Profiling: Finding the Bottleneck

Rodolfo Azevedo

Institute of Computing, University of Campinas (UNICAMP), Brazil

rodolfo.azevedo@unicamp.br

http://www.ic.unicamp.br/~rodolfo/mo801

Goal of this class

Module 4, Class 3: cycle counts per layer, arithmetic intensity, and the back-of-envelope accelerator spec.

> "Premature optimization is the root of all evil." — Knuth. The corollary for hardware: *measure first, accelerate second.* This class is about collecting the data that drives Project 3's design decisions.

At the end of this class, you should be able to:

- Instrument the KWS kernel with `rdcycle` wrappers to collect per-layer cycle counts.
- Compute arithmetic intensity (MACs per byte transferred) for the Conv2D and FC layers.
- Identify the bottleneck layer and quantify the speedup needed to meet a latency target.
- Derive a back-of-envelope accelerator specification from the profiling data.

Profiling methodology

Wrap each layer call with cycle counter reads:

```
uint32_t t0, t1;

t0 = read_cycle();
conv2d_int8(input, weights_conv, act1, ...);
t1 = read_cycle();
uart_printf("conv2d:  %u cycles\r\n", t1 - t0);

t0 = read_cycle();
fc_int8(act1, weights_fc, output, ...);
t1 = read_cycle();
uart_printf("fc:      %u cycles\r\n", t1 - t0);
```

Run on the Tang Nano 9K via the UART bootloader. Read the output on your serial terminal. The numbers you collect here are your **baseline** — write them down before any optimization.

Before you trust your numbers — a checklist

Profiling numbers are only as good as the measurement setup. Verify these before recording your baseline:

- [ ] **Reset the cycle counter before each measurement**: a stray `rdcycle` without a matching reset will give cumulative numbers across layers.
- [ ] **Run the inference twice and discard the first**: the first run may include cold-start effects (cache fills on systems with caches, BRAM initialization).
- [ ] **Disable UART output during the timed region**: `uart_printf` is slow (~10,000 cycles per character at 115200 baud). Print before and after, not during.
- [ ] **Use the same binary for baseline and accelerated runs**: different compiler flags between runs invalidate the comparison.
- [ ] **Verify correctness before recording speed**: a fast but wrong implementation is not a baseline — it is a bug. Check `output == expected_output` before logging cycle counts.
- [ ] **Record the exact `-march` and `-O` flags**: the same source compiled with `-O0` vs `-O2` can differ by 10× — your baseline flag must match what you use in the final comparison.

> A common mistake: comparing a carefully optimized, `-O2` accelerated run against a sloppy `-O0` software baseline. The "speedup" then includes compiler benefit, not accelerator benefit. Control the variable.

Expected profiling results

At 27 MHz, RV32I + Zicsr + Zicntr + Zmmul + CMAC, `-O2`:

Layer	MACs	Cycles (est.)	% of total
Conv2D (main layer)	K×K×C_in×C_out×H_out×W_out	dominates	**> 90%**
Fully Connected	C_out×4	negligible	< 10%
Total	**from ref. impl.**	**to be measured**	**100%**

The exact numbers depend on the layer dimensions from the reference implementation; measure your actual baseline before committing to an accelerator design. What you should confirm:

- The Conv2D layer accounts for **the vast majority of total cycles**.
- KWS needs < 1 second; verify your baseline fits the budget (with margin for MFCC computation).
- Record the total cycle count — this is your **System v0 baseline** for Project 3.

The bottleneck: Conv2D

The Conv2D layer dominates runtime. Why?

- **Many MACs**: the vast majority of all multiply-accumulate operations are in this single layer — the FC layer contributes negligibly.
- **Memory access pattern**: for each output channel at each spatial position, we read KH×KW×C_in activation values and KH×KW×C_in weight values. Our core has no cache — every access goes directly to BRAM or DMEM (1 cycle each).
- **Weight reuse**: the same C_out filter banks are applied at every spatial position (H_out×W_out positions). With a local BRAM weight buffer in the accelerator, this reuse is exploited: weights are loaded once and applied across all positions, dramatically improving arithmetic intensity.

This is why the accelerator in Project 3 targets the Conv2D inner loop: it moves the weight buffer on-chip, turning a memory-bound operation into a compute-bound one.

Arithmetic intensity: MACs per byte

Arithmetic intensity (AI) measures how much compute we do per byte of data moved:

$$\text{AI} = \frac{\text{MACs}}{\text{bytes loaded from memory}}$$

For Conv2D (one output channel, one spatial position, in software):

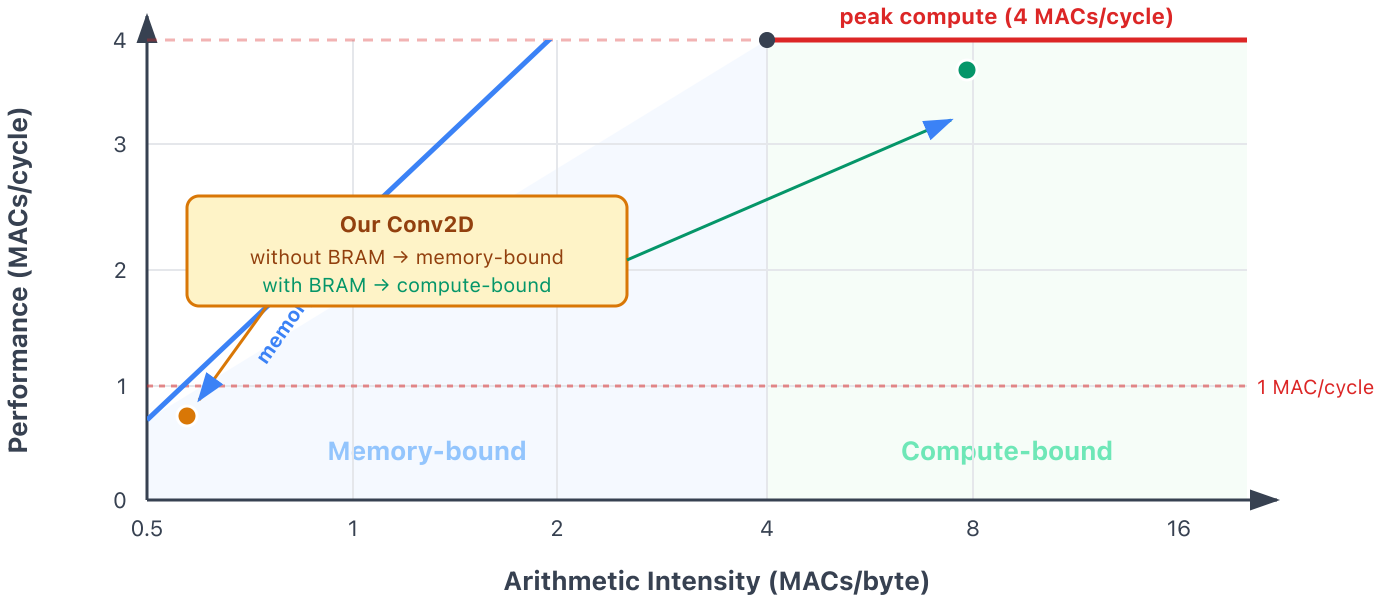
- **MACs**: KH×KW×C_in (one dot product over the kernel × channels)
- **Bytes loaded**: KH×KW×C_in (input activations) + KH×KW×C_in (weights) = 2×KH×KW×C_in bytes
- **AI** = 0.5 MACs/byte

This is very low — for every MAC, we load 2 bytes. The operation is **memory-bandwidth bound**, not compute bound. Adding more MAC units without increasing memory bandwidth will not help proportionally.

With a local BRAM weight buffer (weights loaded once, reused across all H_out×W_out positions):

$$\text{AI} \approx \frac{C_{out} \times K^2 \times C_{in}\ \text{MACs}}{K^2 \times C_{in}\ \text{bytes (activations only)}} = C_{out}\ \text{MACs/byte}$$

The roofline preview



At AI = 0.5 MACs/byte, even a 4-wide SIMD accelerator is mostly waiting for memory. To get a real speedup, we either:

1. **Increase AI** — tile the computation so weights are reused (keep them in a local BRAM buffer instead of fetching from DMEM every time).
2. **Increase bandwidth** — use a wider bus or DMA.

Project 3's accelerator does option 1: a local weight buffer reduces DMEM traffic.

From profiling data to accelerator spec

A useful back-of-envelope: how fast do we *need* the accelerator to be?

Target: < 100 ms total inference (target 10× speedup over your measured baseline).

Conv2D dominates (>90% of runtime). After acceleration it should drop to < 10 ms:

$$\text{Required speedup on Conv2D} = \frac{0.9 \times \text{baseline}}{10\,\text{ms}}$$

With a 4-wide int8 MAC array running at 27 MHz with local BRAM weight buffering:

- 4 MACs/cycle × 27M cycles/s = 108M MACs/s
- Conv2D is compute-bound (weight buffer eliminates memory bottleneck)
- Expected layer speedup: **~4×** (matching the MAC-array width)

Because Conv2D is >90% of runtime, Amdahl's law gives:

$$\text{System speedup} = \frac{1}{0.10 + 0.90/4} \approx 3.1\times$$

Conclusion: **a 4-wide MAC array with local BRAM buffering gives ~3× system speedup** — meaningful and achievable. The local weight buffer is the architectural decision that matters most: without it, the MAC array is memory-bound and the speedup collapses.

Amdahl's Law — applied to your numbers

Amdahl's Law gives the theoretical maximum system speedup when only a fraction f of the workload is accelerated by factor S:

$$\text{Speedup}_{system} = \frac{1}{(1-f) + f/S}$$

Fill in your measured values:

Variable	Formula	Your value
f = Conv2D fraction	(Conv2D cycles) / (total cycles)	
S = layer speedup	(SW Conv2D cycles) / (HW Conv2D cycles)	
System speedup	$1/((1-f) + f/S)$	

Example: $f = 0.92$, $S = 4$ → system speedup = $1/(0.08 + 0.92/4) = 1/(0.08 + 0.23)$ = **3.2×**

Key insight: even with a **perfect** Conv2D accelerator ($S \to \infty$), the maximum possible speedup is $1/(1-f) = 1/0.08 = 12.5\times$. The non-accelerated 8% is the hard ceiling.

> This is why profiling comes **before** design: without measuring f, you cannot predict what speedup any accelerator will achieve, regardless of how fast it is.

In-class exercise — from profiling to spec

Given the following (hypothetical) profiling output from a student's board:

```
conv2d:  3,200,000 cycles   (91.4%)
fc:        185,000 cycles   ( 5.3%)
overhead:  116,000 cycles   ( 3.3%)
total:   3,501,000 cycles
```

Questions (10 min):

1. What is the arithmetic intensity of the Conv2D layer in this run, assuming `KH=KW=3`, `C_in=1`, `C_out=8`, `H_out=24`, `W_out=19`, and the memory is not cached? (Hint: MACs ÷ bytes loaded)
2. Using Amdahl's Law with $f = 0.914$, what system speedup does a 4× layer speedup achieve?
3. If you could only choose one: (a) double the MAC array width from 4 to 8, or (b) add a weight prefetch buffer that doubles the effective memory bandwidth — which gives a larger system speedup? Why?

> Expected: (1) AI = 0.5 MACs/byte (memory-bound without local buffer). (2) System speedup ≈ 3.3×. (3) The weight buffer (b) wins if currently memory-bound — doubling MACs with a memory-bound design gives < 2× improvement. The buffer moves you into the compute-bound region where more MACs help.

What to report in Project 3

Your Project 3 submission must include:

Measurement	Where to get it
Baseline cycles (per layer)	M04A03 profiling output
Accelerator cycles (per layer)	M05A03 end-to-end measurement
Speedup per layer and system-wide	ratio of the above
Arithmetic intensity of your design	MACs ÷ bus bytes (calculated)
LUTs, FFs, BRAMs used by accelerator	nextpnr utilization report
Was the bottleneck compute or memory?	Roofline analysis (M05A04)

These six numbers tell the complete story of your design.

Lab 3 due — `-O0` vs. `-O2` cycle measurement

Lab 3 (assigned in M02A05) is due this class:

- Cycle counts for the 64-element int8 dot product at `-O0` and `-O2`.
- Disassembly of both versions with annotation: which instructions disappeared?
- (Bonus) With the `M` extension enabled: how much does `mul` change the count?

Submit the cycle count table, the annotated disassembly, and the source + Makefile.

Next class

Design Space for the Accelerator: interface options, datapath width trade-offs, BRAM budgeting, and the Project 3 kickoff. You leave that class with a concrete spec to implement.