MO801/MC972 · Topics in Computer Architecture and Hardware · Rodolfo Azevedo · CC BY-SA 4.0

The KWS Inference Kernel

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 2: a line-by-line walkthrough of the dependency-free C kernel you will profile and accelerate.

> The kernel is provided — you do not write it from scratch. But you need to understand every line, because your accelerator must produce bit-identical results to it. This class is a code reading session with arithmetic.

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

- Walk through the KWS C kernel function by function: `conv2d_int8`, `fc_int8`, and the softmax output stage.
- Compute the number of multiply-accumulate (MAC) operations for the Conv2D and FC layers by hand.
- Identify the most computationally intensive layer and explain why it dominates the cycle count.
- Explain the weight layout and int8 accumulation logic that any hardware accelerator must replicate exactly.

File layout

```
kws/
  kws_kernel.c      ← the inference engine (~300 lines)
  kws_kernel.h      ← public API: kws_init(), kws_run()
  weights.h         ← const int8_t arrays: conv weights, biases, scales
  test_vectors.h    ← const int8_t input[49*40]; expected int8_t output[4]
  platform.h        ← read_cycle(), uart_putc() — thin platform abstraction
```

The only dependency is `<stdint.h>`. Compile on any C99 toolchain — on your PC for fast iteration, on the RV32I core for the real measurement.

```
# On your PC (sanity check before sending to the board)
gcc -O2 -o kws_test kws_kernel.c && ./kws_test
# Expected output: "yes (score=112, 3847291 cycles)"
```

tiny_conv layer dimensions — the numbers you'll use everywhere

Tensor	Shape	Notes
Input MFCC	`[H][W][C_in]`	Read from `test_vectors.h`; `H×W×C_in` bytes total
Conv2D weights	`[C_out][KH][KW][C_in]`	Stored row-major in `weights.h`
Conv2D output	`[H_out][W_out][C_out]`	`H_out = (H−KH)/stride + 1`
FC weights	`[4][C_out]`	4 output classes × C_out inputs
FC output	`[4]`	Scores for: yes / no / unknown / silence

Run `python3 inspect_model.py tiny_conv.tflite` (from M04A01's TFLite introspection slide) to read off the exact values of H, W, C_in, KH, KW, C_out, and stride.

MAC count formula (the inner product from `conv2d_int8`):

$$\text{MACs} = H_{out} \times W_{out} \times C_{out} \times K_H \times K_W \times C_{in}$$

Fill in your values: MACs = `___ × ___ × ___ × ___ × ___ × ___` = `___`

> These numbers are the spec for your accelerator. Write them on paper and keep them handy — every design decision in M04A04 (datapath width, BRAM depth) traces back to them.

The int8 MAC: fixed-point arithmetic

The inner product of two int8 vectors with int32 accumulation:

```
int32_t dot_product_int8(const int8_t *a, const int8_t *b, int n) {
    int32_t acc = 0;
    for (int i = 0; i < n; i++)
        acc += (int32_t)a[i] * (int32_t)b[i];
    return acc;
}
```

Key points:

- `(int32_t)a[i] * (int32_t)b[i]` — both operands promoted to 32-bit before multiply. Without the cast, `int8 * int8` would overflow before accumulation.
- The accumulator is int32 — wide enough for up to 2^{24} int8 × int8 products before overflow ($127 \times 127 \times 2^{24} < 2^{31}$).
- For our layer sizes (max 9 × 64 = 576 MACs per output), there is no overflow risk.

Requantization: int32 accumulator → int8 output

After accumulating, we must scale the int32 result back to int8 range. TFLite uses **per-channel quantization**: each output channel `co` has its own `mult[co]` and `shift[co]`, exported as arrays by the quantizer.

```
// Integer approximation (no float on CPU): acc_scaled = (acc * mult) >> shift
int8_t requantize(int32_t acc, int32_t bias,
                  int32_t mult, int shift, int8_t zero_point) {
    // 1. Add bias
    acc += bias;
    // 2. Multiply-shift approximation of the per-channel scale
    int64_t scaled = ((int64_t)acc * mult) >> shift;
    // 3. Apply ReLU (clamp to ≥ 0) and add output zero-point
    int32_t q = (int32_t)scaled + zero_point;
    // 4. Saturate to int8
    return (int8_t)(q < -128 ? -128 : (q > 127 ? 127 : q));
}
```

`mult` and `shift` come from the per-channel arrays in `weights.h`: `conv_mult[co]`, `conv_shift[co]`. One pair per output channel — not one per layer. This is the standard from the GEMMLOWP / TFLite quantization paper, and it is what makes Stage 4's bit-exact match with TFLM possible.

Conv2D: the inner loop

Standard Conv2D applies each filter across all input channels and spatial positions:

```
void conv2d_int8(
    const int8_t  *input,       // [H][W][C_in]
    const int8_t  *weights,     // [C_out][KH][KW][C_in]
    const int32_t *bias,        // [C_out]
    int8_t        *output,      // [H_out][W_out][C_out]
    int H, int W, int C_in, int KH, int KW, int C_out, int stride,
    const int32_t *mult,        // [C_out] — per-channel multiplier
    const int     *shift,       // [C_out] — per-channel shift
    const int8_t  *zero_point   // [C_out] — per-channel output zero-point
) {
    int H_out = (H - KH) / stride + 1;
    int W_out = (W - KW) / stride + 1;
    for (int oh = 0; oh < H_out; oh++)
    for (int ow = 0; ow < W_out; ow++)
    for (int co = 0; co < C_out; co++) {     // ← one filter per output channel
        int32_t acc = 0;
        for (int kh = 0; kh < KH; kh++)
        for (int kw = 0; kw < KW; kw++)
        for (int ci = 0; ci < C_in; ci++)   // ← cross-channel, cross-spatial
            acc += (int32_t)input[(oh*stride+kh)*W*C_in+(ow*stride+kw)*C_in+ci]
                 * (int32_t)weights[co*KH*KW*C_in+kh*KW*C_in+kw*C_in+ci];
        // per-channel requantization: mult[co] and shift[co] differ per filter
        output[oh*W_out*C_out + ow*C_out + co] =
            requantize(acc, bias[co], mult[co], shift[co], zero_point[co]);
    }
}
```

MAC count per layer

Layer	Output shape	MACs
Conv2D (K×K, N filters, stride S)	H_out×W_out×N	H_out×W_out×N×K×K×C_in
Fully Connected (N×4)	4	4×(H_out×W_out×N)
Total		**» FC (Conv2D dominates)**

The exact numbers depend on the layer dimensions from the reference implementation. The key property: **Conv2D accounts for the vast majority of all MACs** in tiny_conv — the FC layer has negligibly few MACs by comparison.

Your Lab 4 task is to instrument the kernel and verify these numbers on your PC before touching the board.

MAC count — worked example

Using symbolic layer dimensions (replace with your actual values after running `inspect_model.py`):

Assume: `H=49`, `W=40`, `C_in=1`, `KH=3`, `KW=3`, `C_out=8`, `stride=2`.

Then:

- `H_out = (49 - 3) / 2 + 1 = 24`
- `W_out = (40 - 3) / 2 + 1 = 19`
- **Conv2D MACs** = 24 × 19 × 8 × 3 × 3 × 1 = **32,832**

For the FC layer (4 output classes):

- **FC MACs** = 4 × (24 × 19 × 8) = 4 × 3,648 = **14,592**

Conv2D share = 32,832 / (32,832 + 14,592) ≈ **69%**

> Note: your actual numbers will differ depending on the real model configuration. The key takeaway is that Conv2D dominates — and your measured `mac_count` from Lab 4 will confirm this. The ratio Conv2D/(Conv2D+FC) is what Amdahl's law applies to in M04A03.

In-class exercise — trace requantization by hand

Given the following values for one output element:

- Raw accumulator: `acc = 12480`
- Bias for this channel: `bias = −200`
- Per-channel multiplier: `mult = 1234`
- Per-channel shift: `shift = 12`
- Output zero-point: `zero_point = −128`

Follow the steps in `requantize()` from this class:

1. `acc += bias` → `acc = ?`
2. `scaled = (acc * mult) >> shift` → (use integer arithmetic; hint: `12280 × 1234 = 15,153,520`) → `scaled = ?`
3. `q = scaled + zero_point` → `q = ?`
4. Saturate to int8: is `q` in [-128, 127]? → `result = ?`

> Expected: `acc = 12280`, `scaled = 15,153,520 >> 12 = 3,699`, `q = 3,699 + (−128) = 3,571`. Since 3,571 > 127, saturation clamps to **127**. This illustrates how an outlier activation saturates — and why QAT (quantization-aware training) is important: without it, many outputs would saturate and accuracy would drop.

Counting MACs: your Lab 4 task

Lab 4 asks you to instrument the kernel and verify these numbers:

```
// Add to each loop body:
static uint64_t mac_count = 0;

// Inside the innermost loop of each layer function:
mac_count++;

// After each layer:
uart_printf("conv2d: %llu MACs\r\n", mac_count);
mac_count = 0;
```

Also measure cycle counts around each layer (using `read_cycle()`). The ratio `cycles / MACs` tells you how many cycles you spend per MAC — and how much room there is for improvement.

Building and running on PC and board

```
# PC (fast development loop)
gcc -O2 -DPLATFORM_PC -o kws_test kws_kernel.c
./kws_test
# Output: class=yes score=112 mac_count=7328256 cycles=N/A

# Board (real measurement)
riscv64-unknown-elf-gcc -march=rv32im -mabi=ilp32 \
    -nostdlib -Wl,-Ttext=0x100 -O2 \
    -o kws.elf kws_kernel.c
riscv64-unknown-elf-objcopy -O ihex kws.elf kws.hex
cat kws.hex > /dev/ttyUSB0
# Read UART: "yes 3847291 cycles"
```

The `PLATFORM_PC` preprocessor switch redirects `read_cycle()` to `clock()` and `uart_putc()` to `putchar()` — the same kernel source compiles for both targets.

Dependency injection via function pointers

The KWS inference kernel (`kws_kernel.c`) must run in two environments:

1. **On the PC** (for testing against TFLite reference): uses `clock()` to measure time, no hardware accelerator
2. **On the Tang Nano 9K** (in production): uses CSR cycle counter, optionally calls the hardware accelerator

Instead of `#ifdef PC_TEST`, the kernel accepts its platform dependencies as function pointers:

```
// kws_kernel.h
typedef uint64_t (*kws_cycle_fn_t)(void);
typedef void     (*kws_accel_write_fn_t)(uint32_t addr, uint32_t val);
typedef uint32_t (*kws_accel_read_fn_t)(uint32_t addr);

void kws_init(kws_cycle_fn_t    cycle_fn,
              kws_accel_write_fn_t write_fn,
              kws_accel_read_fn_t  read_fn);
```

On PC: `kws_init(pc_cycle, NULL, NULL)` — cycle counter uses `clock()`, no accelerator.
On board: `kws_init(board_cycle, mmio_write, mmio_read)` — uses CSR + hardware.

This pattern (dependency injection) keeps `kws_kernel.c` free of any `#include <hardware.h>` — it compiles and tests identically on both platforms.

Next class

Profiling: instrument the kernel, collect per-layer cycle counts on the board, confirm that Conv2D dominates, compute arithmetic intensity, and decide which operation to accelerate — and why hardware wins.