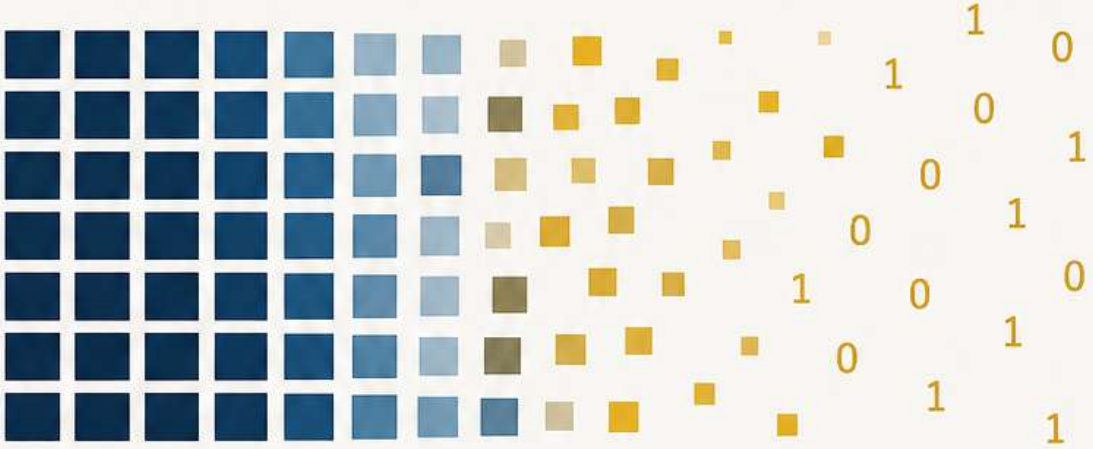
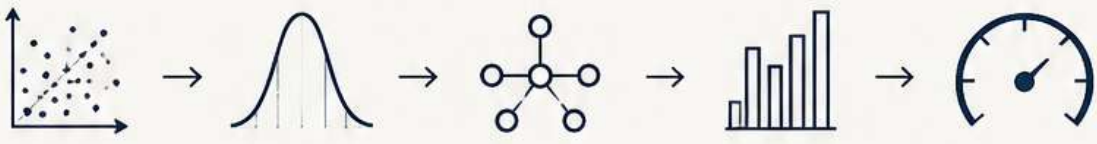
LLM QUANTIZATION

FROM THE BITS UP

PRINCIPLES. ALGORITHMS. IMPLEMENTATIONS.
REAL NUMBERS. NO BLACK BOXES.





BUILD IT. BREAK IT. MEASURE IT.

HATEM M.

LLM Quantization: From the Bits Up — Book Four.

This book develops the theory and practice of large language model quantization entirely from first principles. Every mechanism is built from scratch in tested Python and PyTorch, deliberately pushed to its breaking point, and measured with real numbers produced by executed code. No result in these pages is asserted without a measurement behind it.

The reference model throughout is TinyGPT, a small GPT-style decoder trained from scratch on the works of Shakespeare, chosen so that every experiment is reproducible on a single CPU core.

Built with the method: **build it, break it, measure it.**

Free Sample

This excerpt contains the complete
Part I — Foundations (Chapters 0–3)
of a seventeen-chapter book.

What's in the full book

The complete book builds large language model quantization from first principles — every mechanism coded from scratch, deliberately broken, and measured with real numbers. Six parts, seventeen chapters:

Part I — Foundations *(included in this sample)*

0 Notation, Tools, and the Baseline
1 The Economics of Bits
2 The Quantization Map
3 Granularity

Part II — The Mathematics of Loss

4 Quantization Noise (the $\Delta^2/12$ law, 6.02 dB per bit)
5 Error Propagation (why output error, not weight error, is what matters)

Part III — The Outlier Problem

6 Weights versus Activations
7 Emergent Outliers
8 Taming Outliers: LLM.int8() and SmoothQuant

Part IV — The PTQ Algorithms

9 RTN and Calibration
10 GPTQ (derived from Optimal Brain Surgeon, implemented from scratch)
11 AWQ and Rotation

Part V — Representation and Kernels

12 GGUF K-Quants, Byte by Byte (the llama.cpp format, fully reconstructed)
13 Dequantization and Direct Block Matmul

Part VI — Measuring and Breaking

14 The Quality Cliff
15 KV-Cache Quantization
16 Capstone: Quantizing a Model End to End

The four chapters that follow are the complete Part I, exactly as they appear in the full book.

PART I

Foundations

CHAPTER 0

Notation, Tools, and the Baseline

This book has exactly one subject: how the numbers inside a large language model are stored in very few bits without destroying the model. Everything that follows is in service of one recurring question.

For every technique in this book we answer three things, and only these three:

(a) **How many bits did we save?**

(b) **How much accuracy did we lose** — measured, never asserted?

(c) **What is the exact mechanism** linking the two — the mathematics of the loss?

If a paragraph does not move one of these forward, it does not belong here.

This short chapter fixes three things so the rest of the book can be terse: the *notation* we use for quantization, the *tools and model* every experiment runs on, and the *baseline numbers* that every later result is a delta against. No technique is introduced yet — this is the ruler, not the measurement.

0.1 Notation

We will be ruthless about symbols, because quantization papers are not, and the inconsistency is half of why the field is hard to read. The following hold for the entire book unless a chapter explicitly overrides them.

Symbol	Meaning
x	a real-valued (FP32) number we wish to store
b	the bit-width of the integer target (e.g. $b = 4$ for INT4)
s	the *scale*: size of one quantization step, $s \in \mathbb{R}_{>0}$
z	the *zero-point*: the integer code that represents 0.0
$Q(x)$	the quantized integer code of x
$\hat{x}$	the reconstructed (dequantized) real value, $\hat{x} \approx x$
$q_{\min}, q_{\max}$	the representable integer range (e.g. $[-8, 7]$ for signed INT4)
round$(\cdot)$	round-to-nearest-integer (ties handled per §2.3)
clip$(\cdot)$	saturate into $[q_{\min}, q_{\max}]$
$\mathbf{W}$	a weight matrix, shape $(d_{\text{out}}, d_{\text{in}})$
$\mathbf{x}$	an activation vector entering a layer
σ	standard deviation of a value distribution
Δ	quantization step when used as a noise quantity ($\Delta = s$)

The one equation the whole book orbits is the **affine quantization map**, derived properly in Chapter 2:

$$Q(x) \;=\; \text{clip}\Big(\,\text{round}\big(\tfrac{x}{s}\big) + z,\; q_{\min},\; q_{\max}\Big), \qquad \hat{x} \;=\; s\,(Q(x) - z). \tag{1}$$

Read it once now; we will spend Chapter 2 earning every term of it.

Measurement symbols. Accuracy is reported as **perplexity** (PPL, lower is better). Representation fidelity is reported as **mean-squared error** (MSE) and **signal-to-quantization-noise ratio** (SQNR, in decibels, higher is better), both defined in Chapter 4. Size is reported in mebibytes (MiB $= 2^{20}$ bytes). Speed is reported in decode **tokens per second**.

0.2 Tools

The book is written in **Python + PyTorch**, not C++, so it reaches the whole ML audience. Every listing in this book was executed; every number in a MEASURED box is the real output of the code beside it. The environment is deliberately humble, which is a feature: if it runs here, it runs anywhere.

Component	Version / value
Python	3.12
PyTorch	2.12 (CPU build)
Hardware	single CPU core (no GPU)
Model	`TinyGPT` — a from-scratch GPT-2-style decoder
Corpus	tiny-shakespeare (1,115,394 characters, char-level)

Why a from-scratch 1.8M model instead of GPT-2 or Llama?

Three reasons, in priority order. **(1) Transparency.** The book's method is *build it, break it, measure it.* A model we wrote line-by-line has no black box; when we quantize a weight matrix you can see exactly which matrix. **(2) Reproducibility.** A reader can retrain this model from a 1 MB text file and a fixed seed in minutes, then rerun every experiment. **(3) Speed.** Quantization is studied by sweeping design choices — block size, group size, bit-width — hundreds of times. A small model makes that practical.
The cost is honest and stated up front: a 1.8M-parameter model does *not* exhibit the **emergent activation outliers** that appear in models above roughly 6B parameters. Those outliers are the entire subject of Chapters 7–8, so for exactly those chapters (and the capstone) we switch to a real pretrained model and report its real numbers. Everywhere else, TinyGPT is not a compromise — it is the better teaching instrument.

0.2.1 The model, exactly

`TinyGPT` is GPT-2's architecture with the scale dialed down: learned absolute position embeddings, pre-LayerNorm blocks, causal multi-head attention with a single fused QKV projection, a 4× GELU MLP, and tied input/output embeddings. Nothing about the architecture is novel — that is the point. The configuration:

Hyper-parameter	**Value**
layers (n_{layer})	4
heads (n_{head})	4
embedding width (n_{embd})	192
head dimension	48
context length	128
vocabulary	65 (characters)
total parameters	**1,816,896**

0.3 The baseline

We trained `TinyGPT` on tiny-shakespeare until the held-out loss stopped improving. The resulting weights are *frozen* as the book's ground truth: every quantized variant in every later chapter is compared against this exact checkpoint.

Quantity	Value	Notes
Parameters	1,816,896	all weights
Validation loss	1.7014	cross-entropy, training-time est.
FP32 size	6.9309 MiB	4 bytes/param
FP16 size	3.4655 MiB	2 bytes/param
INT8 size (raw)	1.7327 MiB	1 byte/param, scales extra
INT4 size (raw)	0.8664 MiB	0.5 byte/param, scales extra
Perplexity (FP32)	**5.4497**	sliding window, ctx 128, stride 64
Perplexity (FP16)	5.4497	identical — see note
Decode throughput	180.4 tok/s	batch 1, CPU, no KV cache
Weight std σ	0.037615	linear layers
Weight absmax	0.302281	linear layers
absmax/4σ	2.01	first sign of outliers

Three of these numbers are worth pausing on, because each previews a theme of the book.

FP16 equals FP32 here — but it will not always. The validation perplexity is identical to four decimals in FP32 and FP16. That is not luck: every weight in this model lives comfortably inside FP16's dynamic range, so casting loses nothing measurable. The lesson is *not* "FP16 is free." It is that a format is only as safe as its range covers your values. In Chapter 6 we meet *activations*, whose range does not fit so politely, and the equality breaks.

Why FP16 was slower to measure, and why we ignore that. Computing the FP16 perplexity took roughly eight times longer than FP32 on this machine. That is a property of a *CPU without native half-precision arithmetic* — it emulates FP16 in software. It says nothing about FP16 on a GPU, where the opposite holds. We flag it here so no reader mistakes a hardware artifact for a property of the number format. Throughout the book, **speed claims are tied to the hardware that produced them**, never stated in the abstract.

The outliers are already here. Look at the last row: absmax/4σ = 2.01. If the weights were perfectly Gaussian, essentially all of them would fall within $\pm 4\sigma$. Instead the most extreme weight sits at *twice* that distance. A handful of rare large values stretch the range that any fixed-point format must cover — and since the step size s is proportional to that range (Eq. 1), those few values make every *other* weight coarser. This is the seed of the outlier problem that dominates Part III. Figure 1 shows it directly.

0.4 How to read this book

Every chapter follows the same rhythm, the author's signature method:

1. **Build it** — from scratch, in tested Python. No library call stands in for understanding. (We may show the production library *after* building the thing ourselves, to connect theory to practice.)

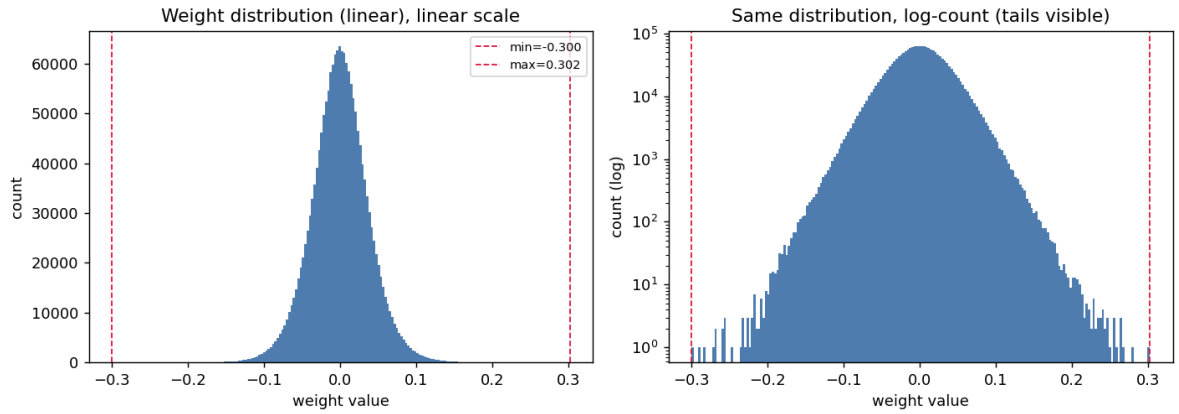


Figure 1: The empirical weight distribution of the baseline model (all linear layers, 1,769,472 weights). **Left:** on a linear count axis the distribution looks like a clean zero-mean bell; 99.8% of weights fall within ± 0.139. **Right:** the same data on a log-count axis exposes the tails — a sparse population of weights reaches ± 0.30, roughly $\pm 8\sigma$. These rare values, not the bulk, set the quantization range. The whole of Part III is about not letting them dictate the precision of everything else.

2. **Break it** — push the technique until it fails, on purpose, to reveal exactly why the next, more sophisticated method has to exist. These appear in WHEN IT BREAKS boxes.
3. **Measure it** — quantify everything: bits saved, accuracy lost, memory, speed. These appear in MEASURED boxes, and every number in them came from a real run.

What this book is *not*: it is not a survey, not an introduction to machine learning, and not a catalogue of libraries. It does not cover quantization-aware *training*, mixed-precision *training*, pruning, sparsity, or distillation — those are different subjects. It covers one thing, post-training quantization for LLM *inference*, all the way to the floor: the affine map, the error mathematics, the outlier problem, the PTQ algorithms (RTN, GPTQ, AWQ), the on-disk formats byte-by-byte (GGUF K-quants), the kernels that make it fast, and the precise points where it all collapses.

With the ruler defined and the ground truth frozen, we can begin. Chapter 1 asks the only question that justifies the entire enterprise: *why bother quantizing at all?* The answer is not what most people assume.

CHAPTER 1

The Economics of Bits

Before we change a single weight, we have to answer the question that justifies the entire book: *why quantize at all?* The usual answer — "to make the model smaller" — is true but shallow, and it hides the real mechanism. The honest answer is sharper and, to most people, surprising:

The thesis of this chapter

During autoregressive decoding, a large language model is **not limited by how fast the hardware can compute**. It is limited by how fast the hardware can **read the weights out of memory**. Quantization is the lever that matters because it attacks that exact bottleneck: fewer bits per weight means fewer bytes to move, and *decode latency is set by bytes moved, not by arithmetic done.*

We will not assert this. We will derive it from first principles, then measure its fingerprint on real hardware, then show the payoff as a concrete latency formula. By the end of the chapter the rest of the book has a reason to exist.

1.1 The memory wall, not the compute wall

Every linear layer in a transformer computes $\mathbf{y} = \mathbf{W}\mathbf{x}$, where $\mathbf{W}$ has shape $(d_{\text{out}}, d_{\text{in}})$. To run it, the hardware must do two physically distinct things: *move* the numbers from memory into the compute units, and *compute* the multiply-adds. The relative cost of these two is captured by one quantity.

Definition 1.1 (Arithmetic intensity)**.** The **arithmetic intensity** of an operation is the ratio of floating-point operations performed to bytes of memory traffic required:

$$I = \frac{\text{FLOPs}}{\text{bytes moved}} \qquad [\text{FLOP/byte}].$$

Let us compute it for a linear layer at batch size B (in decoding, B is the number of sequences generated in parallel; for a single user, $B = 1$).

Derivation 1.1 (Arithmetic intensity of a linear layer)**.** *The matrix–vector (or matrix–matrix) product touches:*

$$FLOPs = 2\, d_{out}\, d_{in}\, B \quad \textit{(one multiply-add per weight, per token)}$$

$$bytes = \underbrace{d_{out}\, d_{in} \cdot c_w}_{weights} + \underbrace{d_{in}\, B \cdot c_a}_{input} + \underbrace{d_{out}\, B \cdot c_a}_{output}$$

where c_w is bytes per weight and c_a is bytes per activation. For the small batch sizes that dominate interactive inference, the weight term swamps the activation terms ($d_{out}d_{in} \gg d_{in}B + d_{out}B$ when B is small), so

$$I \approx \frac{2\, d_{out}\, d_{in}\, B}{d_{out}\, d_{in}\, c_w} = \boxed{\frac{2B}{c_w}}.$$

This result is worth staring at. The layer's dimensions *cancelled*. The arithmetic intensity of a linear layer at small batch depends on only two things: the batch size and the bytes per weight. Table values for $B = 1$:

Precision	**bytes/weight** c_w	**intensity** $I = 2/c_w$
FP32	4	0.5 FLOP/byte
FP16	2	1.0 FLOP/byte
INT8	1	2.0 FLOP/byte
INT4	0.5	4.0 FLOP/byte

Now we need something to compare these against. A processor has a peak compute rate (FLOP/s) and a peak memory bandwidth (byte/s). Their ratio is the **ridge point** of the roofline model: the arithmetic intensity at which a workload transitions from memory-bound to compute-bound.

$$I^\star = \frac{\text{peak FLOP/s}}{\text{peak bytes/s}}.$$

For published accelerator specifications (these are vendor numbers, not measured here):

Accelerator	**peak FLOP/s**	**peak BW**	**ridge** $I^\star$
NVIDIA A100 (FP16)	312 TFLOP/s	2039 GB/s	153 FLOP/byte
NVIDIA H100 (FP16)	990 TFLOP/s	3350 GB/s	296 FLOP/byte
Apple M2 (FP16)	7 TFLOP/s	100 GB/s	70 FLOP/byte

Here is the punchline. A linear layer at $B = 1$ has intensity around 0.5 to 4 FLOP/byte. The hardware does not become compute-bound until intensity reaches *one to three hundred*. So single-stream decoding sits **two to three orders of magnitude** inside the memory-bound region. The compute units spend almost all their time waiting for weights to arrive. Figure 1.1 places every precision on the A100 roofline.

1.2 Measuring the bottleneck

A derivation is a claim about the world; this book's standard is to check it against the world. Memory-bound execution has a clean experimental fingerprint, and we can measure it directly.

The fingerprint. If an operation is limited by reading **W** from memory, then processing more tokens *through the same weights* is nearly free: the weights are read once and reused. So as we increase the batch size B, the total time per forward pass should grow only slowly, which means the *per-token* time should fall roughly as $1/B$ — until B grows large enough that compute finally becomes the limit. If instead the operation were compute-bound from the start,

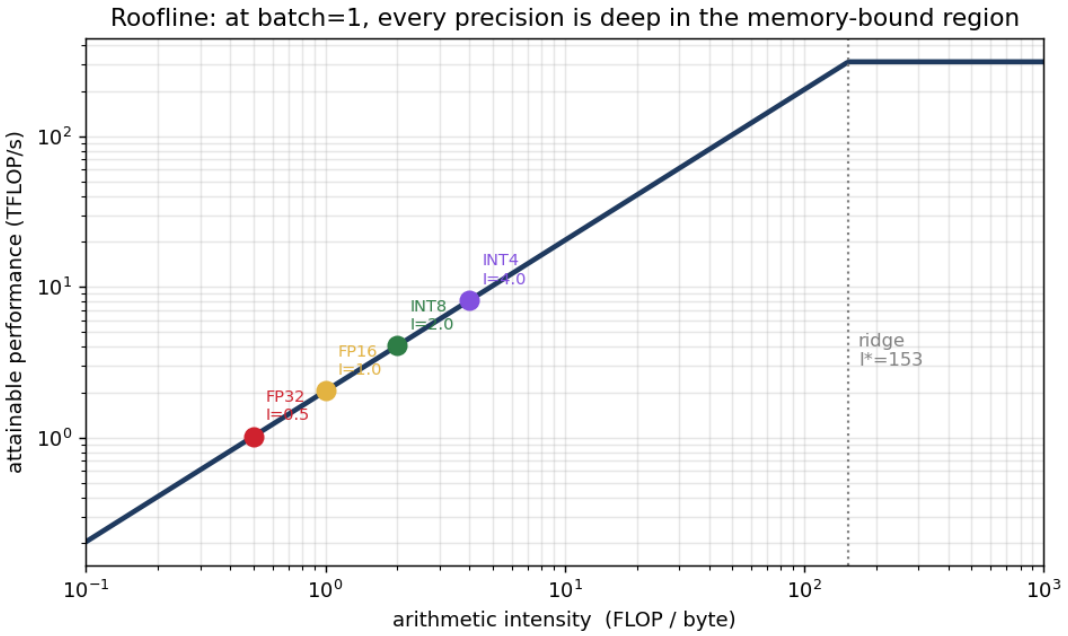


Figure 1.1: The roofline model with an A100's published peak compute and bandwidth. The sloped line is the memory-bound ceiling (performance = bandwidth × intensity); the flat line is the compute ceiling. Every batch-1 operating point — FP32, FP16, INT8, INT4 — lands far down the *sloped* part, nowhere near the ridge at $I^\star = 153$. Two consequences follow. First, at batch 1 you are wasting essentially all of the chip's arithmetic capacity. Second, and this is the whole point of the book: moving from FP16 toward INT4 slides the operating point *up the sloped line*, because halving the bytes per weight doubles the attainable performance in the regime where you actually live.

total time would grow linearly with B and per-token time would stay flat. The shape of the per-token curve is therefore a direct test of which regime we are in.

We time a single 4096×4096 linear layer — the dominant operation inside a transformer block — at increasing batch sizes on the book's machine (one CPU core, FP32).

Listing 1.1: The batch-scaling probe (core of `ch1_experiments.py`).

```
d = 4096
lin = torch.nn.Linear(d, d, bias=False).eval()

def time_batch(B, iters=10):
    x = torch.randn(B, d)
    with torch.no_grad():
        lin(x)                          # warmup
        t0 = time.time()
        for _ in range(iters):
            lin(x)
        return (time.time() - t0) / iters

for B in [1, 2, 4, 8, 16, 32, 64]:
    dt = time_batch(B)
    print(B, dt*1e3, (dt/B)*1e3, B/dt)   # batch, total_ms, per_tok_ms, tok/s
```

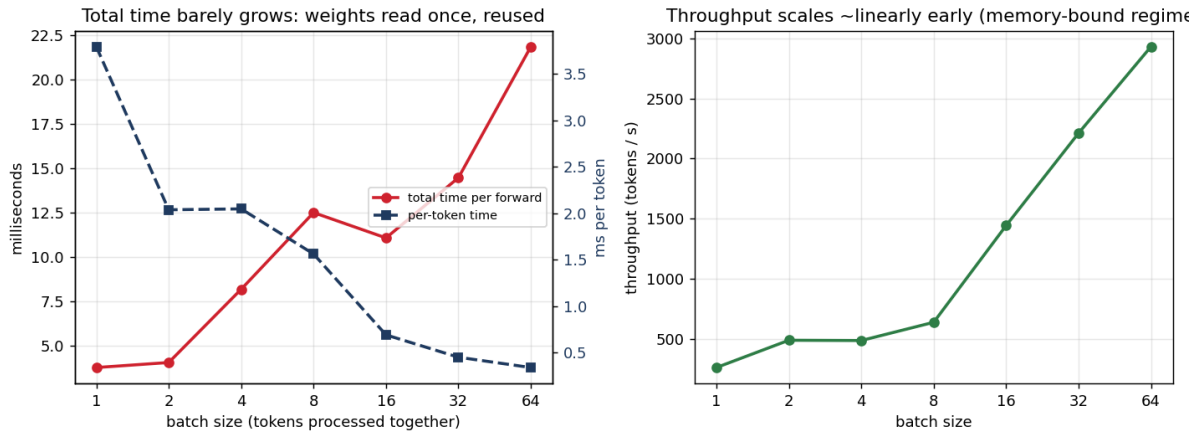


Figure 1.2: Left: total time per forward pass (red, left axis) grows slowly while per-token time (blue dashed, right axis) collapses — the signature of a memory-bound operation amortizing its weight reads. **Right:** throughput climbs steeply with batch in the memory-bound regime. For a single interactive user, however, $B = 1$ is forced, and the system is pinned at the worst point on these curves. That is the regime quantization targets.

batch	total (ms)	per-token (ms)	tok/s	speedup vs. B=1
1	3.80	3.7952	263.5	1.00×
2	4.08	2.0394	490.3	1.86×
4	8.20	2.0492	488.0	1.85×
8	12.51	1.5636	639.6	2.43×
16	11.08	0.6928	1443.5	5.48×
32	14.47	0.4522	2211.4	8.39×
64	21.84	0.3412	2930.4	11.12×

The per-token time falls **11.1**× going from batch 1 to batch 64. The compute per token is identical across every row — the same multiply-adds happen for each token regardless of batch — so an *eleven-fold* drop in per-token time cannot be a compute effect. It is the weight reads being amortized over more tokens. At batch 1, those reads *were* the bottleneck, exactly as the intensity argument predicted. Figure 1.2 shows the curves.

How fast can this machine actually move bytes? The latency floor of a memory-bound workload is set by bandwidth, so we measure it. A triad ($a = b + c$ over arrays too large to cache) reads two arrays and writes one:

A 64M-element FP32 triad moved $3 \times 64\text{M} \times 4 = 768$ MB in a best-of-5 pass, giving an effective bandwidth of **14.55 GB/s**. (A server GPU is 100–200× this; the *mechanism* is identical, only the constant changes.)

1.3 The arithmetic of shrinking

We now have everything needed to state the payoff exactly. If decode is memory-bound, then the minimum possible time to produce one token is the time to stream the weights through memory once:

Result 1.1 (Memory-bound latency floor)**.** For a model with N parameters at c_w bytes per weight, on hardware with memory bandwidth β bytes/s, the per-token decode latency cannot beat

$$t_{\text{floor}} = \frac{N \cdot c_w}{\beta}.$$

Latency is *linear* in bytes per weight. Halving c_w halves the floor.

This is the entire economic case for quantization in one equation. It does not depend on a clever kernel; it is a property of physics and the memory system. Quantization is the only lever that reduces c_w, so it is the only lever that lowers this floor. We evaluate it for the book's baseline model (using *this machine's* measured 14.55 GB/s) and for a 7B-parameter model (using an A100's published 2039 GB/s, since 7B will not run here):

Precision	**TinyGPT 1.8M (measured 14.55 GB/s)**	**7B model (A100 spec, 2039 GB/s)**
FP32	0.500 ms/tok	13.732 ms/tok
FP16	0.250 ms/tok	6.866 ms/tok
INT8	0.125 ms/tok	3.433 ms/tok
INT4	0.062 ms/tok	1.717 ms/tok

For the 7B model, the floor falls from 6.9 ms/token in FP16 to 1.7 ms/token in INT4 — a 4× speedup, available before writing a single line of kernel code, purely from moving fewer bytes. In throughput terms that is the difference between roughly 146 and 583 tokens per second. This is why every serving stack in production — llama.cpp, vLLM, TensorRT-LLM — quantizes: not to save disk, but to break the bandwidth wall.

And the disk savings are real too. The same shrinkage decides whether a model *fits* at all. A 7B model needs roughly 28 GB in FP32 and 14 GB in FP16 — already past a 12 GB consumer GPU. At INT4 it is about 3.5 GB, which fits with room for the KV cache. Quantization is frequently the difference between "runs on the user's laptop" and "does not run."

Model	**FP32**	**FP16**	**INT8**	**INT4**
TinyGPT (1.8M)	6.93 MiB	3.47 MiB	1.73 MiB	0.87 MiB
7B	28.0 GB	14.0 GB	7.0 GB	3.5 GB
70B	280 GB	140 GB	70 GB	35 GB

(These are raw weight bytes; the true on-disk size is slightly larger because quantization formats store scales and zero-points alongside the codes. Exactly how much larger, and how those extra bytes are packed, is the subject of Chapter 12.)

1.4 What this buys, and what it costs

The benefit is now quantified and mechanistic: a latency floor and a footprint that both fall linearly with bytes per weight. But this is a book about a *tradeoff*, and we have only shown the upside. The bill comes due in accuracy. Reducing c_w means representing each weight with fewer distinct values, and fewer values means each stored weight is further from its true FP32 value. Those errors propagate through every layer and show up, eventually, as higher perplexity — or, past a point, as a model that produces garbage.

This is the question the rest of the book answers, technique by technique:

(a) **bits saved** — quantified in this chapter: latency and size fall linearly in c_w;

(b) **accuracy lost** — measured for every method, starting in Chapter 4;

(c) **the mechanism** — the mathematics connecting the two, which is where we go next.

We have established *why* the prize is worth chasing. Chapter 2 begins the real work: the precise map from a real number to an integer code and back, every term derived — because you cannot analyze an error you cannot write down.

CHAPTER 2

The Quantization Map

Chapter 1 proved *why* we want to store weights in fewer bits. This chapter builds the machine that does it, and — because this is a book about where the accuracy goes — it does so in a way that makes every source of error explicit and measurable. You cannot analyze an error you cannot write down, so we write down the map first, derive every term, and only then quantize the real model and watch the perplexity move.

What a quantizer is, exactly

A quantizer is a pair of functions. The *quantize* function Q maps a real number to one of a small, fixed set of integer codes. The *dequantize* function maps a code back to a real number. The composition $\hat{x} = \text{dequant}(Q(x))$ is not the identity — the gap $x - \hat{x}$ is the quantization error, and the entire book is the study of that gap.

2.1 Deriving the affine map

We want to represent real numbers from some range $[r_{\min}, r_{\max}]$ using integer codes from $[q_{\min}, q_{\max}]$. The simplest faithful relationship is *affine* (linear plus offset): a real value x and its code q are related by

$$x \;=\; s\,(q - z), \tag{2.1}$$

where $s > 0$ is the **scale** (the real-world size of one integer step) and z is the **zero-point** (the integer code that represents real zero). Everything follows from forcing the endpoints to correspond.

Derivation 2.1 (Scale and zero-point)**.** *Demand that the real endpoints map to the integer endpoints:*

$$r_{\min} = s\,(q_{\min} - z), \qquad r_{\max} = s\,(q_{\max} - z).$$

Subtract the first from the second; the z terms cancel and we can solve for the scale:

$$s \;=\; \frac{r_{\max} - r_{\min}}{q_{\max} - q_{\min}}. \tag{2.2}$$

Now solve either equation for z. From the first, $q_{\min} - z = r_{\min}/s$, hence

$$z \;=\; \text{round}\Big(q_{\min} - \frac{r_{\min}}{s}\Big). \tag{2.3}$$

The rounding is essential: z must be an integer *code so that real zero lands exactly on a representable value, with no error. To go from a real x to its code we invert Eq. (2.1), round to the nearest integer, and clip into the valid range:*

$$Q(x) \;=\; \text{clip}\Big(\,\text{round}\big(\tfrac{x}{s}\big) + z,\; q_{\min},\; q_{\max}\Big). \tag{2.4}$$

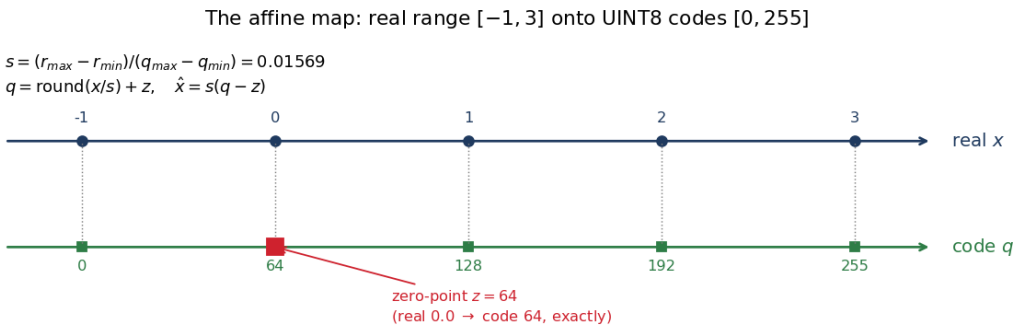


Figure 2.1: The affine map for $[-1, 3] \to [0, 255]$. Real values (top, blue) and their integer codes (bottom, green) are tied by $x = s(q - z)$. The zero-point $z = 64$ guarantees that real 0.0 has an exact code; this matters because zeros are common in neural networks (padding, masked positions, dead ReLUs) and a quantizer that cannot represent zero exactly injects error everywhere one appears.

Equations (2.2)–(2.4) are the whole map. Figure 2.1 shows it concretely for the range $[-1, 3]$ onto unsigned 8-bit codes: the scale is $4/255 \approx 0.0157$, and real zero lands exactly on code 64.

2.1.1 The reference implementation

We build this from scratch — no library quantizer stands in for understanding. The two parameter-choosing functions are direct transcriptions of Eqs. (2.2)–(2.3):

Listing 2.1: Choosing scale and zero-point (from `quant.py`).

```
def choose_qparams_affine(rmin, rmax, bits=8, signed=False):
    qmin, qmax = int_range(bits, signed)
    rmin = min(rmin, 0.0)            # ensure 0 is inside the range...
    rmax = max(rmax, 0.0)            # ...so zero stays representable
    s = (rmax - rmin) / (qmax - qmin)          # Eq. (2.2)
    s = max(s, 1e-12)
    z = round(qmin - rmin / s)                 # Eq. (2.3)
    z = int(min(max(z, qmin), qmax))
    return s, z
```

2.2 Symmetric versus asymmetric

The map above is **asymmetric**: it uses the true range $[r_{\min}, r_{\max}]$ and produces a generally nonzero zero-point. There is a simpler, important special case.

Definition 2.1 (Symmetric quantization)**.** Fix the real range to be symmetric about zero, $[-\alpha, +\alpha]$ with $\alpha = \max(|r_{\min}|, |r_{\max}|) = \text{absmax}$. Then real zero already sits at the center, the zero-point is $z = 0$, and the map collapses to a pure scaling

$$x = s\,q, \qquad s = \frac{\alpha}{2^{b-1} - 1}.$$

For signed INT8 the denominator is 127.

Symmetric quantization has two practical virtues. There is no zero-point to store (saving memory and an addition in the inner loop), and real zero is exact for free. Its cost is range efficiency: if the data is lopsided, symmetric quantization must stretch its range to ±absmax and *wastes all the codes on the empty side*. Whether that cost matters depends entirely on the data — which is exactly what we now measure.

2.2.1 On real weights: a near tie

The baseline model's linear weights are centered and roughly symmetric (we saw this in Chapter 0: mean $\approx 5 \times 10^{-5}$, a clean bell). So we should expect symmetric and asymmetric to perform almost identically. They do.

bits	scheme	MSE	SQNR (dB)	max err
8	sym	4.716×10^{-7}	34.77	0.0012
8	asym	4.642×10^{-7}	34.84	0.0012
4	sym	1.551×10^{-4}	9.60	0.0216
4	asym	1.342×10^{-4}	10.23	0.0201
2	sym	1.403×10^{-3}	0.04	0.1511
2	asym	1.304×10^{-3}	0.35	0.1004

Two things to notice, both of which echo through the book. First, the two schemes are within a few percent on this centered data — asymmetric's advantage is real but small when the distribution is already symmetric. Second, look at the SQNR column: it falls by roughly 25 dB from INT8 to INT4, i.e. about 6.25 dB for each of the 4 bits dropped. That "6 dB per bit" is not a coincidence; it is a theorem we derive in Chapter 4. Here it simply shows up in the measurements, unbidden.

2.2.2 Breaking symmetric: real activations

To see asymmetric earn its keep, we need lopsided data — and the network is full of it. The output of a GELU nonlinearity is bounded below near -0.17 but has a long positive tail. We capture the *real* GELU activations from the model running on validation text and quantize them both ways.

Distribution: min $= -0.170$, max $= 4.504$, mean $= -0.012$, with 81.4% of values negative but a positive tail reaching 4.5 — strongly skewed.

	MSE	SQNR (dB)	step s	zero-point z
INT8 symmetric	1.109×10^{-4}	29.60	0.03546	0
INT8 asymmetric	2.570×10^{-5}	35.95	0.01833	-119
INT4 symmetric	1.865×10^{-2}	7.34	0.64339	0
INT4 asymmetric	1.305×10^{-2}	8.89	0.31158	-7

When it breaks

On this skewed tensor, symmetric INT8 has **4.31**× the MSE of asymmetric INT8. The mechanism is visible in the step sizes: symmetric is forced to cover $[-4.5, +4.5]$, giving a coarse step $s = 0.0354$, while asymmetric covers only the occupied range $[-0.17, 4.5]$ with a step nearly half as large, $s = 0.0183$. Symmetric spends roughly half of its 256 codes on negative values that essentially never occur. *This is why activations are quantized asymmetrically while weights often are not* — a theme we develop fully in Chapter 6.

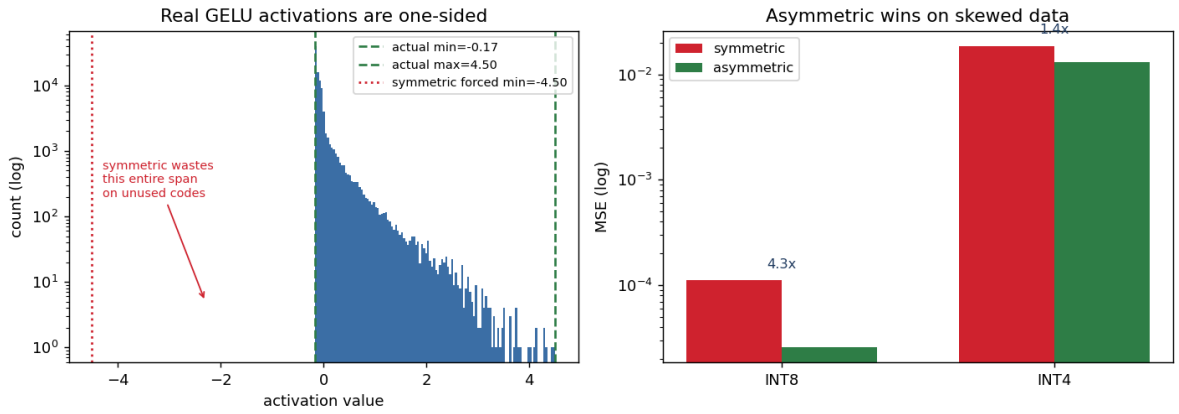


Figure 2.2: **Left:** the real GELU activation distribution is one-sided; the green lines mark its true support $[-0.17, 4.50]$, while the red line marks where symmetric quantization is forced to place its lower bound (-4.50). The shaded gap is pure waste. **Right:** the resulting MSE, symmetric versus asymmetric, at INT8 and INT4 — asymmetric is 4.3× better at INT8 on this data.

2.3 Rounding modes

Equation (2.4) contains a round$(\cdot)$. The obvious choice is round-to-nearest, but it is not the only one, and the differences are not cosmetic. We compare three modes on the real weights, quantized to INT4.

- **Round-to-nearest (RTN).** Map to the closest code. The error per value is bounded by half a step and, over a smooth distribution, is nearly mean-zero.
- **Truncation (floor).** Drop the fractional part. Fast, but it *always rounds down*, so it injects a systematic negative bias of about half a step into every value.
- **Stochastic rounding.** Round up with probability equal to the fractional part. Unbiased in expectation, at the cost of higher variance.

mode	MSE	bias (mean error)
nearest	1.551×10^{-4}	-1.19×10^{-6}
floor	6.219×10^{-4}	-2.16×10^{-2}
stochastic	3.104×10^{-4}	$+6.16 \times 10^{-6}$

The numbers tell the story cleanly. Truncation has 4× the MSE of RTN and a bias four orders of magnitude larger — that systematic offset is poison in a deep network, where biases compound layer over layer (Chapter 5). RTN and stochastic rounding are both essentially unbiased, but RTN has half the MSE of stochastic here, because adding randomness adds variance. **For post-training quantization, round-to-nearest is the default, and for good reason.** (Stochastic rounding earns its place in quantization-aware *training*, where unbiasedness across many gradient steps matters more than per-step variance — but that is a different book.)

2.4 Clipping: the range is a choice

So far we set the range to the data's exact extremes (absmax, or $[r_{\min}, r_{\max}]$). But the extremes may be rare outliers, and covering them forces a coarse step on the common values. We can

instead *clip*: shrink the range to some fraction of absmax, accept that the few values outside get saturated, and reap a finer step for everyone else. This is a genuine trade-off — clipping error versus rounding error — and it has an optimum.

clip ratio	step s	MSE	SQNR (dB)
1.00	0.04318	1.551×10^{-4}	9.60
0.90	0.03886	1.257×10^{-4}	10.51
0.80	0.03455	9.938×10^{-5}	11.53
0.70	0.03023	7.609×10^{-5}	12.69

Clipping to 70% of absmax *halves* the MSE versus using the full range, a 3 dB SQNR gain, purely by refusing to let a handful of extreme weights dictate the step size for the other 99.9%. We do not search for the true optimum here — doing that well, against a calibration set rather than the weights' own MSE, is the subject of Chapter 9. The point for now is structural: **the quantization range is a free parameter, and the naive choice** (absmax) **is rarely the best one.** This single observation is the seed of every calibration method in the book.

2.5 The first model-level verdict

Per-value MSE is the microscope; perplexity is the verdict. We now quantize *every* linear weight in the model (per-tensor) and measure the real validation perplexity — the first time in the book we answer "where did the accuracy go?" at the level of the whole model.

Listing 2.2: Quantizing the whole model, then measuring (from `ch2_experiments.py`).

```
def quantized_model(bits, symmetric):
    m = TinyGPT(cfg); m.load_state_dict(ck["model"]); m.eval()
    with torch.no_grad():
        for n, p in m.named_parameters():
            if p.dim() == 2 and "wte" not in n and "wpe" not in n:
                xh, _, _ = fake_quant_tensor(p.data, bits=bits,
                                             signed=True, symmetric=symmetric)
                p.data.copy_(xh)            # replace weight with its dequantized
    value
    return m
```

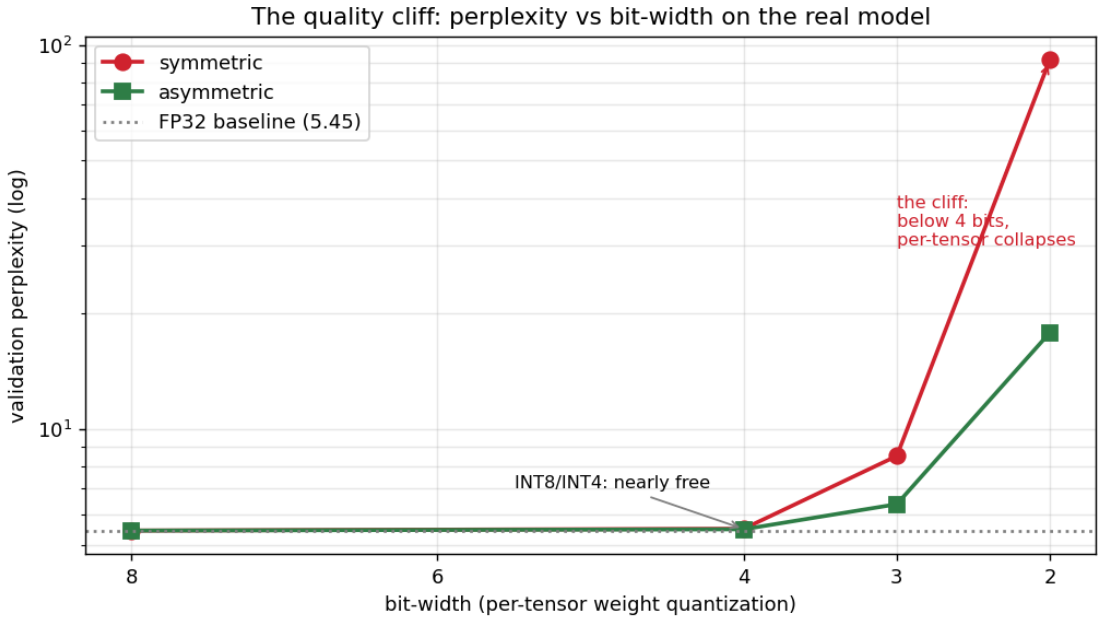


Figure 2.3: Validation perplexity versus bit-width for full-model, per-tensor weight quantization. The curve is flat and nearly free down to INT4, then falls off a cliff. Asymmetric (green) survives meaningfully longer than symmetric (red) at the bottom, but neither makes per-tensor INT2 usable. Closing the gap between this cliff and the flat FP32 line at low bit-widths is what Parts III–IV of the book are about.

configuration	perplexity	Δ vs FP32
FP32 baseline	5.4497	—
INT8 symmetric	5.4500	+0.0003
INT8 asymmetric	5.4495	−0.0003
INT4 symmetric	5.5105	+0.0607
INT4 asymmetric	5.4927	+0.0429
INT3 symmetric	8.5224	+3.07
INT3 asymmetric	6.3839	+0.93
INT2 symmetric	91.6263	+86.2
INT2 asymmetric	17.8578	+12.4

This table is a preview of the entire book, and it deserves to be read slowly. Figure 2.3 plots it.

- **INT8 is free.** A 4× size reduction for a perplexity change in the fourth decimal place. This is why INT8 weight quantization is considered a solved, no-brainer baseline.
- **INT4 is cheap.** A 0.04–0.06 perplexity increase for an 8× reduction — with the crude *per-tensor* method we are using. Chapter 3 will make INT4 nearly free by changing only the granularity.
- **Below INT4, per-tensor collapses.** INT3 already adds 0.9–3 points; INT2 destroys the model (perplexity 17–92). The naive method has hit its floor.

The spine, answered for this chapter's technique (per-tensor affine quantization):

(a) **bits saved:** 4× (INT8) to 16× (INT2) versus FP32.

(b) **accuracy lost:** negligible at INT8, +0.04 at INT4, catastrophic below — measured above.

(c) **mechanism:** each weight is displaced from its true value by up to half a step $s/2$; the step grows as bits shrink and as outliers stretch the range; those displacements propagate to the output and raise perplexity. Quantifying that displacement precisely is Chapter 4.

We have a working quantizer and the first evidence of a cliff. The next question is structural and cheap to exploit: we have been computing *one* scale for an entire weight matrix. What if each row, or each small block, got its own? That is granularity, and it is the most cost-effective idea in the entire field.

CHAPTER 3

Granularity

Chapter 2 ended on a cliff: per-tensor quantization is free at INT8, cheap at INT4, and catastrophic below. This chapter introduces the single most cost-effective idea in quantization — and it is almost embarrassingly simple. We have been computing *one* scale for an entire weight matrix. What if each row got its own? Each small block? The accuracy we recover, for the bytes we spend, is the best trade in the field.

Granularity in one sentence

Granularity is the number of weights that share a single scale. Coarser granularity (one scale per tensor) stores almost no overhead but forces unrelated weights to share a step size; finer granularity (one scale per row, or per small group) tracks local structure and isolates outliers, at the cost of storing more scales.

3.1 The mechanism: why one scale is too few

Recall that the step size is proportional to the range: $s = \text{absmax}/(2^{b-1} - 1)$ for symmetric quantization. A single scale for the whole matrix is set by the single largest-magnitude weight *anywhere in the matrix*. Every other weight is then quantized with that same coarse step — including weights in rows whose own values are ten or a hundred times smaller. Those small-magnitude rows are quantized far more coarsely than they need to be. Their information is thrown away to accommodate a large weight they have nothing to do with.

Figure 3.1 shows the three granularities we study. They differ only in how the matrix is partitioned into scale-sharing regions.

Why per-channel, specifically, is the natural choice for weights. For a linear layer $y_i = \sum_j W_{ij} x_j$, each output i is an independent dot product over row i of W. If row i has its own scale s_i, it factors straight out of the sum:

$$y_i \;=\; \sum_j W_{ij} x_j \;=\; \sum_j s_i \, q_{ij} \, x_j \;=\; s_i \sum_j q_{ij} x_j .$$

The inner sum is pure integer–activation arithmetic; the per-row scale is applied once, afterward. Per-channel quantization therefore costs essentially nothing at inference time — it is a single extra multiply per output — which is why it is the default for weight quantization everywhere.

3.1.1 Implementation

Per-group quantization, written naively as a double loop over rows and groups, is unusably slow. The vectorized form reshapes each row into its groups and computes all per-group ranges in one reduction:

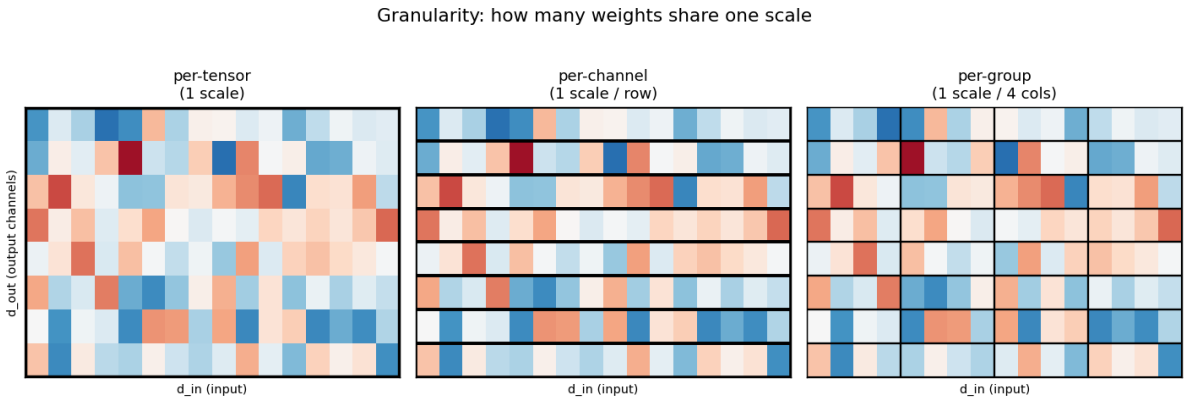


Figure 3.1: Three granularities on one weight matrix (rows are output channels, columns are inputs). **Per-tensor:** a single scale covers all weights; one large weight anywhere sets the step for everything. **Per-channel:** each output row gets its own scale, so a large weight in one row no longer coarsens the others. **Per-group:** each row is split into fixed-width groups (here, 4 columns) with a scale each, isolating local outliers to a single group.

Listing 3.1: Vectorized per-group quantization (from `quant.py`).

```python
def fake_quant_per_group_vec(W, bits=8, group_size=128, signed=True):
    d_out, d_in = W.shape
    G  = group_size
    ng = d_in // G
    qmax_pos = 2**(bits-1) - 1
    Wg = W.reshape(d_out, ng, G)                          # split each row into groups
    absmax = Wg.abs().amax(dim=2, keepdim=True).clamp_min(1e-12)
    s = absmax / qmax_pos                                  # one scale per (row, group)
    q = torch.clamp(torch.round(Wg / s), -qmax_pos-1, qmax_pos)
    return (s * q).reshape(d_out, d_in), d_out * ng
```

3.2 The recovery, measured

We quantize all the real model's linear weights at INT4, INT3, and INT2, sweeping from per-tensor to per-group-32, and measure both the weight MSE and the full-model validation perplexity. First the per-weight error:

granularity	INT4 SQNR	INT3 SQNR	INT2 SQNR
per-tensor	12.47 dB	5.39 dB	0.22 dB
per-channel	17.47 dB	10.11 dB	1.76 dB
group-128	18.47 dB	11.10 dB	2.38 dB
group-64	19.10 dB	11.73 dB	2.83 dB
group-32	20.05 dB	12.69 dB	3.59 dB

Going from per-tensor to per-channel buys a flat $\sim$5 dB at every bit-width — and recall from Chapter 2 that 6 dB is worth about one bit. So per-channel quantization recovers nearly a full bit of fidelity for $1/d_{\text{in}}$ of a scale per weight. Each halving of the group size adds roughly another 0.6–1 dB. But MSE is only the microscope; perplexity is the verdict.

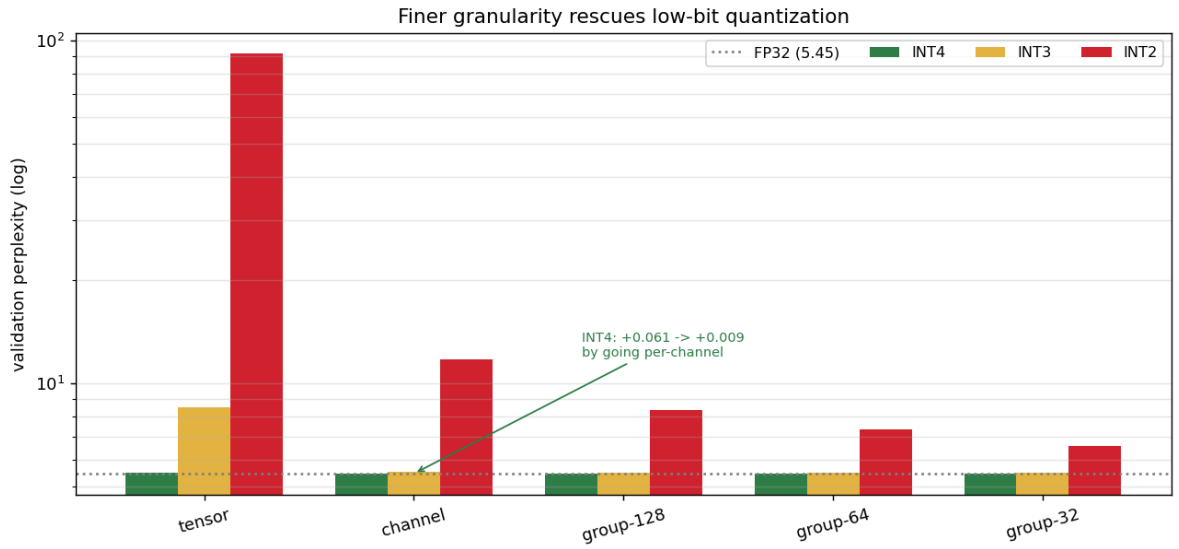


Figure 3.2: Perplexity by granularity at each bit-width (log scale; FP32 dotted). The INT2 bars (red) shrink from a catastrophic 91.6 to 6.6 as granularity refines; the INT3 bars (gold) drop below visibility against the FP32 line. INT4 (green) is essentially flat — already solved by per-channel.

granularity	INT4	INT3	INT2
per-tensor	5.5105 (+0.061)	8.5224 (+3.07)	91.63 (+86.2)
per-channel	5.4582 (+0.009)	5.5268 (+0.077)	11.79 (+6.34)
group-128	5.4643 (+0.015)	5.5101 (+0.060)	8.378 (+2.93)
group-64	5.4645 (+0.015)	5.5067 (+0.057)	7.330 (+1.88)
group-32	5.4669 (+0.017)	5.4942 (+0.044)	6.588 (+1.14)

This table contains three findings, in increasing order of importance.

INT4 becomes nearly free. Per-tensor INT4 cost +0.061 perplexity. Per-channel INT4 costs +0.009 — a 7× reduction in the damage — for an overhead of $16/d_{\text{in}}$ bits per weight, which for these matrices is about 0.06 bits. We bought back almost all of the INT4 penalty for less than a tenth of a bit. This is why production 4-bit formats are *never* per-tensor.

INT3 is rescued from collapse. Per-tensor INT3 was unusable (+3.07). Group-32 INT3 costs +0.044 — comparable to per-channel INT4. A technique that was off the table is now competitive, purely by partitioning scales more finely.

INT2 improves enormously but is still not free. This is the most instructive result in the chapter. Granularity takes per-tensor INT2 from a dead model (+86) to a 76× better +1.14 at group-32. That is a stunning recovery — and yet +1.14 perplexity is still a large, visible degradation.

When it breaks

Granularity alone does not make INT2 usable. Even at group-32 — already a heavy 0.5 bits/weight of scale overhead — 2-bit weights cost more than a full point of perplexity. The reason is fundamental: with only four code levels per group, no choice of scale can place those four levels well enough to represent a group of 32 varied weights. The error is not coming from a badly-sized *range* any more; it is coming from having too few *levels*. Fixing that needs methods that decide *which* value each weight rounds to in light of its effect on the output — GPTQ and AWQ, in Part IV. Granularity is necessary but not sufficient below 3 bits.

3.3 The overhead, and the real decision

Finer granularity is not free: each scale is a number that must be stored alongside the codes, conventionally in FP16. The overhead, in bits per weight, is the number of scales times 16, divided by the number of weights:

$$\text{overhead (bits/weight)} = \frac{(\text{scales}) \times 16}{(\text{weights})}.$$

For per-channel that is $16/d_{\text{in}}$; for group size G it is $16/G$.

granularity	overhead (bits/weight)	effective bits at INT4
per-tensor	0.0001	4.0001
per-channel	0.0625	4.0625
group-128	0.1528	4.1528
group-64	0.2500	4.2500
group-32	0.5000	4.5000

The honest way to compare quantization schemes is not by their nominal bit-width but by their **effective bit-width** — nominal bits plus scale overhead — against the accuracy they deliver. Plotting every configuration this way gives the decision curve in Figure 3.3.

A subtlety worth flagging. At INT4, per-channel (+0.009) slightly *beats* group-128 (+0.015) on perplexity, even though group-128 has the lower weight MSE. Lower MSE did not produce lower perplexity. This is our first encounter with a theme that recurs through the rest of the book: **MSE on the weights is a proxy, not the target.** What actually matters is the error at the model's output, and the mapping from weight error to output error is not a simple monotone function of MSE — it depends on which weights erred and how the activations amplify them. Chapter 5 makes this precise; for now, note only that the cheapest-MSE option is not always the best-perplexity option.

Figure 3.3: Validation perplexity versus effective bits per weight, for every bit-width × granularity combination measured. Color is nominal bit-width; marker shape is granularity. The dark line is the Pareto frontier — the best perplexity achievable at each storage budget. Two lessons stand out. First, the frontier is owned by INT4 above ∼4 effective bits and by finely-grouped INT3 between 3 and 4. Second, group-32 INT3 (3.5 effective bits) nearly matches INT4 while storing a full half-bit less — the kind of trade that only becomes visible once overhead is counted honestly.

3.4 What granularity is, and is not

The spine, for granularity (taking per-channel INT4 as the headline):

(a) **bits saved:** the same 8× as nominal INT4, costing only ∼0.06 extra bits/weight of scales (4.06 effective bits).

(b) **accuracy lost:** +0.009 perplexity — a 7× improvement over per-tensor INT4's +0.061, measured on the real model.

(c) **mechanism:** giving each row (or group) its own scale stops a few large weights from coarsening unrelated small ones; the step size now tracks local magnitude, so per-weight error falls ∼5 dB (≈ one bit) per refinement level.

Granularity is the highest-leverage idea we will meet, and it is purely structural: no calibration data, no optimization, just a finer partition of scales. It single-handedly makes INT4 production-ready and INT3 viable. But the INT2 result drew the boundary of what structure alone can do. Below 3 bits, the problem is no longer the *range* of the codes but their *number*, and no partition fixes that.

There is also a second boundary we have so far ignored. Everything in this chapter was about *weights*, whose distributions are benign. The next part of the book confronts the values that are not benign at all — the activations, and within them, the emergent outliers that make large-model quantization a genuinely hard problem. But first, Part II makes the error mathematics exact: we have been measuring MSE and SQNR and invoking "6 dB per bit" as if it were obvious. Chapter 4 derives it.

End of the free sample

You've read Part I. Thirteen more chapters await —
the noise mathematics, the outlier problem, GPTQ and AWQ
derived from scratch, the GGUF byte layouts, and the
end-to-end capstone where every number is explained.

LLM Quantization: From the Bits Up
Book Four • by Hatem M.

Build it, break it, measure it.