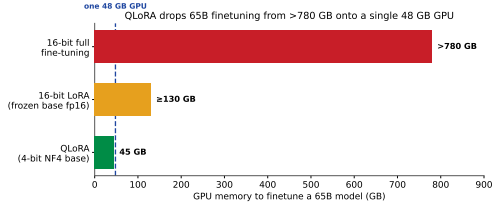
LoRA saved the adapters – but not the base model

LoRA (*[LLM-3]*) froze W_0 and trained a tiny BA – the *adapters* shrank to megabytes. But the frozen base still sits in GPU memory in **16-bit**:

- A 65B model in fp16 is $65\text{B} \times 2\text{B} \approx$ **130 GB** – just to *hold* the frozen weights.
- Full 16-bit fine-tuning of LLaMA 65B needs $>$**780 GB** – a rack of GPUs.
- So LoRA cut the *trainable* cost, not the **memory floor**.

QLoRA: store the frozen base in **4-bit**, dequantize on the fly, backprop into 16-bit LoRA adapters. 65B finetuning drops onto **one 48 GB GPU** with *no* loss in quality.



Outline

Where does the memory go?

With tiny adapters, the frozen base weights become the whole bill.

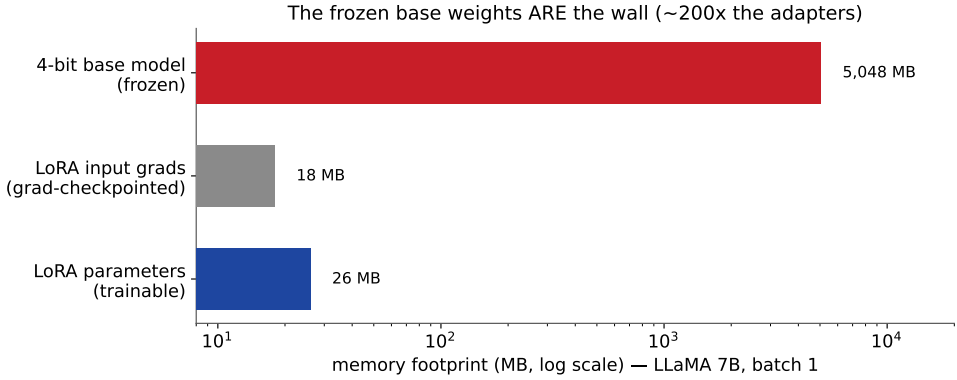
LoRA's memory floor is the frozen base

Recall LoRA (*[LLM-3]*): $Y = XW_0 + s\,XL_1L_2$, with W_0 frozen and only L_1, L_2 trained. The paper measures a 7B LoRA run (batch 1, gradient checkpointing):

- LoRA **parameters**: 26 MB. LoRA **input gradients**: 18 MB/seq.
- The frozen **4-bit base model**: **5,048 MB** – everything else is rounding error.

Aggressively shrinking the *adapter* barely moves the needle. The lever that matters is the **precision of the frozen base**. Drop it from 16-bit to 4-bit and the whole footprint collapses.

The base weights ARE the wall



So the plan writes itself: keep the base frozen (as LoRA does) but **quantize it to 4-bit**, and keep gradients only for the small 16-bit adapters. The rest of the deck is *how to quantize to 4-bit without losing accuracy*.

A better 4-bit number

Weights are normal-shaped – so build a datatype that knows that.

Block-wise quantization, and its weakness

Absmax quantization rescales a tensor into the low-bit range:

$$X_{\text{Int8}} = \text{round}\left(\frac{127}{\text{absmax}(X)}\, X\right) = \text{round}(c \cdot X), \qquad c = \text{quantization constant.}$$

- A single **outlier** inflates absmax, so most values pile into a few bins – wasted precision. Fix: chunk into **blocks** (e.g. size 64), one constant c each.
- That helps the outlier problem, but **Int4/FP4 space their levels uniformly** – and weights are *not* uniform.

Pretrained weights are $\approx$ zero-centered **normal**. A datatype whose levels match a normal will spend its 16 codes far more wisely than a uniform grid.

The intuition: put the levels where the weights are

You have only **16 values** to represent millions of weights. Where do you place them? A uniform grid wastes most of its levels on the near-empty **tails** and starves the crowded region around 0.

Better: place the 16 levels so each is the "representative" of an **equal number of weights**.

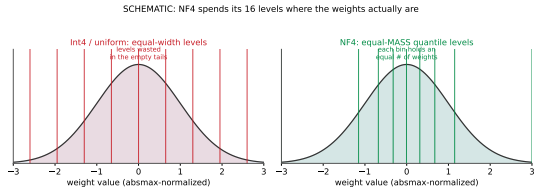
- Crowded near 0 $\Rightarrow$ **many** fine levels there.
- Sparse tails $\Rightarrow$ a **few** coarse levels.
- Every level does an equal share of the work; none is wasted.

Like drawing **voting districts by population, not by area**: each district (level) holds the same number of people (weights) – dense cities get many small districts, empty countryside gets few big ones.

That is *quantile quantization* – for bell-shaped weights, provably the best you can do at 4 bits. The next slide builds it.

NF4: equal-mass quantiles of a normal

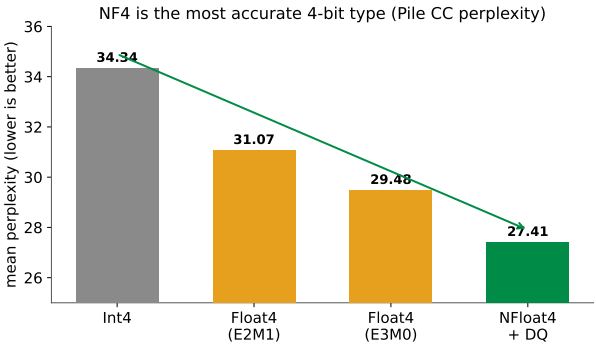
NormalFloat (NF4) is built from *quantile quantization*: pick the 16 levels so each bin holds an **equal number of weights** (equal probability mass under $\mathcal{N}(0,1)$), not equal width.



- Information-theoretically **optimal** for normally-distributed data.
- Levels cluster near 0 (where the mass is), sparse in the tails.
- An **exact 0** is kept (for padding / zero weights).

Because every block is normalized by its own absmax into $[-1, 1]$, all blocks share the *same* quantiles – so the level set is computed once, not per block.

NF4 beats FP4 and Int4 – empirically



The optimality is not just theory. Across OPT / BLOOM / Pythia / LLaMA (125M–13B), at the *same* 4 bits:

- NF4 gives the **lowest perplexity** of any 4-bit type.
- It also lifts zero-shot accuracy over FP4 and Int4 (paper Fig. 3).

Same bit budget, better numbers: NF4 is a free lunch over the naive 4-bit formats.

Squeezing the last bytes

Double Quantization and Paged Optimizers.

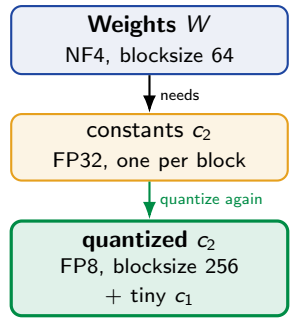
Double Quantization: quantize the quantizers

Small blocks give precise 4-bit – but each block needs a 32-bit constant c. At blocksize 64 that is $32/64 = \mathbf{0.5}$ bits *per parameter*, pure overhead.

Double Quantization (DQ): treat the constants c_2 as a new tensor and quantize *them* (FP8, blocksize 256):

$$\frac{32}{64} = 0.5 \longrightarrow \frac{8}{64} + \frac{32}{64 \cdot 256} \approx \mathbf{0.127} \text{ bits/param.}$$

Saves $\approx$**0.37 bits/param** – about **3 GB** on a 65B model – with *no* measured quality loss.

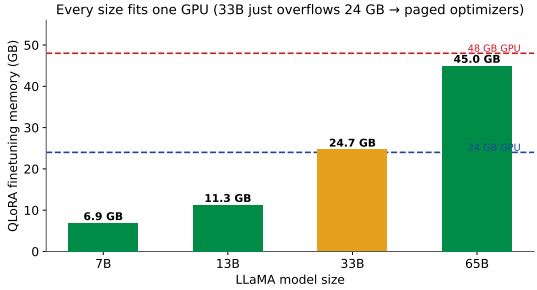


Paged Optimizers: survive the memory spikes

Gradient checkpointing keeps average memory low, but a **long sequence** can spike the optimizer state and trigger an **out-of-memory** crash mid-run.

Paged Optimizers use NVIDIA *unified memory*: optimizer states are paged out to CPU RAM on a spike and paged back for the update step – like OS memory paging to disk. The run survives instead of crashing.

This is exactly what lets 33B fit a 24 GB card and 65B fit a 48 GB card – the peak, not the average, is what OOMs you.



Putting it together

4-bit storage, 16-bit compute – gradients only into the adapters.

One storage type, one compute type

For a single linear layer with one LoRA adapter, QLoRA computes:

$$Y^{\text{BF16}} = X^{\text{BF16}}\,\text{doubleDequant}\big(c_1^{\text{FP32}}, c_2^{\text{k-bit}}, W^{\text{NF4}}\big) \;+\; X^{\text{BF16}} L_1^{\text{BF16}} L_2^{\text{BF16}}$$

- **Storage** datatype: 4-bit NF4 for the frozen base W (+ FP8 constants via DQ).
- **Compute** datatype: BFloat16. On every use, W^{NF4} is **dequantized to bf16** for the matmul, then discarded.
- Gradients flow *through* W but are computed **only for the adapters** L_1, L_2 – never for the 4-bit weights.

The base is 4-bit *at rest* and 16-bit *in flight*. You pay 16-bit compute briefly, block by block, but never store 16-bit weights – so accuracy is 16-bit while memory is 4-bit.

The tuning detail that makes it match: adapters everywhere

Default LoRA adapts only $\{W_q, W_v\}$. At 4-bit that is **not enough** to recover full quality.

- QLoRA puts LoRA adapters on **every linear layer** (attention *and* MLP).
- Because adapters are nearly free in memory (previous section), using *many* of them costs almost nothing.
- The rank r barely matters; the **number of adapted layers** is the critical knob (paper Fig. 2).

This is the fix for the accuracy trade-offs seen in earlier PEFT work: cheap adapters mean you can afford to adapt *all* layers, and that closes the gap to 16-bit.

Does 4-bit cost anything?

The headline: no measurable drop – and a state-of-the-art chatbot on one GPU.

Predict-first: 4 bits vs 16 bits

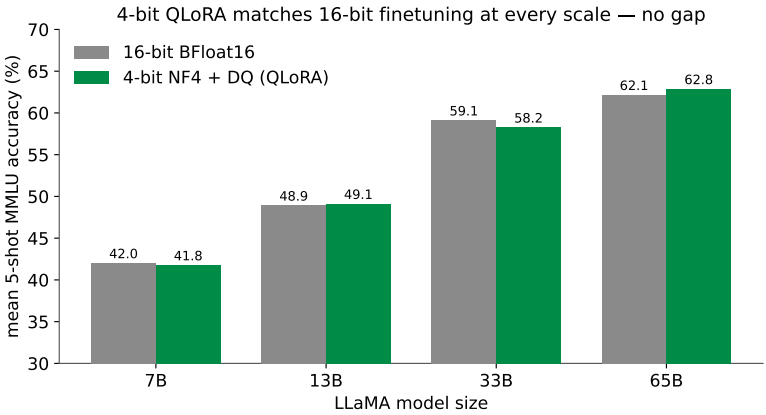
You are about to compress the frozen base to a **quarter** of its bits.

Guess: how much MMLU accuracy does 4-bit QLoRA lose versus 16-bit fine-tuning at 7B / 13B / 33B / 65B?

Predict-first: 4 bits vs 16 bits

You are about to compress the frozen base to a **quarter** of its bits.

Guess: how much MMLU accuracy does 4-bit QLoRA lose versus 16-bit fine-tuning at 7B / 13B / 33B / 65B?



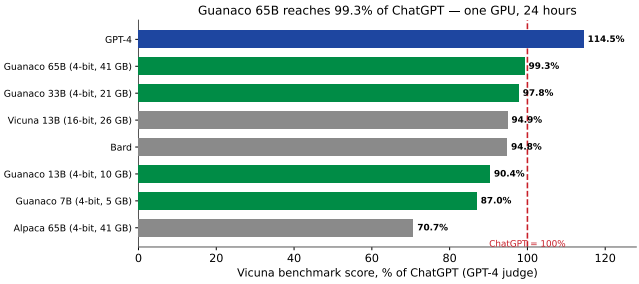
No gap: QLoRA (4-bit) matches 16-bit

Data type	T5-80M	T5-250M	T5-780M	T5-3B	**Mean (all)**
BF16 full FT	40.1	42.1	48.0	54.3	53.0
LoRA BF16	40.5	42.6	47.1	55.4	–
QLoRA FP4	40.3	42.4	47.5	55.6	52.2
QLoRA NF4 + DQ	**40.4**	**42.7**	**47.7**	55.3	**53.1**

- On GLUE / Super-NaturalInstructions and on MMLU (7B–65B), **4-bit NF4+DQ recovers full 16-bit performance**.
- FP4 trails NF4 by $\approx$**1 point** – the NF4 datatype is doing real work.

"Lost" precision is fully **recovered by the adapters**: quantize to 4-bit, then let 16-bit LoRA finetuning heal the damage.

Guanaco: a ChatGPT-class chatbot on one GPU



Guanaco = QLoRA on LLaMA + OASST1.

- **65B**: **99.3%** of ChatGPT on Vicuna – best open model, trained in **24 h** on *one* GPU.
- **33B**: 97.8%, trainable in $<$12 h on a 24 GB consumer card.
- **7B**: 5 GB, beats a 13B Alpaca by $\sim$20 points.

Efficiency turned into capability: cheaper finetuning let them train 1,000+ models and find the best recipe.

Recap

- **The problem:** LoRA freezes the base but still holds it in 16-bit – 65B is $\sim$130 GB, so many GPUs. The base weights are the memory wall.
- **NF4:** a 4-bit datatype with *equal-mass* levels, optimal for normal-shaped weights – beats FP4/Int4 at the same bits.
- **Double Quantization:** quantize the constants too – -0.37 bits/param.
- **Paged Optimizers:** unified memory absorbs checkpointing spikes so long sequences do not OOM.
- **Mechanism:** store base in 4-bit NF4, dequantize to bf16 per matmul, backprop only into 16-bit LoRA adapters on *every* layer.
- **Result:** 4-bit *matches* 16-bit with no drop; Guanaco 65B hits **99.3%** of ChatGPT, trained in a day on a single 48 GB GPU.

In one line: QLoRA = quantized LoRA – it keeps LoRA's tiny adapters and adds a lossless 4-bit base, so *one* GPU now finetunes the largest open models.