/hls/$v && $VITIS_HLS_BIN -f run_hls.tcl > run.log 2>&1) \
    && echo "$v: done" || echo "$v: FAILED, see hls/$v/run.log"
done
```

Each version writes its report in its own folder. Do not start the `OC_PAR` sweep (Section 4) before this loop has finished: the sweep reconfigures `hls/common/conv2d_config.h`, which the versions read.

2 The Conv2D layer

The layer is set in `hls/common/conv2d_config.h`:

```
#define IN_H    128
#define IN_W    128
#define IN_CH   3
#define K       3
#define OUT_CH  64
#define OC_PAR  8
```

It is a valid convolution, stride 1, no padding: each output channel comes from one $3 \times 3 \times 3$ filter, and the layer has 64 filters (Figure 1).

With $N = \text{OUT_H} \times \text{OUT_W}$ output positions per channel and a dot product of length $D = \text{IN_CH} \times K^2$:

$$\text{OUT_H} = \text{IN_H} - K + 1, \qquad \text{Weights} = \text{OUT_CH} \times D, \qquad \text{MACs/frame} = N \times \text{OUT_CH} \times D,$$

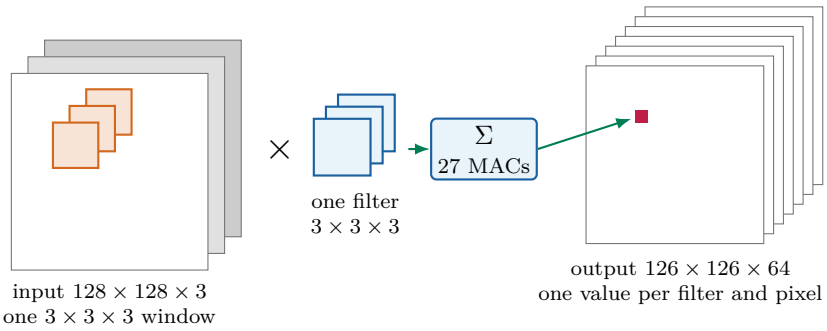


Figure 1: One output value: a $3 \times 3 \times 3$ window of the input times one filter, 27 multiply-accumulates (MACs). The 64 filters give the 64 output channels.

$$\text{GMAC/s} = \frac{\text{MACs/frame} \times \text{FPS}}{10^9}, \qquad \text{GOPS} = 2 \times \text{GMAC/s}.$$

Bias additions are not counted as MACs: they are applied after the dot product. For the reference layer, OUT_H = OUT_W = 126, $N = 15{,}876$, $D = 27$, 1,728 weights and 64 biases.

Exercise 1 — The Conv2D workload

For the reference layer: how many MACs give one output value, how many output values one filter produces, how many MACs one frame needs, and which compute rate (GMAC/s and GOPS) 60 FPS requires. Then keep IN_CH = 3, $K = 3$, OUT_CH = 64 and change only the image size:

Input size	Output size	N	D	MACs/frame
64×64				
128×128				
224×224				

Which quantity grows with the image, and which stays fixed?

3 From naive convolution to line buffers

The MAC count is one cost; the activations that must be read to feed those MACs are another. A naive implementation reads a complete $K \times K \times$ IN_CH window for every output value and every output channel:

$$R_{\text{naive}} = N \times \text{OUT_CH} \times D \quad \text{activation reads per frame.}$$

But neighbouring windows overlap: when the window moves by one pixel, most of its input values are reused (Figure 2).

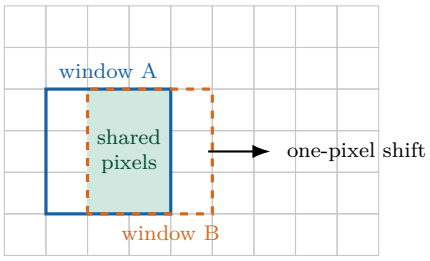


Figure 2: Adjacent convolution windows share most of their input data.

A streaming implementation reads the input once, in raster order, and keeps on chip what later windows still need:

$$R_{\text{stream}} = \text{IN_H} \times \text{IN_W} \times \text{IN_CH}.$$

Both perform the same MACs; they differ in how the activations reach the multipliers.

Exercise 2 — Activation traffic

For the reference layer, compute R_{naive}, R_{stream} and the reduction factor $R_{\text{naive}}/R_{\text{stream}}$. Why can two implementations with the same MAC count have very different activation traffic?

To form a $K \times K$ window at each new pixel, the accelerator keeps the $K-1$ previous rows of each input channel in **line buffers**, FIFOs of one row (`IN_W` words), and the current window in registers (Figure 3):

$$N_{\text{FIFO}} = \text{IN_CH} \times (K-1), \qquad \text{FIFO words} = \text{IN_CH} \times (K-1) \times \text{IN_W}.$$

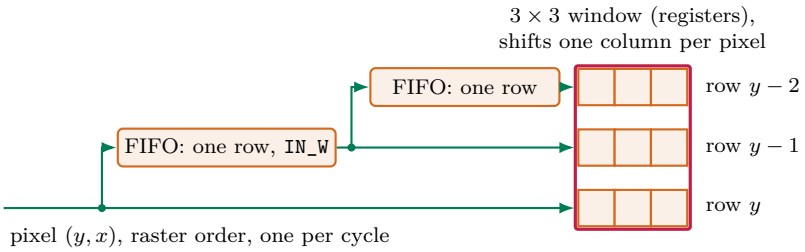


Figure 3: The line buffers of one input channel for a 3×3 window: $K-1=2$ FIFOs of one row. Every input channel has its own copy: IN_CH $\times (K-1)$ FIFOs, and a window of IN_CH $\times K \times K$ values.

In the line-buffer versions (01–04), the structure is:

```
hls::stream<data_t> linebuf[IN_CH][K - 1];
#pragma HLS ARRAY_PARTITION variable=linebuf complete dim=0
#pragma HLS STREAM variable=linebuf depth=<2*IN_W> type=fifo

for (int y = 0; y < IN_H; y++) {
  for (int x = 0; x < IN_W; x++) {
    ... update the FIFOs and the window ...
    if (y >= K - 1 && x >= K - 1) {
      ... compute one output position ...
    }
  }
}
```

Exercise 3 — FIFO storage and valid windows

Complete the table, in activation words:

Case	IN_CH	K	IN_W	N_{FIFO}	FIFO words
Reference RGB 3×3	3	3	128		
Larger width	3	3	224		
More channels	8	3	128		
Larger kernel	3	5	128		

At which input coordinate does the first valid window of a 3×3 convolution appear? If IN_CH doubles, what doubles in the FIFO structure? (The code declares each FIFO with depth $2 \cdot$ IN_W, slack for scheduling: the FIFO storage of the report is about twice the logical value of this table.)

4 One layer, five architectures

The pragmas of this lab

The C++ describes the computation; pragmas tell Vitis HLS what hardware to build.

Pragma	Effect
`UNROLL (factor=n)`	duplicates the body of the next loop: its iterations run in parallel hardware
`PIPELINE II=1`	starts a new iteration of the loop every cycle; check in the report that `II=1` was reached
`ARRAY_PARTITION`	splits an array into several memories or registers, so that unrolled hardware can read many elements in the same cycle
`BIND_OP ... impl=DSP`	implements a multiplication in a DSP block
`STREAM ... depth=n`	sets the depth of an `hls::stream` FIFO

Unrolling only helps if the data arrive in parallel: the more parallel versions partition the window and the weights. A loop that misses `II=1` usually waits for a memory port, a dependence between iterations, or a FIFO. The FIFO depths are written by `tools/configure_layer.py` (`LINEBUF_DEPTH` $= 2 \cdot$ IN_W, `POOL_LINEBUF_DEPTH` $= 2 \cdot$ OUT_W): Vitis HLS 2021.2 wants literal numbers in pragmas, so the tool, not you, edits them.

Versions 00–04

The same layer is implemented five times; only the hardware parallelism changes (Figure 4). All versions are synthesized for the KV260 (`xck26-sfvc784-2LV-c`) with a 10 ns clock (100 MHz). The report gives the latency of one call, one frame, in cycles:

$$\text{FPS}_{\text{est}} = \frac{100 \times 10^6}{\text{max latency (cycles)}}, \qquad 60 \text{ FPS} \iff \text{max latency} \leq 1{,}666{,}666 \text{ cycles}.$$

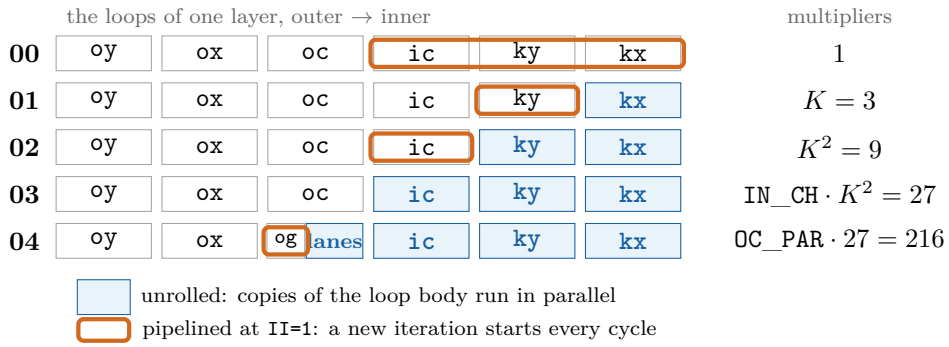


Figure 4: The loops of the Conv2D nest that each version unrolls (blue) and pipelines (orange frame). Version 00 pipelines its flattened $ic \cdot ky \cdot kx$ loop through one multiplier; Version 04 splits the output channels into groups (og) of `OC_PAR` lanes and pipelines the groups. Source: `hls/0*/conv2d_top.cpp`.

- **00** (`00_naive_serial_1mul`): the mathematical definition, no line buffer, one multiplier forced by `#pragma HLS ALLOCATION operation instances=mul limit=1`: the controlled baseline.
- **01**: line FIFOs and a window; only `kx` is unrolled: a K-multiplier row engine, `ky` and `ic` still sequential.
- **02**: `ky` and `kx` unrolled: K^2 multipliers for one input channel; `ic` is time-multiplexed.
- **03**: the whole dot product unrolled (IN_CH $\times K^2 = 27$ products); the window and the weights are partitioned so that all operands arrive together. No loop is pipelined: each output channel waits for the full latency of the dot product.
- **04**: `OC_PAR` copies of the dot product (lanes) for the same window, one output channel each, and the output-group loop pipelined at `II=1`; the lanes are packed into one output word of $8 \cdot$ `OC_PAR` bits (Figure 5).

Automatic loop pipelining is disabled (`config_compile -pipeline_loops 0` in `hls/run_hls_stage.tcl`): only the framed loops of Figure 4 are pipelined, so part of the 03→04 gain, visible even with `OC_PAR=1`, comes from the pipeline and not from the lanes. `configure_layer.py` also prints a first-order cycle model of each version, to compare with the reports.

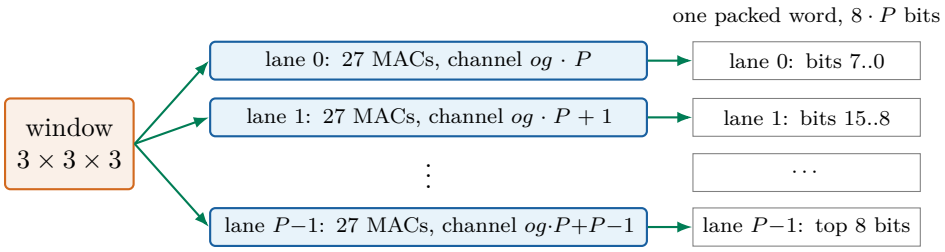


Figure 5: Version 04: $P =$ `OC_PAR` lanes evaluate P output channels of the same window in parallel; their results are packed into one output word (`out_vec_t`). Output group og covers channels $og \cdot P$ to $og \cdot P + P - 1$.

Exercise 4 — Five architectures, one table

When the loop of Section 1 has finished, complete the table from the five reports:

Version	Expected multipliers	Reported DSPs	LUTs	Latency (cycles)	Estimated FPS
00 serial baseline	1				
01 row parallel	K				
02 spatial parallel	K^2				
03 dot-product parallel	IN_CH K^2				
04 output-channel lanes	`OC_PAR` IN_CH K^2				

1. Why is Version 00 still a useful baseline?
2. What did the line FIFOs of Version 01 improve, and what did the `kx` unroll improve?
3. Why does Version 02 use about 9 multipliers and not 27? Which loop is still time-multiplexed?
4. What does the partitioning of Version 03 cost, and why is it needed?
5. For `OC_PAR=2`: the number of output groups and of multipliers. Why is the fastest design not automatically the best one for a 60 FPS target?

The DSP count is read from the report; it follows the multiplier structure, but scheduling, binding and control logic make small differences.

The frugal choice: `OC_PAR`

Version 04 exposes the main design parameter of the lab, the number of output channels computed in parallel:

$$\text{OUT_GROUPS} = \left\lceil \frac{\text{OUT_CH}}{\texttt{OC_PAR}} \right\rceil, \qquad \text{DSP}_{\text{est}} = \texttt{OC_PAR} \times \text{IN_CH} \times K^2, \qquad W_{\text{out}} = 8 \times \texttt{OC_PAR} \text{ bits.}$$

The goal: **the smallest `OC_PAR` that reaches 60 FPS**. The sweep synthesizes Version 04 for `OC_PAR` = 1, 2, 4, each in its own run folder under `DSE_sweep/runs/`, and writes `DSE_sweep/reports/ocpar_sweep.md` (and `.csv`):

```
# host (after the loop of Section 1 has finished)
cd $TP06
bash DSE_sweep/run_ocpar_sweep_and_report.sh
OCPARS="1 2 3 4" bash DSE_sweep/run_ocpar_sweep_and_report.sh   # optional: 3 does not divide 64
```

While the sweep runs, read; do not edit the stage sources

The sweep configures the layer for each value, then restores every `hls/*/conv2d_top.cpp` and `conv2d_config.h` as they were when it started. An edit made to Stage 05 during the sweep would be overwritten at the end: while it runs, fill the deterministic columns below and read Section 5

(Exercise 6), and start TODO A and B only when the sweep has finished.

Exercise 5 — The `OC_PAR` sweep

Before the synthesis, fill the deterministic columns: `OUT_GROUPS`, estimated DSPs, output width. Then fill the report columns from `DSE_sweep/reports/ocpar_sweep.md`.

`OC_PAR`	`OUT_GROUPS`	DSP est.	Out width	DSP	LUT	Latency	FPS
1							
2							
4							
3 (optional)							

Choose the smallest `OC_PAR` that reaches 60 FPS, and justify it with the FPS and the resources.

5 Your code: operator fusion (Stage 05)

The previous versions optimized one operator. A CNN chains several, so the next question is whether an intermediate feature map must leave the accelerator at all. For the reference layer:

$$126 \times 126 \times 64 \xrightarrow{\text{ReLU}} 126 \times 126 \times 64 \xrightarrow{\text{MaxPool}_{2\times 2,\ s=2}} 63 \times 63 \times 64.$$

As two separate accelerators, Conv2D+ReLU writes the whole $126 \times 126 \times 64$ tensor to DDR and MaxPool reads it back. Stage 05 computes Conv2D → ReLU → MaxPool inside one HLS accelerator: the Conv2D map is consumed on chip, never stored whole. This is **operator fusion**.

Fusion removes the intermediate tensor, but MaxPool still needs a little local state: a 2×2 window spans two rows and two columns. For each output-channel lane, Stage 05 provides

```
hls::stream<data_t> pool_linebuf[OUT_GROUPS][OC_PAR];   // the value above: one Conv2D row
data_t pool_top_left_reg[OUT_GROUPS][OC_PAR];            // the values on the left
data_t pool_bottom_left_reg[OUT_GROUPS][OC_PAR];
```

and `conv_lanes[lane]` is the current Conv2D+ReLU value, the bottom-right one of the four pooling values:

```
top_left     top_right
bottom_left  bottom_right
```

Stage 05 still computes `OC_PAR` channels in parallel, but writes scalar outputs (not the packed words of Version 04), so that the pooling state is easy to read.

Exercise 6 — What fusion saves, and what it keeps

1. How many bytes has the $126 \times 126 \times 64$ int8 tensor? How many bytes of DDR traffic per image does a round trip through DDR cost? How many values remain after the pooling? Does fusion change the Conv2D MACs?
2. Why is one previous Conv2D row enough for a 2×2 window, and why are the left registers needed as well?
3. Do the pooling FIFOs hold input pixels or Conv2D outputs? Is the whole intermediate tensor stored anywhere in Stage 05?

Your implementation

The file to complete is

```
hls/05_conv_relu_pool2x2_unpacked_student/conv2d_top.cpp
```

The Conv2D datapath, the `OC_PAR` lanes, ReLU, the row FIFO and the left registers are written: read them, and make sure you can explain them. Complete only:

1. `TODO A` — the pooling value: the maximum of the four scalar values in `pool2x2_scalar`. Three comparisons are enough.
2. `TODO B` — the pooling schedule: when zero-based (oy, ox) is the bottom-right corner of a complete, non-overlapping 2×2, stride-2 window. The pooled coordinates `py = oy/2` and `px = ox/2` are given.

Exercise 7 — Complete and validate Stage 05

Before coding: $(oy, ox) = (1, 1) \to (py, px) = (\ ?,\ ?)$, $(1, 3) \to (\ ?,\ ?)$, $(3, 3) \to (\ ?,\ ?)$.
Then work so that each C simulation tests one idea: complete `TODO B` alone and run the C simulation (the pooled outputs are now written, some values still wrong); then `TODO A`, and the C simulation again. C simulation only while editing (the default of this stage), the synthesis once it passes:

```
# host
cd $TP06/hls/05_conv_relu_pool2x2_unpacked_student
$VITIS_HLS_BIN -f run_hls.tcl                          # C simulation only
RUN_CSIM=0 RUN_CSYNTH=1 $VITIS_HLS_BIN -f run_hls.tcl  # after PASS: synthesis
```

The testbench (`tb_conv2d_relu_pool2d_top.cpp`, same folder) names the failing part:

```
DIAGNOSTIC: the accelerator did not write any pooled outputs.           -> TODO B
DIAGNOSTIC: some pooled output locations were never written.            -> TODO B
DIAGNOSTIC: pooled outputs are being written, but some values are wrong. -> TODO A
```

and, when both are correct:

```
PASS: fused Conv2D + ReLU + unpacked scalar pooling output matches reference.
```

In the synthesis report: why does MaxPool add comparison and buffering logic, but no multipliers?

6 Towards Lab 7

What carries over to the final challenge

Lab 7 takes a small trained MNIST CNN to one streaming accelerator on the KV260, fed by an AXI DMA. Its convolution and pooling stages are provided and follow the ideas of this lab:

- line buffers and a window keep the spatial reuse on chip (Section 3);
- unrolled lanes buy the parallelism that just reaches a target (Section 4);
- fusion keeps the intermediate maps on chip, with a little local state (Section 5);
- tasks connected by FIFOs under `DATAFLOW`, behind an AXI-Stream boundary (Stage 06, Part B1).

In Lab 7 you write the fully-connected stage and you connect the stages in the `DATAFLOW` top yourself. If you do not do Part B, read Section 7 before Lab 7: it is the pattern you will use there.

Part B — Optional extension

Before Part B

Part B starts from the reference layer. If you changed the layer, configure it back and check it:

```
# host
cd $TP06
python3 tools/configure_layer.py --preset clean128_c64_k3_oc8
grep -E "^#define (IN_|OUT_CH|K |OC_PAR)" hls/common/conv2d_config.h
```

Section 7 needs only Vitis HLS. Section 8 also needs Vivado and a KV260 with the Ubuntu/PYNQ image; its bitstream build takes a while.

7 Optional: from operator fusion to DATAFLOW (Stage 06)

Stage 05 answered a memory question: the next operator consumes the data on chip. Stage 06 asks a scheduling question: how can the accelerator be split into producer and consumer *tasks* that run at the same time? It is the way a whole network is built in Lab 7.

Exercise 8 — Fusion is not DATAFLOW

Stage 05 is fused: its intermediate map stays on chip. Does that alone mean that several tasks run at the same time? Explain the difference between operator fusion and `DATAFLOW`.

In `hls/06_conv_relu_pool_stream_dataflow_student/`, the top-level computation is split into four tasks connected by bounded `hls::stream` FIFOs (Figure 6):

```
#pragma HLS DATAFLOW
read_input_stream(in, pixel_stream, numReps);
conv2d_relu_stream(pixel_stream, conv_stream, numReps);
pool2d_stream(conv_stream, pool_stream, numReps);
write_output_stream(pool_stream, out, numReps);
```

To keep the stage focused on streaming and scheduling, the weights and biases are deterministic values computed in `conv_pool_stream_weights.h`, not fetched from DDR: the top function only needs the input stream, the output stream and `numReps`. Lab 7 uses trained weights stored the same way, inside the accelerator.

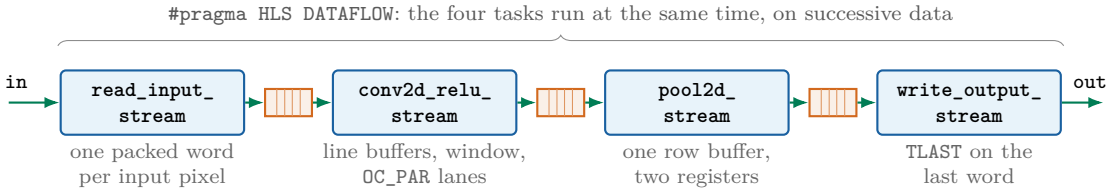


Figure 6: Stage 06: four tasks under `#pragma HLS DATAFLOW`, connected by the `hls::stream` FIFOs `pixel_stream`, `conv_stream` and `pool_stream` (depth 64). The slowest task sets the throughput.

A full FIFO stalls its producer and an empty FIFO stalls its consumer. This flow control, **back-pressure**, synchronizes the tasks while the activations stay on chip. The lab now uses three kinds of parallelism:

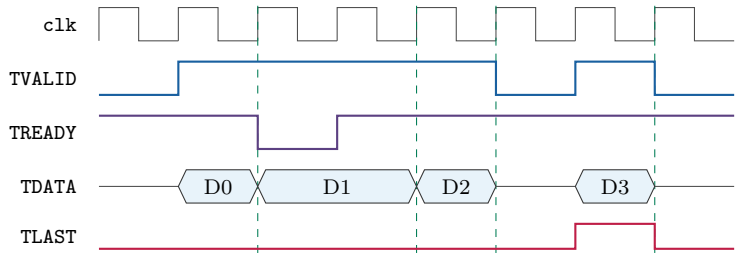
Pragma	Overlaps	In this lab
`UNROLL`	copies of hardware	MACs, `OC_PAR` lanes
`PIPELINE`	successive loop iterations	pixels, output groups
`DATAFLOW`	different tasks	read, Conv+ReLU, pool, write

The AXI-Stream boundary

The top function has no address and no weight port:

```
void conv2d_top(hls::stream<axis_in_t> &in, hls::stream<axis_out_t> &out, const unsigned int numReps);
#pragma HLS INTERFACE axis register both port=in        // and port=out
#pragma HLS INTERFACE s_axilite port=numReps bundle=control
#pragma HLS INTERFACE s_axilite port=return  bundle=control
```

One input word packs the `IN_CH` channels of one pixel; one output word packs `OC_PAR` pooled lanes. On the wires, a word moves only when the sender has one and the receiver can take it (Figure 7); `TLAST` marks the last word of the call, of `numReps` images, so that a DMA knows where its transfer ends.



A word moves at a rising edge where `TVALID` and `TREADY` are both 1 (dashed lines). `TREADY` at 0: the receiver is full, and the sender holds the same word (back-pressure). `TLAST`: the last word of the transfer.

Figure 7: The AXI-Stream handshake: a word moves on a rising edge where `TVALID` and `TREADY` are both 1.

DATAFLOW is not DMA

There is no AXI DMA in the Stage 06 source. `DATAFLOW` schedules HLS tasks; an AXI DMA is a separate Vivado block that moves data between DDR and an AXI-Stream accelerator (Figure 8(b)). Lab 7 uses one.

Exercise 9 — Read the architecture, complete and validate Stage 06

First read the code: which task produces and which consumes each internal FIFO? Why does no FIFO need room for a whole feature map? What happens to a producer whose output FIFO is full, and to a consumer whose input FIFO is empty?

The streaming Conv2D+ReLU, the input task and the pooling state are written. Complete only:

1. `TODO 1` — in `pool2x2_word`: apply the provided scalar maximum `max4_scalar` independently to every packed pooling lane;
2. `TODO 2` — in `write_output_stream`: assert `TLAST` only on the final output word of the whole call.

```
# host
cd $TP06/hls/06_conv_relu_pool_stream_dataflow_student
$VITIS_HLS_BIN -f run_hls.tcl                          # C simulation only (default of this stage)
RUN_CSIM=0 RUN_CSYNTH=1 $VITIS_HLS_BIN -f run_hls.tcl  # after PASS: synthesis, for Exercise 10
```

The testbench sends two images in one call and names what to check (wrong pooled values: `TODO 1`; packet boundary: `TODO 2`). At the end:

```
PASS: streaming Conv2D+ReLU+Pool matches reference.
```

Exercise 10 — Find the DATAFLOW bottleneck

After the synthesis, read the report (or the Dataflow Viewer of the Vitis HLS GUI) and record, for each task, the achieved II or other throughput evidence:

Task	Relevant loop or report item	Achieved II or evidence
`read_input_stream`	input-word loop	
`conv2d_relu_stream`	raster loop, output groups	
`pool2d_stream`	pooling loop	
`write_output_stream`	output-word loop	

Which task limits the steady-state throughput? Why does accelerating a task that is already faster than this bottleneck not improve the end-to-end throughput?

Read Stage 06 qualitatively

The weights of Stage 06 are compile-time constants: Vitis HLS may simplify the constant multiplications, so do not compare its DSP count with Stages 04–05. Use this stage for the streams, the task concurrency, the initiation intervals, back-pressure and the AXI-Stream boundary.

8 Optional: Version 04 on the KV260

How an accelerator reaches the memory

The images and the results live in the DDR of the board. An HLS accelerator reaches them in one of two ways (Figure 8): memory-mapped, as Version 04, which this section puts on the board; or streaming behind an AXI DMA, as Stage 06 and Lab 7.

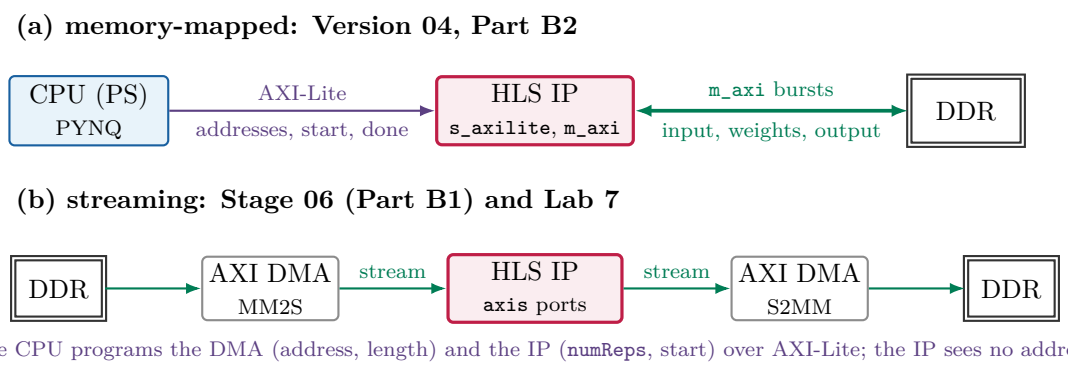


Figure 8: (a) Memory-mapped: the CPU writes the buffer addresses and starts the IP; the IP then reads and writes the DDR itself through its AXI master ports, in bursts. (b) Streaming: an AXI DMA reads the DDR and pushes a stream into the IP, and writes the stream that comes out back to DDR.

Version 04 is memory-mapped; its interface pragmas say so:

```
#pragma HLS INTERFACE m_axi     port=input   offset=slave bundle=gmem0
#pragma HLS INTERFACE m_axi     port=output  offset=slave bundle=gmem1
#pragma HLS INTERFACE m_axi     port=weights offset=slave bundle=gmem2
#pragma HLS INTERFACE m_axi     port=bias    offset=slave bundle=gmem3
#pragma HLS INTERFACE s_axilite port=input   bundle=control   // likewise output, weights, bias
#pragma HLS INTERFACE s_axilite port=return  bundle=control
```

Pragma	Hardware meaning	Used later by
`m_axi`	the IP reads and writes DDR itself	the Vivado data path
`s_axilite`	a small register map written by the CPU	PYNQ register writes
`offset=slave`	the buffer address is written there at run time	PYNQ physical addresses
`port=return`	`AP_START`, `AP_DONE`, `AP_IDLE`	PYNQ control

So the PYNQ code writes physical addresses, then starts the IP:

```
write_u64(ip, REG_INPUT, physical_address(input_buf))     # likewise output, weights, bias
ip.write(REG_CTRL, 0x01)                                  # AP_START
```

We deploy Version 04 rather than Stage 06 on purpose: its `m_axi` interface lets you focus on the PS/PL connections, the address map, the bitstream and PYNQ without an AXI DMA at the same time. The workflow is

$$\text{HLS C++} \rightarrow \text{HLS RTL IP} \rightarrow \text{Vivado block design} \rightarrow \text{bitstream} \rightarrow \text{PYNQ on the KV260},$$

on a small layer, so that the export, the implementation and the board test stay short. PYNQ is only the software used here: the same hardware could be driven by a C or XRT application.

Step 1 — A small layer

```
# host
cd $TP06
python3 tools/configure_layer.py --preset kv260_small_oc2       # 8x8x3 -> 4 channels, K=3, OC_PAR=2
grep -E "^#define (IN_|OUT_|K |OC_PAR)" hls/common/conv2d_config.h
grep "CONFIGURE_LAYER_OC_PAR_UNROLL" hls/04_linefifo_oc_parallel_packed/conv2d_top.cpp
```

`OC_PAR` is fixed before the export: it changes the hardware, the output packing and the buffer layout. The PYNQ application of Step 6 supports `OC_PAR` $\in \{1, 2, 4, 8\}$, for which one packed word occupies exactly `OC_PAR` bytes in DDR; other values (3 in the sweep) are fine for synthesis estimates, but the `m_axi` port then pads each word to a power-of-two number of bytes.

Step 2 — Export Version 04 as a Vivado IP

C simulation, synthesis, C/RTL co-simulation and export, once (at this small size the co-simulation takes a few minutes), then extract the IP:

```
# host
cd $TP06/hls/04_linefifo_oc_parallel_packed
RUN_COSIM=1 RUN_EXPORT=1 $VITIS_HLS_BIN -f run_hls.tcl
cd $TP06
mkdir -p kv260/exported_ip/conv2d_04
unzip -o exported_ip/conv2d_04_linefifo_oc_parallel_packed_hls_ip.zip -d kv260/exported_ip/conv2d_04
find kv260/exported_ip/conv2d_04 -name component.xml
```

The last command must find `component.xml`, the IP descriptor. To export the same code again later (a lost ZIP, for example), skip both simulations: `RUN_CSIM=0 RUN_EXPORT=1`. In Vivado this folder is an *IP repository*: never add `conv2d_top.cpp` or the generated Verilog/VHDL as design sources.

Step 3 — The system follows from the interfaces

Before opening Vivado, derive what the IP needs from the system around it:

HLS interface	AXI role	Initiates transactions?	Purpose
`s_axi_control`	AXI-Lite slave	no	control, status and buffer-address registers
`m_axi_gmem0`	AXI master	yes	reads the input activations
`m_axi_gmem1`	AXI master	yes	writes the output activations
`m_axi_gmem2`, `gmem3`	AXI master	yes	read the weights and the biases

`M` and `S` are named from the block on which the interface appears: the master `m_axi_gmem0` of the IP connects to an `Sxx_AXI` input of an interconnect, whose `Mxx_AXI` output reaches a slave such as `S_AXI_HPC0_FPD` of the processing system. Two paths follow (Figure 9): the PS initiates the register accesses of the control path; the IP initiates the bulk DDR accesses of the data path. The processor never pushes the activations through AXI-Lite.

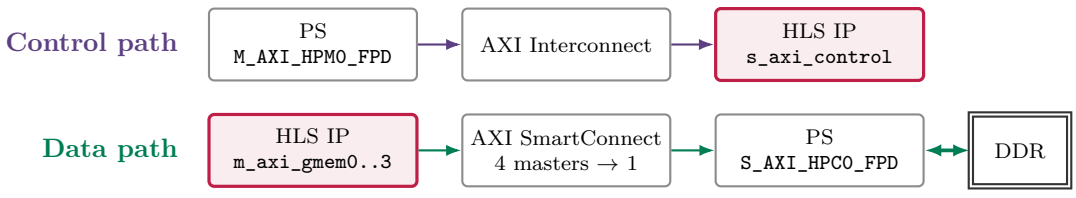


Figure 9: The two communication paths of Version 04. Clocks and resets are not drawn.

Exercise 11 — Derive the connections before Vivado

1. Who initiates a write to `AP_START`: the PS or the HLS IP? Is `s_axi_control` a master or a slave?
2. Who initiates the read of `input[]` from DDR?
3. Why do the four `m_axi_gmem*` interfaces need an interconnect before one PS port?
4. Which path carries the tensors: the control path or the data path?

Step 4 — The block design in Vivado

Start `$VIVADO_BIN` and let the IP Integrator automation build the infrastructure:

1. **File** → **Project** → **New Project**: an RTL project; in the **Boards** tab, the **Kria KV260 Vision AI Starter Kit** (the K26 device `xck26-sfvc784-2LV-c` alone lacks the board preset of the processing system, DDR included: the KV260 board files must be installed in Vivado).
2. **Tools** → **Settings** → **IP** → **Repository**, +: the folder `kv260/exported_ip/conv2d_04`. The repository makes the IP available; it does not instantiate it.
3. **IP Integrator** → **Create Block Design** (`design_1`), **Add IP**: `Zynq UltraScale+ MPSoC`, then **Run Block Automation** with the board preset.
4. Double-click the PS, **PS-PL Configuration**: enable, if needed, `M_AXI_HPM0_FPD` (the master of the control path) and `S_AXI_HPC0_FPD` (the slave through which the HLS masters reach DDR); check that `pl_clk0` is at 100 MHz.
5. **Add IP**: `conv2d_top` (if it is missing, check the repository and refresh the catalog; never import the RTL files).
6. **Run Connection Automation**: `s_axi_control` to the master `M_AXI_HPM0_FPD`; each of `m_axi_gmem0..3` to the slave `S_AXI_HPC0_FPD`. Vivado inserts the AXI Interconnect, the AXI SmartConnect and the Processor System Reset.

Automation inserts blocks, not just wires

Instance names, order and number of ports may differ from Figure 10; check the AXI endpoints, the clocks and the resets, not the drawing: a wrong choice in a Connection Automation dialog still gives a connected-looking diagram. Do not use **Make External** on the AXI interfaces: they are internal connections, not KV260 pins.

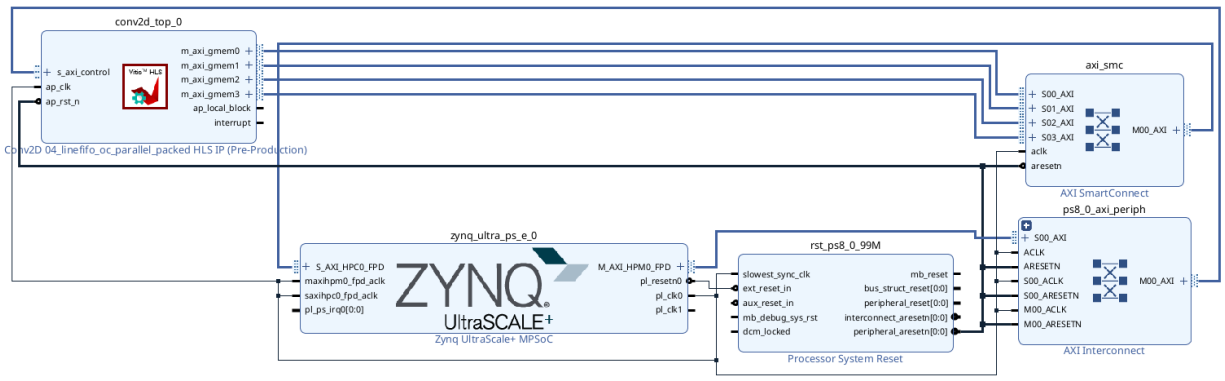


Figure 10: The reference design built with Block and Connection Automation. `ps8_0_axi_periph` (AXI Interconnect) carries the control path to `s_axi_control`; `axi_smc` (AXI SmartConnect) combines the four HLS `m_axi` masters before `S_AXI_HPC0_FPD`; `rst_ps8_0_99M` distributes the resets. Names and layout may differ, and the automation may also enable `M_AXI_HPM1_FPD`: match the connections, not the drawing.

Exercise 12 — Verify the block design

Trace in your own diagram, before any bitstream:

- the PS reaches `conv2d_top_0/s_axi_control` through an interconnect (an extra `M_AXI_HPM1_FPD` added by the automation is harmless);
- the four `m_axi_gmem*` reach the SmartConnect (in the reference design `gmem0..3` on `S00..S03_AXI`: input, output, weights, bias; your numbers may differ), and its `M00_AXI` reaches `S_AXI_HPC0_FPD`;
- `pl_clk0` (100 MHz) drives `ap_clk`, the interconnect clocks, the enabled PS–PL interface clocks

and `slowest_sync_clk` of the reset block;
- every active-low reset input (`ap_rst_n`, `*aresetn`) is driven by an active-low reset output (`peripheral_aresetn`, `interconnect_aresetn`);
- no AXI interface of the IP is made external.

The `interrupt` output may stay unconnected (the application polls for completion), and an unused `pl_clk1` or `ap_local_block` is not an error. If a required connection is missing, correct the architecture before going on.

Step 5 — Addresses, validation, bitstream

In the **Address Editor**, assign the addresses, then check two things: `conv2d_top_0/s_axi_control` has a base address seen by the PS (PYNQ uses it to reach the registers), and each `m_axi_gmem*` can address the DDR segment through `S_AXI_HPC0_FPD` (these are the addresses the IP uses once PYNQ has written the physical buffer addresses). Then run **Validate Design** and fix every error.

Exercise 13 — One complete transaction

PYNQ allocates an input buffer in DDR and obtains its physical address. How does the HLS IP get that address? Which interface then initiates the read of the input tensor, and through which Vivado blocks does that read travel to DDR?

1. In **Sources**, right-click `design_1.bd`: **Generate Output Products**; then **Create HDL Wrapper**, letting Vivado manage it.
2. **Before synthesis**, right-click `design_1_wrapper`: **Set as Top**. In the Tcl Console, the first command must answer `design_1_wrapper`; otherwise, run the other two:

```
get_property top [get_filesets sources_1]
set_property top design_1_wrapper [get_filesets sources_1]
update_compile_order -fileset sources_1
```

3. **Generate Bitstream** (synthesis and implementation run first; read their resource reports). It takes a while.

Placement error: more I/O pins than the KV260 has

`[Place 30-415] IO Placement failed due to overutilization` (or an `ap_clk_IBUF_BUFG` placement error) means that Vivado implements the HLS module alone as top, all its AXI ports as pins, instead of the block-design wrapper. Set `design_1_wrapper` as top as above, reset the synthesis and implementation runs, and run them again; remove from the project (not from disk) any `conv2d_top.v` or `.vhd` added by hand.

PYNQ needs the bitstream and the hardware handoff file of the **same** Vivado build. For a project `kv260_conv2d` and a block design `design_1` they are usually (otherwise `find . -name "*.bit" -o -name "*.hwh"` in the project folder):

```
kv260_conv2d.runs/impl_1/design_1_wrapper.bit
kv260_conv2d.gen/sources_1/bd/design_1/hw_handoff/design_1.hwh
```

PYNQ loads the `.bit` to program the fabric and reads the `.hwh` to find the IP names, the AXI-Lite register map and the addresses.

Step 6 — Run it with PYNQ

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 KV260 now runs the Ubuntu image with PYNQ, not the PetaLinux image of Labs 2–5: the account is `ubuntu`, which the lab scripts use. `BOARD` is the network name of your board, as in Lab 2. Give it once per terminal and check the access by hand:

```
# host
export BOARD=maquette-07          # the name of your board, or its IP address
```

```
ssh ubuntu@${BOARD#*@} hostname   # BOARD without a user@ part, as in Lab 2
```

The first time, accept the key of the board. `REMOTE HOST IDENTIFICATION HAS CHANGED` means the name was used with the PetaLinux image, which has another key: `ssh-keygen -R ${BOARD#*@}`, then connect again. If the name does not answer, read the address on the serial console as in Lab 2 (`tio`, log in as `ubuntu`, `ip -4 addr show eth0`) and put it in `BOARD`.

Copy the two files of the same build under one base name (PYNQ looks for the `.hwh` next to the `.bit`), then deploy:

```
# host
cd $TP06
cp <vivado_project>/kv260_conv2d.runs/impl_1/design_1_wrapper.bit kv260/bitstreams/conv2d.bit
cp <vivado_project>/kv260_conv2d.gen/sources_1/bd/design_1/hw_handoff/design_1.hwh kv260/bitstreams/conv2d.hwh
bash host/deploy_pynq_to_kv260.sh
```

The script shares one SSH connection (the password is asked once) and copies six files to `/home/ubuntu/c5/TP06/` on the board: the bitstream and its hardware handoff (`conv2d.bit`, `conv2d.hwh`), the application `kv260_conv2d_pynq_app.py`, and the three scripts `preflight_pynq.sh`, `ensure_pynq_runtime.sh` and `run_pynq_smoke.sh`. On the board, run the pre-flight first:

```
# host
ssh ubuntu@${BOARD#*@}
# KV260
cd /home/ubuntu/c5/TP06
./preflight_pynq.sh
```

It checks the files and the PYNQ runtime. After a boot of the board it also enables the PYNQ device-tree overlay once, with `ensure_pynq_runtime.sh` (`sudo` password): without it, the ZOCL driver is not registered, XRT sees no device and loading the overlay fails with `ENODEV`. Run it again after each reboot. If the SmartCam application (separate optional material) is running, it asks you to reboot the board first. The pre-flight runs in its own shell and cannot set variables in yours: set up the PYNQ environment in this shell, with the layer parameters of the export:

```
# KV260
source /etc/profile.d/pynq_venv.sh
export XILINX_XRT=/usr
export PATH=/usr/local/share/pynq-venv/bin:$PATH
export IN_H=8 IN_W=8 IN_CH=3 OUT_CH=4 K=3 OC_PAR=2
```

First check that the overlay loads and that the register map is visible, then run the correctness test:

```
# KV260
sudo -E env PATH="$PATH" XILINX_XRT="$XILINX_XRT" \
  /usr/local/share/pynq-venv/bin/python3 kv260_conv2d_pynq_app.py \
  --in-h ${IN_H} --in-w ${IN_W} --in-ch ${IN_CH} \
  --out-ch ${OUT_CH} --k ${K} --oc-par ${OC_PAR} --mode smoke
sudo -E env PATH="$PATH" XILINX_XRT="$XILINX_XRT" \
  /usr/local/share/pynq-venv/bin/python3 kv260_conv2d_pynq_app.py \
  --in-h ${IN_H} --in-w ${IN_W} --in-ch ${IN_CH} \
  --out-ch ${OUT_CH} --k ${K} --oc-par ${OC_PAR} --mode golden --runs 10
```

If `smoke` fails, check the `.bit`/`.hwh` pair, the IP instance name `conv2d_top_0` and the block design before debugging the convolution. `golden` initializes deterministic inputs, weights and biases, runs the accelerator, computes the same Conv2D in Python, unpacks the packed output and compares every value:

```
PASS: hardware output matches the reference.
```

It also prints a small benchmark: for this 8×8 layer (144 output values) the time is that of the Python calls and the AXI-Lite accesses, not of the accelerator. It is a functional test, not the throughput of the reference layer. For later projects, `kv260/software/pynq_hls_m_axi_template.py` is a starting point for an `m_axi` IP, and `pynq_dma_axis_template.py` for an AXI-Stream IP behind a DMA (Lab 7).

Finally, configure the reference layer again on the host:

```
# host
cd $TP06 && python3 tools/configure_layer.py --preset clean128_c64_k3_oc8
```

A Using the layer configuration tool

The active layer configuration is `hls/common/conv2d_config.h`. Do not edit it by hand during the lab: the configuration tool keeps every derived parameter consistent. It updates the layer dimensions (`IN_H`, `IN_W`, `IN_CH`, `OUT_CH`, `K`), the output dimensions (`OUT_H`, `OUT_W`, `OUT_GROUPS`), `OC_PAR`, the FIFO capacities (`LINEBUF_DEPTH = 2*IN_W`, `POOL_LINEBUF_DEPTH = 2*OUT_W`) and the unroll factor of the `OC_PAR` lane loop.

```
# host
python3 tools/configure_layer.py --preset clean128_c64_k3_oc8   # reference layer (default)
python3 tools/configure_layer.py --preset kv260_small_oc2       # C/RTL co-simulation, KV260 test
python3 tools/configure_layer.py --list-presets                 # other presets (64x64, 224x224, ...)
python3 tools/configure_layer.py --in-h 64 --in-w 64 --in-ch 3 --out-ch 64 --k 3 --oc-par 4
grep -E "^#define (IN_|OUT_|K |OC_PAR|LINEBUF|POOL_)" hls/common/conv2d_config.h   # always check
```

The fourth form sets every parameter by hand; `--preset P --oc-par N` starts from a preset and overrides one value. `OC_PAR` is a compile-time hardware parameter: after a change, run the synthesis again and, for the board, export a new IP. It changes the number of lanes, the packed output width, `OUT_GROUPS`, the resources and the PYNQ output-buffer layout.

Scaling rules: the image size (`IN_H`, `IN_W`) sets the output pixels and the MACs; `IN_CH` the dot-product length, the number of FIFOs and the window; `OUT_CH` the outputs, the weights and the MACs; `K` the line-buffer height and the dot-product length; `OC_PAR` the parallel hardware and the number of output groups.

B Self-check (optional)

These questions check the main ideas of the lab: Conv2D, HLS pragmas, streaming buffers, fusion, and, for Part B, DATAFLOW, AXI interfaces and PYNQ. They are not graded.

Self-check

Choose the correct answer.

Q1. For valid convolution, stride 1, and no padding, the output size is:

A. $H \times W$
B. $(H - K + 1) \times (W - K + 1)$
C. $(H + K - 1) \times (W + K - 1)$
D. $(H/K) \times (W/K)$

Q2. Line buffers are mainly used to:

A. reduce the number of weights
B. reduce the number of MACs
C. reuse recent input rows locally
D. increase the output image size

Q3. As in the lab code, the pragma below is the first line of a loop body. It applies to:

```
for (int kx = 0; kx < K; kx++) {
#pragma HLS UNROLL
    ...
}
```

A. all loops in the function
B. the loop whose body contains it (here the `kx` loop)
C. only loops named `kx`
D. only loops inside test benches

Q4. Fully unrolling the loops `ic`, `ky`, and `kx` for `IN_CH` $= 3$ and $K = 3$ creates approximately:

A. 3 multipliers
B. 9 multipliers
C. 27 multipliers
D. 64 multipliers

Q5. A partial unroll such as:

```
#pragma HLS UNROLL factor=2
```

usually means:

A. the loop runs twice slower
B. two loop iterations are executed in parallel
C. the loop is removed from the design
D. the clock frequency is doubled

Q6. The pragma below is mainly used because unrolled hardware needs parallel access to array elements:

```
#pragma HLS ARRAY_PARTITION variable=window complete dim=0
```

A. True
B. False

Q7. A loop with:

```
#pragma HLS PIPELINE II=1
```

means HLS tries to:

A. finish the whole loop in one cycle
B. start a new loop iteration every cycle
C. use exactly one DSP
D. remove all memory accesses

Q8. In Version 04, increasing `OC_PAR` mainly:

A. computes more output channels in parallel
B. changes the convolution kernel size

C. changes the input image resolution
D. reduces the dot-product size

Q9. If `OC_PAR=4`, one packed output word contains:
A. one int8 output value
B. four int8 output values
C. one complete output feature map
D. all output pixels

Q10. In an HLS top function, an interface such as:

```
#pragma HLS INTERFACE m_axi port=input offset=slave bundle=gmem0
```

means that `input` is:
A. a register written directly by the CPU
B. an AXI master port used to access external memory
C. a local BRAM array only visible inside the IP
D. a debug-only signal

Q11. In PYNQ, input and output buffers are typically created using:
A. normal Python lists
B. `numpy.random` only
C. physically contiguous allocated buffers
D. text files

Q12. When launching the HLS IP from PYNQ, the Python code usually needs to:
A. pass the physical addresses of the input and output buffers
B. manually edit the Vivado address map
C. re-run C synthesis on the board
D. rewrite the Verilog source code

Q13. C Simulation checks:
A. C/C++ functional correctness
B. FPGA timing closure
C. PYNQ buffer allocation
D. board-level power consumption

Q14. C Synthesis estimates:
A. latency and hardware resources
B. only software runtime
C. only Python execution time
D. only test-bench correctness

Q15. Operator fusion in Stage 05 mainly means:
A. the Conv2D and pooling stages must use AXI DMA
B. the complete intermediate Conv2D tensor does not need to be written to DDR
C. all loops are automatically unrolled
D. the pooling layer uses DSP multipliers

Q16. `#pragma HLS DATAFLOW` is mainly used to:
A. duplicate every multiplier in a function
B. overlap different producer/consumer tasks connected by streams
C. assign AXI addresses in Vivado
D. allocate physically contiguous PYNQ buffers

Q17. In the streaming Stage 06 architecture, AXI DMA would be:
A. an HLS pragma required for `DATAFLOW`
B. a separate Vivado block that can move data between DDR and AXI-Stream
C. another name for the line buffer
D. required for every CNN accelerator

End of lab.