Lecture 2: PyTorch and Resource Accounting

Language Modeling from Scratch · lecture notes · rebuilt for NVIDIA Blackwell, late 2026

Section 1: Back-of-the-Envelope Resource Accounting

> **KEY IDEA**
> Training a modern language model is an optimization problem constrained by two physical budgets: **compute** (how many arithmetic operations the hardware can execute per second) and **memory** (how many bytes fit inside GPU memory). Every architectural choice, precision format, batch size, and training loop decision sits downstream of one core question: *what is the best language model we can train given a fixed budget of compute and memory?*

Launching a multi-week training run on a GPU cluster without first checking whether the model fits in memory or how many days the run will take leads directly to out-of-memory crashes and wasted hardware. Maximizing computational efficiency starts with knowing where every byte and every floating-point operation (FLOP) goes. Three capabilities make this possible:

- **Mechanics:** understanding PyTorch tensor semantics and CUDA execution fundamentals.
- **Mindset:** performing back-of-the-envelope accounting before launching any run.
- **Intuition:** recognizing compute-bound versus memory-bound operations on Blackwell GPUs.

To make these ideas concrete before deriving every formula from first principles, consider two back-of-the-envelope calculations that every systems engineer performs before touching training code: estimating total wall-clock training time and bounding maximum model size.

Napkin Calculation I: Training Time on Blackwell

> **WORKED EXAMPLE**
> Suppose we want to train a dense 70B-parameter (70×10^9 parameters) language model on 15T (15×10^{12}) tokens using $1{,}024$ NVIDIA B200 (Blackwell) GPUs, assuming the cluster achieves a **Model FLOPs Utilization (MFU)** of 0.5 (meaning the training run sustains 50% of the GPUs' advertised dense peak arithmetic speed). How many days will this training run take?
>
> We calculate the answer in four steps:
>
> 1. **Total compute required.** For a dense model with N parameters trained on D tokens, the total training work is approximately:
>
> $$\text{FLOPs} = 6ND$$
>
> We will derive this factor of 6 from matrix multiplication and backpropagation in Section 9. Here

$N = 70 \times 10^9$ is the parameter count and $D = 15 \times 10^{12}$ is the number of training tokens. Substituting these values gives:

$$\text{FLOPs} = 6 \times (70 \times 10^9) \times (15 \times 10^{12}) = 6 \times 1{,}050 \times 10^{21} = 6.3 \times 10^{24} \text{ FLOPs}$$

2. **Per-GPU dense peak speed.** A Blackwell B200 datasheet advertises 4.5 PFLOP/s ($4.5 \times 10^{15} \text{ FLOP/s}$) in BF16, but that headline figure assumes 2:4 structured sparsity (half of the weights are zero in a rigid hardware pattern). Dense matrix multiplications receive exactly half of that peak:

$$F_{\text{peak, dense}} = \frac{4.5 \times 10^{15} \text{ FLOP/s}}{2} = 2.25 \times 10^{15} \text{ FLOP/s per GPU}$$

   You might think you can plug the datasheet's headline 4.5 PFLOP/s straight into a training-time estimate, but doing so underestimates your training duration by a factor of 2 whenever your model uses standard dense weights. We will examine precision and sparsity peaks in detail in Section 6.

3. **Effective cluster throughput per day.** Multiplying the per-GPU dense peak by the utilization factor $\text{MFU} = 0.5$, the number of GPUs ($1{,}024$), and the number of seconds in a day ($86{,}400 \text{ s/day}$) yields the useful arithmetic work performed by the entire cluster each day:

$$\text{Cluster throughput} = (2.25 \times 10^{15} \text{ FLOP/s}) \times 0.5 \times 1{,}024 \times 86{,}400 \text{ s/day}$$

   Breaking the multiplication down step by step: $2.25 \times 0.5 = 1.125$; $1.125 \times 1{,}024 = 1{,}152$; $1{,}152 \times 86{,}400 = 99{,}532{,}800$; multiplying by 10^{15} gives $9.953 \times 10^{22} \approx 9.95 \times 10^{22} \text{ FLOP/day}$.

4. **Wall-clock time.** Dividing total required work by daily cluster throughput gives:

$$T = \frac{6.3 \times 10^{24} \text{ FLOPs}}{9.953 \times 10^{22} \text{ FLOP/day}} \approx 63.3 \text{ days}$$

For a historical baseline, running the exact same $6.3 \times 10^{24} \text{ FLOPs}$ job on $1{,}024$ NVIDIA H100 GPUs (whose dense BF16 peak is $989.5 \text{ TFLOP/s} = 9.895 \times 10^{14} \text{ FLOP/s}$) at $\text{MFU} = 0.5$ yields a daily throughput of $(9.895 \times 10^{14}) \times 0.5 \times 1{,}024 \times 86{,}400 \approx 4.38 \times 10^{22} \text{ FLOP/day}$, taking $6.3 \times 10^{24} / (4.38 \times 10^{22}) \approx 144 \text{ days}$—more than twice as long.

DEFINITION

One architectural qualification applies to sparse **Mixture-of-Experts (MoE)** models: because each token only routes through a fraction of the model's parameters, the compute formula substitutes the number of **active parameters** N_{active} for N, giving $\text{FLOPs} = 6 N_{\text{active}} D$.

Napkin Calculation II: Model Size Upper Bound

WORKED EXAMPLE

Before worrying about how long training takes, the model must fit into GPU High Bandwidth Memory (HBM). Suppose we have 8 GPUs and train with **BF16/FP32 AdamW** mixed precision. What is the largest model (measured in parameter count $P_{\max}$) that can possibly fit?

Under standard BF16/FP32 AdamW training, every trainable parameter requires a fixed static memory budget:

- **BF16 parameters:** 2 bytes
- **BF16 gradients:** 2 bytes
- **FP32 AdamW optimizer moments** (two running averages per parameter, each stored in 4-byte FP32): $4 + 4 = 8$ bytes

Summing these three components gives a total static footprint of 12 bytes per parameter.

If each of the 8 GPUs has HBM capacity M_{GPU} (in bytes), the aggregate cluster memory is $8\,M_{\text{GPU}}$. Dividing total available bytes by 12 bytes/parameter produces an upper bound on parameter count:

$$P_{\max} \leq \frac{8\,M_{\text{GPU}}}{12}$$

Working through the historical H100 baseline where each GPU has $M_{\text{GPU}} = 80$ GB $= 80 \times 10^9$ bytes:

$$P_{\max} \leq \frac{8 \times 80 \times 10^9 \text{ bytes}}{12 \text{ bytes/parameter}} = \frac{640 \times 10^9}{12} \approx 53.3 \times 10^9 = 53.3\text{B parameters}$$

COMMON MISCONCEPTION

Notice the inequality sign ($\leq$). You might think you can actually train a 53.3B model on eight 80 GB H100s, but that breaks for two reasons. First, this calculation counts *only* static model state and leaves zero bytes for intermediate **activations**, which grow with batch size and context length (we will quantify and tame activations in Sections 10 and 11). Second, for sparse MoE models, even though compute scales with N_{active}, GPU memory must still store **all** N_{total} parameters, gradients, and optimizer states.

Both napkin calculations relied on numbers we have not yet justified: why BF16 takes 2 bytes while FP32 takes 4 bytes, why we mix them, why a training step costs $6ND$ FLOPs, and why MFU sits around 0.5 rather than 1.0. To derive every one of these quantities from first principles, we start with the fundamental data structure of deep learning—the tensor—and the binary formats used to store its numbers.

✎ Check Your Understanding

1. **Quick Computation:** Suppose you cut the number of Blackwell B200 GPUs in Napkin Calculation I from $1{,}024$ to 256, keeping everything else (70B dense model, 15T tokens, $\text{MFU} = 0.5$) unchanged. Approximately how many days will the training run take?
   Answer: Approximately 253.2 days. Reducing the GPU count by a factor of 4 reduces the cluster's daily throughput by $4\times$, which multiplies the wall-clock time (63.3 days) by 4.
2. **What Would Happen If:** You replace a 70B dense model with a sparse MoE model that has $N_{\text{total}} = 70\text{B}$ total parameters but only $N_{\text{active}} = 10\text{B}$ active parameters per token. How do (a) the total training FLOPs and (b) the static HBM memory footprint change?
   Answer: Total training FLOPs drop by $7\times$ because sparse MoE compute scales with $6N_{\text{active}}D$, but the static HBM memory footprint stays unchanged at $12N_{\text{total}}$ bytes because all 70B parameters, their gradients, and their optimizer states must remain in GPU memory.

Section 2: Tensors and Floating-Point Number Representations

A training script juggles many conceptually different objects—input data batches, intermediate layer activations, learnable weight matrices, gradients, and optimizer running averages—yet hardware cannot treat each differently without massive complexity. PyTorch unifies all of them into a single data structure: the **tensor**.

Tensors, Rank, and Memory Accounting

DEFINITION

A **tensor** is a multi-dimensional array of numbers of a uniform data type (`dtype`). Every training object is stored as a tensor:

- Data batches and intermediate activations
- Model parameters and their gradients
- Optimizer states (such as running moments)

Large modern models, such as **DeepSeek-V4.1-Flash**, organize their weights as hundreds of sharded tensors distributed across attention layers, routers, and sparse MoE experts.

DEFINITION

The **rank** of a tensor is its number of axes (dimensions):

```
x = torch.zeros(4)        # rank 1 (vector of length 4)
x = torch.zeros(4, 8)     # rank 2 (4 x 8 matrix)
x = torch.zeros(4, 8, 2)  # rank 3 (3D array)

# Rank-4 Transformer attention tensor
B = 32   # batch size
S = 16   # sequence length
H = 16   # attention heads
D = 64   # head dimension
q = torch.zeros(B, S, H, D)
```

In this attention example, `q` has shape `(32, 16, 16, 64)` and rank 4. Tracking tensor shapes and element data types is the entire foundation of memory and FLOP accounting.

DEFINITION

The exact memory footprint of any tensor depends on only two numbers: how many elements it holds (`numel`, short for "number of elements") and how many bytes each element occupies (`element_size`):

$$\text{Memory (bytes)} = \text{numel}(x) \times \text{element_size}(x)$$

Let's break this equation apart:

- $\text{numel}(x)$ is the product of all dimension sizes along every axis of x. For a tensor of shape $(d_1, d_2, \ldots, d_r)$, $\text{numel}(x) = \prod_{i=1}^{r} d_i$.
- $\text{element_size}(x)$ is determined exclusively by the tensor's `dtype` (for example, 4 bytes for 32-bit floating point, 2 bytes for 16-bit floating point, 1 byte for 8-bit floating point).
- Together, the product gives the exact contiguous payload size in bytes required to store x in memory.

WORKED EXAMPLE

Worked Example 1 (Small PyTorch Tensor): Consider `x = torch.zeros(4, 8)`. Because no `dtype` is specified, PyTorch defaults to `torch.float32`, where each element takes 4 bytes:

```
x = torch.zeros(4, 8)         # default dtype: torch.float32 (4 bytes)
assert x.numel() == 32 and x.element_size() == 4
bytes_used = x.numel() * x.element_size()   # 32 * 4 = 128 bytes
```

The shape `(4, 8)` contains $4 \times 8 = 32$ elements, and $32 \times 4 = 128$ bytes.

WORKED EXAMPLE

Worked Example 2 (Single GPT-3 Feedforward Matrix in FP32): To see why default FP32 becomes unworkable at language-model scale, compute the footprint of a *single* feedforward weight matrix from the historical dense GPT-3 architecture, which has shape $49{,}152 \times 12{,}288$ in FP32:

1. Factor each dimension by $1{,}024$: $49{,}152 = 48 \times 1{,}024$ and $12{,}288 = 12 \times 1{,}024$.
2. Compute the element count:

$$\text{numel} = (48 \times 1{,}024) \times (12 \times 1{,}024) = 576 \times 1{,}024^2 \approx 6.04 \times 10^8 \text{ elements}$$

3. Multiply by 4 bytes/element:

$$\text{Memory} = 576 \times 4 \times 1{,}024^2 \text{ bytes} = 2{,}304 \times 1{,}024^2 \text{ bytes} = 2{,}304 \text{ MiB} = 2.25 \text{ GiI}$$

KEY IDEA

Over 2.25 GiB of HBM is consumed by *one* weight matrix in *one* layer of a multi-layer network—before storing its gradient, optimizer state, or activations. Furthermore, because every matrix multiplication must read those bytes from HBM into the processor, **cutting bytes per element cuts both HBM capacity usage and HBM memory traffic proportionally**. This dual payoff motivates shrinking floating-point formats below 32 bits.

Floating-Point Formats: FP32, FP16, and BF16

Recall that standard binary floating-point numbers (IEEE 754 style) represent a real value using three bit fields:

$$\text{value} = (-1)^S \times (1.\text{mantissa})_2 \times 2^{\text{exponent}-\text{bias}}$$

Here:

- The **Sign bit (S)** (1 bit) selects positive (0) or negative (1).
- The **Exponent bits** set the **dynamic range**—how astronomically large or vanishingly small a non-zero number can be before overflowing to infinity or underflowing to zero.
- The **Mantissa (or fraction) bits** set the **numerical resolution**—how finely spaced representable numbers are between consecutive powers of two.

Comparing the bit layouts of **FP32**, **FP16**, and **BF16** reveals a fundamental design tradeoff. Picture three horizontal bit bars stacked vertically, each starting with a 1-bit sign block (`S`) on the left, followed by an exponent block in the middle and a mantissa block on the right:

DEFINITION

- **FP32 (4 bytes = 32 bits):** `[ S: 1 bit | Exponent: 8 bits | Mantissa: 23 bits ]`. This is the scientific-computing default. Its 8-bit exponent gives an enormous dynamic range (normal magnitudes from $\approx 1.18 \times 10^{-38}$ to $\approx 3.40 \times 10^{38}$), and its 23-bit mantissa provides roughly 7 decimal digits of precision.

DEFINITION

- **FP16 (2 bytes = 16 bits):** `[ S: 1 bit | Exponent: 5 bits | Mantissa: 10 bits ]`. To cut memory in half, classic IEEE FP16 shrinks *both* the exponent (from 8 down to 5 bits) and the mantissa (from 23 down to 10 bits). The shorter bar is half as wide as FP32, saving 50% memory.

DEFINITION

- **BF16 (Brain Floating Point, 2 bytes = 16 bits, introduced 2018):** `[ S: 1 bit | Exponent: 8 bits | Mantissa: 7 bits ]`. BF16 takes the exact same 16-bit budget as FP16, but aligns its exponent box directly underneath FP32's 8-bit exponent box—preserving all 8 exponent bits—and pays for it by shrinking the mantissa aggressively down to 7 bits.

KEY IDEA

Why did deep learning abandon pure FP16 in favor of BF16? Because deep learning training is surprisingly tolerant of coarse mantissa resolution (weights and gradients are already noisy estimates from mini-batches), but it is catastrophically sensitive to **exponent range**.

WORKED EXAMPLE

Worked Example 3 (Exposing FP16 Underflow vs. BF16 Stability): Consider a small but legitimate gradient value encountered during neural network training: $g = 10^{-8}$.

- **In FP16 (5-bit exponent, bias $= 15$):** The smallest normal exponent is $1 - 15 = -14$, giving a smallest normal magnitude of $2^{-14} \approx 6.10 \times 10^{-5}$. Even using subnormal numbers (which sacrifice mantissa bits to represent smaller values down to $2^{-14} \times 2^{-10} = 2^{-24} \approx 5.96 \times 10^{-8}$), the halfway rounding threshold to zero is 2.98×10^{-8}. Because $10^{-8} < 2.98 \times 10^{-8}$, FP16 rounds 10^{-8} straight to 0.0 (**underflow**). The gradient vanishes completely, and learning stalls unless you add complex software workarounds like dynamic loss scaling.
- **In BF16 (8-bit exponent, bias $= 127$):** The smallest normal magnitude is $2^{-126} \approx 1.18 \times 10^{-38}$, identical to FP32. Representing 10^{-8} requires an exponent around $2^{-27} \approx 7.45 \times 10^{-9}$, which sits comfortably inside the 8-bit exponent range. Thus in BF16, $10^{-8} \neq 0$; the gradient survives intact, albeit rounded to 7 bits of mantissa precision ($\approx 0.78\%$ relative spacing).

COMMON MISCONCEPTION

You might think that because FP16 has 10 mantissa bits compared to BF16's 7 mantissa bits, FP16 is always "more accurate" for training. That intuition breaks because relative precision only helps if the number is representable at all; once a gradient underflows to zero or an activation exceeds FP16's maximum of $65{,}504$ and overflows to ∞ (`NaN`), extra mantissa bits are worthless. BF16 keeps FP32's 8-bit dynamic range at half the memory cost, eliminating most underflow instability.

However, keeping only 7 bits of mantissa introduces a subtler numerical hazard when adding very small updates onto much larger weights over thousands of training steps. Addressing that hazard—and pushing even further down to 8-bit and 4-bit formats on Blackwell hardware—requires mixed precision and microscaling.

✎ Check Your Understanding

1. **Fill in the Blank:** A tensor's memory footprint in bytes is equal to its `numel` multiplied by its `__________`; halving that second quantity reduces both HBM capacity usage and HBM memory traffic by a factor of 2.
   Answer: `element_size` (or bytes per element). By $\text{Memory} = \text{numel}(x) \times \text{element_size}(x)$, cutting bytes per element halves both the stored footprint and the number of bytes transferred per read/write.
2. **Yes/No with Justification:** Can a standard FP16 tensor represent the gradient value 10^{-8} without underflowing to zero?
   Answer: No. FP16 has only a 5-bit exponent (smallest positive subnormal $\approx 5.96 \times 10^{-8}$), so 10^{-8} underflows to 0, whereas BF16 preserves FP32's 8-bit exponent and represents 10^{-8} cleanly.

Section 3: Mixed Precision, FP8, and Microscaled NVFP4

DEFINITION

In Section 2, we saw that pure FP32 wastes HBM capacity and memory bandwidth (and as we will see in Section 6, leaves high-throughput low-precision Tensor Cores unused). Yet switching the entire training pipeline to pure BF16 fails for a specific numerical reason: **swamping during accumulation**.

With only 7 mantissa bits, the spacing between adjacent BF16 numbers around 1.0 is $2^{-7} = 1/128 \approx 0.0078125$. Suppose a parameter has value $w = 1.0$, and the optimizer computes a parameter update of $-\eta g = -10^{-4}$. When you compute $1.0 - 10^{-4} = 0.9999$ in pure BF16, the nearest representable BF16 number is still 1.0 (since 10^{-4} is far smaller than half of 2^{-7}, which is ≈ 0.0039). The update disappears completely! Over thousands of steps, tiny updates fail to accumulate, and sensitive nonlinearities like exponentials (`exp`), softmax denominators, and long sums (reductions) accumulate severe rounding drift.

Mixed Precision Training and PyTorch AMP

DEFINITION

Mixed precision resolves this conflict by assigning each tensor and operation the lowest precision it can safely tolerate:

- **BF16 (2 bytes):** parameters, intermediate activations, and gradients during matrix multiplications—where the bulk of compute and memory traffic lives.
- **FP32 (4 bytes):** optimizer states (running moment averages that accumulate tiny updates over tens of thousands of steps) and numerically sensitive nonlinearities such as `exp` , softmax, layer normalizations, and reductions.

DEFINITION

In PyTorch, **Automatic Mixed Precision (AMP)** automates these casts via a context manager:

```
with torch.amp.autocast("cuda", dtype=torch.bfloat16):
    logits = model(x)   # casts matmuls to bf16, keeps exp/norms in fp32
```

KEY IDEA

Inside the `autocast` block, PyTorch inspects each operation: matrix multiplications and linear layers execute in BF16 on Tensor Cores, while reductions, exponentials, and normalizations stay in FP32. Moreover, Blackwell Tensor Cores natively accumulate the inner products of low-precision matrix multiplications in higher precision (FP32) internally before writing the final result out, preventing rounding collapse inside long dot products.

WORKED EXAMPLE

Worked Example 1 (Swamping in Pure BF16 vs. Survival in Mixed Precision): Suppose an optimizer maintains a running first moment $m = 1.0$ and needs to add an update $\Delta = 0.001$ on every step for 100 steps (true final value: $1.0 + 100 \times 0.001 = 1.1$).

- **In pure BF16:** Around 1.0, representable numbers are spaced by $2^{-7} = 0.0078125$, so the rounding midpoint above 1.0 is $1.0 + 0.00390625 = 1.00390625$. On step 1, $1.0 + 0.001 = 1.001 < 1.00390625$, which rounds right back to 1.0. After 100 steps, the stored value is still 1.0 (100% of the signal was lost).
- **In FP32 optimizer state:** Around 1.0, representable numbers are spaced by $2^{-23} \approx 1.19 \times 10^{-7}$. Adding 0.001 is over $8{,}000\times$ larger than the spacing, so every step accumulates accurately, reaching 1.1000 after 100 steps.

We will rely directly on this BF16/FP32 mixed-precision split when we count the $12\ \text{bytes/parameter}$ static memory budget in Section 10.

Low Precision on Blackwell: FP8 and NVFP4

Since memory traffic and Tensor Core throughput both improve as bit width shrinks, hardware architectures pushed past 16 bits to **8-bit floating point (FP8)** (standardized in 2022, introduced natively on Hopper) and **4-bit floating point (NVFP4)** (introduced by NVIDIA in 2025 and accelerated natively on Blackwell Tensor Cores).

At 8 bits (1 byte), there are only 7 bits left after the sign bit, so no single split satisfies both the precision needs of weights/activations and the wider dynamic range needed by gradients. Consequently, FP8 defines **two** complementary formats:

DEFINITION

- **FP8 E4M3 (1 byte = 8 bits):** `[ S: 1 bit | Exp: 4 bits | Mant: 3 bits ]`. Its representable range is $[-448, 448]$. Because it allocates 3 bits to the mantissa, it gives finer resolution and is used for **weights and activations**, whose magnitudes are typically well-behaved and bounded.

DEFINITION

- **FP8 E5M2 (1 byte = 8 bits):** `[ S: 1 bit | Exp: 5 bits | Mant: 2 bits ]`. Its representable range is $[-57{,}344, 57{,}344]$—matching FP16's 5-bit exponent. Sacrificing one mantissa bit for an extra exponent bit extends the dynamic range by more than $100\times$, making E5M2 suited for **gradients**, which span many orders of magnitude during backpropagation.

What happens when we try to cut the representation in half *again* to **4 bits**? With only 4 bits per number, there are at most $2^4 = 16$ distinct bit patterns in total! A raw 4-bit floating-point format without any extra scaling can encode only 16 discrete values:

$$0,\ \pm0.5,\ \pm1,\ \pm1.5,\ \pm2,\ \pm3,\ \pm4,\ \pm6$$

(accounting for $+0$ and -0 plus 7 non-zero magnitudes with two signs: $2 + 14 = 16$ patterns). If an entire weight matrix or activation tensor had to fit directly into $[-6, 6]$ with only those 16 grid points, large values would clip catastrophically and small values (like 0.01) would vanish to zero.

DEFINITION

Blackwell solves this with **NVFP4** via **microscaling**. Visually, think of an NVFP4 block as a highlighted box labeled **Block Scale (shared factor)** multiplied (`×`) by a contiguous row of small 4-bit elements `[ 4b | 4b | 4b | ... | 4b ]` :

$$\text{decoded value}_i = (\text{Block Scale}) \times q_i, \qquad q_i \in \{0, \pm0.5, \pm1, \pm1.5, \pm2, \pm3, \pm4, \pm6\}$$

Let's unpack what this representation accomplishes:

- Instead of using one global scale for an entire million-element tensor (where a single outlier destroys resolution for every other entry), the tensor is partitioned into small blocks of neighboring elements.
- Every block stores one higher-precision **shared scale factor** that absorbs the local magnitude of that block.
- Each individual element inside the block stores only a 4-bit value q_i drawn from the 16-level grid, representing its relative size within that local block.
- Microscaling thus restores wide dynamic range *across* blocks while keeping the storage cost close to 4 bits per element *within* each block. Crucially, Blackwell Tensor Cores consume the 4-bit elements and per-block scale factors directly in hardware during matrix multiplication. This is not merely theoretical: **Nemotron 3 Super** (2026) was trained natively in NVFP4 using NVIDIA libraries.

WORKED EXAMPLE

Worked Example 2 (NVFP4 Microscaling on Two Blocks of Different Magnitudes): Suppose a layer contains two adjacent blocks of values:

- Block A has small entries: $[0.002,\ -0.006,\ 0.004,\ 0.003]$.
- Block B has much larger entries: $[120,\ -360,\ 240,\ 180]$.

If we tried to quantize both blocks directly onto the fixed 16-value grid $\{0, \pm0.5, \ldots, \pm6\}$ without microscaling, every number in Block A would round to 0 (underflow) and every number in Block B would clip to ±6 (overflow). With NVFP4 per-block scaling:

1. For **Block A**, choose shared scale $s_A = 0.001$. Dividing Block A by s_A yields $[2, -6, 4, 3]$—every value hits an exact 4-bit grid point! Reconstructing $s_A \times q_i$ recovers $[0.002, -0.006, 0.004, 0.003]$ with zero error.
2. For **Block B**, choose shared scale $s_B = 60$. Dividing Block B by s_B yields $[2, -6, 4, 3]$—again exact 4-bit grid points! Reconstructing $s_B \times q_i$ recovers $[120, -360, 240, 180]$ with zero error.

> Even though Block A and Block B differ in magnitude by a factor of $60{,}000$, microscaling lets both blocks use the full dynamic range of the 4-bit grid simultaneously.

Now that we know how numbers are packed into tensors and how precision controls byte counts, a physical question arises: *where* in the computer do these tensor bytes actually live, how does the GPU move them to its compute engines, and how do we accurately measure the time those operations take?

> **✎ Check Your Understanding**
>
> 1. **Yes/No with Justification:** In FP8 training, would you store gradients in E4M3 and weights in E5M2?
>
>   **Answer:** No—it is the reverse. Weights and activations use E4M3 (more mantissa precision, narrower range $[-448, 448]$), while gradients require E5M2's extra exponent bit (range $[-57{,}344, 57{,}344]$) to capture wide dynamic range without underflow or overflow.
>
> 2. **Quick Computation:** Suppose an NVFP4 block has a shared block scale of $s = 0.25$ and stores the 4-bit quantized elements $[1.5,\ -4,\ 0.5,\ 6]$. What are the four decoded values?
>
>   **Answer:** Multiply each 4-bit element by $s = 0.25$: $[0.375,\ -1.0,\ 0.125,\ 1.5]$.

Section 4: GPU Memory Hierarchy and Asynchronous CUDA Execution

Knowing that a tensor occupies $2.3\ \text{GiB}$ is incomplete unless you know *which* memory chip holds those bytes and *how* the processor accesses them. A machine equipped with GPUs is not a single unified memory pool; it is two separate computers connected by a relatively slow cable (the host–device interconnect).

CUDA Memory Spaces: Host (CPU) vs. Device (GPU)

Consider the system architecture linking the **Host (CPU)** and the **Device (Blackwell GPU)**:

- On the left sits the **Host (CPU)** attached to **System DRAM**. By default, calling `torch.zeros(...)` allocates tensors here in System DRAM.
- On the right sits the **Device (Blackwell GPU)**, which contains two internal regions: the compute die (**SMs + Tensor Cores + on-chip SRAM**) and the GPU's own attached **GPU HBM (High Bandwidth Memory)**, where GPU tensors are stored.
- Connecting the two boxes is a bidirectional **Host–Device Interconnect** (such as PCIe or NVLink-C2C). Calling `x.to("cuda")` triggers a Host-to-Device (H2D) copy across this link.

KEY IDEA

The critical hardware fact is that **CPU and GPU have separate physical memory spaces, and the host–device interconnect is far slower than GPU HBM**. Moving tensors back and forth across that interconnect starves the GPU's compute units and stalls training. Therefore, allocate tensors directly on CUDA whenever possible:

```
x = torch.zeros(32, 32)       # Allocates in CPU System DRAM
x = x.to("cuda")              # Slow Host-to-Device (H2D) copy across interconnect

# Better: create directly in GPU HBM with zero CPU allocation or transfer
with torch.device("cuda"):
    w = torch.zeros(32, 32)
```

Blackwell GPU Architecture and Memory Hierarchy

Zooming inside the Blackwell GPU itself reveals that "GPU memory" is not flat either. It forms a strict four-tier **memory hierarchy pyramid** where moving upward brings dramatically faster latency, higher bandwidth, and smaller capacity, while moving downward brings larger capacity and slower bandwidth:

DEFINITION

1. **Top of pyramid — Registers & TMEM (Tensor Memory):** Per-SM (Streaming Multiprocessor) Tensor Core state, fastest and smallest. Blackwell introduces dedicated **Tensor Memory (TMEM)** inside each SM specifically to hold Tensor Core matrix accumulators without pressuring the general-purpose register file.

DEFINITION

2. **Second tier — Shared Memory (SMEM):** An on-chip, per-SM programmer-managed SRAM scratchpad.

DEFINITION

3. **Third tier — L2 Cache:** On-chip cache shared across all SMs on the GPU.

DEFINITION

4. **Base of pyramid — GPU HBM (Global Memory):** Largest capacity (tens to hundreds of gigabytes), slowest bandwidth on the device.

Inside each Blackwell **Streaming Multiprocessor (SM)** live two distinct compute engines:

DEFINITION

- **CUDA cores:** execute general scalar and vector operations (standard FP32 arithmetic, elementwise activations like ReLU/GELU, indexing).

DEFINITION

- **Tensor Cores:** dedicated matrix multiply-accumulate hardware supporting low-precision formats (BF16, FP8, NVFP4) at vastly higher throughput than CUDA cores.

KEY IDEA

Every PyTorch tensor lives in **GPU HBM** at the bottom of the pyramid. Whenever a kernel executes, the SM cannot compute directly on HBM; it must load chunks ("tiles") of tensors from HBM up into SMEM and Registers/TMEM, run arithmetic on them, and write results back down to HBM. High performance therefore requires **staging tiles on-chip and reusing them many times before writing back to HBM**. We will quantify the dramatic speed difference between reusing tiles versus streaming once from HBM in Section 8.

CUDA Asynchronous Execution and Benchmarking

Once tensors live in GPU HBM, how does Python actually run operations on the Blackwell SMs? A crucial systems trap lurks here: **CUDA kernel launches are asynchronous**.

Visualize two parallel timelines running horizontally from left to right—**CPU thread** on top, and **GPU stream** underneath:

1. On the CPU timeline, Python executes `launch K1` (a brief blue block), then immediately executes `launch K2` (another brief blue block). All the CPU did was push two work descriptors onto a command queue (the CUDA stream) and return immediately—it did *not* wait for the GPU to finish either kernel.
2. Down on the GPU stream timeline, slightly after `launch K1`, the GPU begins executing the long yellow block `Kernel 1 executing`, and only after `Kernel 1` finishes does it begin the long yellow block `Kernel 2 executing`.
3. If Python reads a wall-clock timer right after enqueueing `K2`, the timer stops while `Kernel 1` is still running and `Kernel 2` has not even started! You just measured Python's CPU launch overhead (a few microseconds), *not* the GPU's execution time (potentially milliseconds).
4. To measure true GPU runtime, the CPU must call `torch.cuda.synchronize()`. On the CPU timeline, this inserts a blocking wait barrier (a red dashed region) that halts the CPU thread until the GPU finishes `Kernel 2` and signals completion back upward. Only *after* `torch.cuda.synchronize()` returns can the CPU validly execute `read timer`.

COMMON MISCONCEPTION

You might think that wrapping a PyTorch matrix multiply with `t0 = time.time(); y = x @ w; t1 = time.time()` measures the matrix multiply's speed, and then wonder why your GPU appears to achieve $100{,}000\ \text{TFLOP}/\text{s}$—well above the physical limit of the silicon. That illusion happens precisely because you timed the asynchronous enqueue rather than the GPU computation.

Correct GPU benchmarking always follows two rules: **warm up first** (to absorb one-time CUDA context initialization, memory allocation, and kernel compilation overheads) and **synchronize both before and after the timed block**:

```python
def benchmark(func, num_trials=5):
    torch.cuda.synchronize()       # Wait for any prior GPU work to finish
    def run():
        func()                     # Enqueue GPU kernels
        torch.cuda.synchronize()   # Wait for those kernels to actually complete
    return timeit.timeit(run, number=num_trials) / num_trials
```

WORKED EXAMPLE

Worked Example (Timing Illusion vs. True Runtime): Suppose `func()` launches two large BF16 matrix multiplications (`K1` and `K2`), each taking 4.0 ms on the GPU, while each CPU kernel launch takes 0.015 ms ($15\ \mu$s).

- **Without `torch.cuda.synchronize()`:** Python executes `launch K1` (0.015 ms) and `launch K2` (0.015 ms), then immediately stops the timer at 0.030 ms—an error of more than $266\times$!
- **With `torch.cuda.synchronize()`:** The CPU waits until both kernels finish executing sequentially on the GPU stream. Ignoring the tiny initial dispatch lag before `K1` starts, the timer records 8.0 ms, accurately reflecting the real GPU compute duration.

We will need accurate measured runtimes t_{actual} from this `benchmark` pattern when we compute real throughput and MFU in Section 6. Before counting FLOPs, however, we need a clean, bug-resistant way to write multi-dimensional tensor operations so that every contracted axis and memory layout assumption is completely transparent.

✎ Check Your Understanding

1. **What Would Happen If:** You benchmark a GPU matrix multiplication in PyTorch using `timeit` *without* calling `torch.cuda.synchronize()` inside the timed function. Would the reported time be too large or too small, and why?
   Answer: Far too small. Because CUDA kernel launches are asynchronous, the CPU only measures the microseconds spent enqueueing the kernel onto the GPU stream rather than waiting for the GPU to actually execute the math.
2. **Fill in the Blank:** Inside each Blackwell SM, scalar/vector operations run on CUDA cores, matrix multiply-accumulate runs on Tensor Cores, and Blackwell adds dedicated on-chip _____ to hold Tensor Core accumulators without pressuring the register file.
   Answer: Tensor Memory (TMEM).

Section 5: Expressing Tensor Operations Explicitly with `einops`

KEY IDEA

In high-rank deep learning tensors—such as the rank-4 attention tensor `(B, S, H, D)` from Section 2—standard PyTorch syntax relies on anonymous integer axis indices (`dim=-1`, `transpose(-2, -1)`). This creates two problems: silent correctness bugs (transposing the wrong pair of equally sized axes produces no runtime error) and opaque resource accounting (you cannot tell at a glance which axes are being multiplied and summed, so counting FLOPs requires manually tracing shapes backward through the code).

Consider a simple batched attention score computation written in raw PyTorch:

```
x = torch.ones(2, 3, 4)      # batch, seq1, hidden
y = torch.ones(2, 3, 4)      # batch, seq2, hidden
z = x @ y.transpose(-2, -1)  # what are -2 and -1?
```

Looking only at the third line, `-2` and `-1` carry no semantic meaning. If someone adds an attention-head axis to `x` and `y` later, or swaps `seq` and `hidden`, `-2` and `-1` may silently contract the wrong dimension.

Named Contractions with `einsum`

DEFINITION

The **`einops`** library replaces anonymous integer indices with explicit **axis names** written directly inside an operation string. Rewriting the exact same batched inner product using `einops.einsum`:

```
z = einsum(x, y, "batch seq1 hidden, batch seq2 hidden -> batch seq1 seq2")
z = einsum(x, y, "... seq1 hidden, ... seq2 hidden -> ... seq1 seq2")
```

Two rules completely govern how `einsum` works:

DEFINITION

1. **Automatic contraction:** Any named axis that appears on the left-hand side of `->` (such as `hidden`) and is **absent** from the right-hand side is multiplied elementwise across the inputs and **summed over** automatically.

DEFINITION

2. **Ellipsis broadcasting (`...`):** The ellipsis `...` matches zero or more arbitrary leading batch dimensions (such as `batch`, or `(batch, heads)`) and preserves them identically in the output.

WORKED EXAMPLE

Worked Example 1 (Tracing Shapes and Sums in `einsum`): Let `x` have shape `(2, 3, 4)` (`batch=2, seq1=3, hidden=4`) and `y` have shape `(2, 3, 4)` (`batch=2, seq2=3, hidden=4`), both filled with `1.0`. In `einsum(x, y, "batch seq1 hidden, batch seq2 hidden -> batch seq1 seq2")`:

- Output axes are `(batch, seq1, seq2)`, so `z` has shape `(2, 3, 3)`.
- For each output entry $z_{b,i,j}$, the missing axis `hidden` (length 4) is contracted:

$$z_{b,i,j} = \sum_{h=0}^{3} x_{b,i,h}\, y_{b,j,h} = 1 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 = 4.0$$

- Notice how the string *immediately* exposes the arithmetic cost: there are $2 \times 3 \times 3 = 18$ output entries, each summing over 4 products.

Named Aggregations (`reduce`) and Dimension Splitting/Merging (`rearrange`)

Beyond contractions, neural networks constantly aggregate over axes (sums, means, maxes) and pack or unpack dimensions (for instance, splitting a hidden dimension into multiple attention heads and merging them back). `einops` provides two matching primitives:

DEFINITION

1. **`reduce`** performs named aggregations (`"sum"` , `"mean"` , `"max"` , `"min"`):

```
# Old: fragile axis index
y = x.sum(dim=-1)

# Einops: self-documenting
y = reduce(x, "... hidden -> ...", "sum")
```

Again, the dropped axis `hidden` is the axis being reduced, and the final string argument specifies the reduction operation.

DEFINITION

2. **`rearrange`** splits, merges, or permutes axes—using parentheses `(heads h1)` to indicate a single axis formed by packing two logical dimensions together:

```
x = torch.ones(3, 8)   # seq=3, (heads * h1)=8
w = torch.ones(4, 4)   # h1=4, h2=4

x = rearrange(x, "... (heads h1) -> ... heads h1", heads=2)
x = einsum(x, w, "... h1, h1 h2 -> ... h2")
x = rearrange(x, "... heads h2 -> ... (heads h2)")
```

WORKED EXAMPLE

Worked Example 2 (End-to-End Multi-Head Projection Trace): Trace every tensor shape and value through the three `einops` lines above:

1. Initial `x` has shape `(3, 8)` (all ones), and `w` has shape `(4, 4)` (all ones).
2. `rearrange(x, "... (heads h1) -> ... heads h1", heads=2)` splits the packed last axis of size 8 into `heads=2` and `h1 = 8 / 2 = 4` . The new shape of `x` is `(3, 2, 4)` .
3. `einsum(x, w, "... h1, h1 h2 -> ... h2")` matches `...` to `(3, 2)` , contracts `h1` (length 4), and outputs `h2` (length 4). The shape stays `(3, 2, 4)` , and each element equals $\sum_{k=0}^{3} 1 \cdot 1 = 4.0$.
4. `rearrange(x, "... heads h2 -> ... (heads h2)")` merges `heads=2` and `h2=4` back into a single packed axis of size $2 \times 4 = 8$. Final shape: `(3, 8)` , all entries equal to `4.0` .

Memory Accounting Note: Views vs. Copies in `rearrange`

COMMON MISCONCEPTION

You might think that every call to `rearrange` allocates a new tensor in GPU HBM and copies all the bytes around. That is not always true, and knowing the difference matters for memory traffic accounting.

KEY IDEA

When `rearrange` only splits or merges adjacent dimensions on a **contiguous** tensor—like `(3, 8) -> (3, 2, 4)` and `(3, 2, 4) -> (3, 8)` above—the flat sequence of numbers stored in HBM does not change order at all! PyTorch simply creates a lightweight **tensor view** by updating the tensor's shape and stride metadata, performing **zero HBM copies**. However, when `rearrange` **transposes** axes (reordering them in a way that breaks contiguity, such as `"b s h d -> b h s d"`), the memory layout is no longer contiguous along the new axes; downstream matrix-multiplication kernels (GEMMs) often require a contiguous layout and may trigger an explicit HBM copy before running.

With explicit contraction strings in hand, we are ready to formalize how to count arithmetic work (FLOPs), measure hardware speed (FLOP/s), and evaluate how much of a Blackwell GPU's peak compute our workloads actually achieve.

✎ Check Your Understanding

1. **Quick Trace:** Given `a` of shape `(8, 16, 64)` (`batch, seq, d_in`) and `b` of shape `(64, 128)` (`d_in, d_out`), what is the output shape of `einsum(a, b, "batch seq d_in, d_in d_out -> batch seq d_out")`, and which axis is summed over?
   Answer: The output shape is `(8, 16, 128)`, and `d_in` (length 64) is summed over because it appears on the left-hand side of `->` but is absent from the right-hand side.
2. **Yes/No with Justification:** Does splitting a contiguous tensor of shape `(3, 8)` into `(3, 2, 4)` using `rearrange` copy data in GPU HBM?
   Answer: No. Splitting or merging adjacent dimensions of a contiguous tensor does not alter the underlying order of elements in memory, so PyTorch returns a tensor view by updating metadata with zero HBM copies.

Section 6: Counting Compute: Matrix Multiplication FLOPs, Hardware Peak, and MFU

To evaluate whether a training run is fast or slow, we must cleanly separate *how much mathematical work an algorithm requires* from *how fast a piece of hardware executes that work*.

FLOPs vs. FLOP/s and Training Scale

Two nearly identical-looking abbreviations have completely different physical units—confusing them is one of the most frequent accounting errors:

DEFINITION

- **FLOPs** (plural, lowercase "s"): a **count** of floating-point operations (such as a single addition $x + y$ or a single multiplication $x \cdot y$). This measures total **work** (dimensionless count, analogous to total distance).

DEFINITION

- **FLOP/s** (also written **FLOPS**, uppercase "S"): floating-point operations **per second**. This measures **hardware speed** or throughput (rate, analogous to velocity).

To build numerical intuition for frontier training scales, consider two historical benchmarks alongside two-week cluster budgets:

- **Historical frontier training runs (total work):**
  - GPT-3 (2020): 3.14×10^{23} FLOPs
  - GPT-4 (2023, speculated): 2×10^{25} FLOPs
- **Cluster compute budgets over 2 weeks ($14 \text{ days} \times 86{,}400 \text{ s/day} = 1{,}209{,}600$ s) at dense BF16 peak:**
  - **8 Blackwell B200s** (2.25×10^{15} FLOP/s each):

    $$8 \times 14 \times 86{,}400 \text{ s} \times 2.25 \times 10^{15} \text{ FLOP/s} = 9{,}676{,}800 \times 2.25 \times 10^{15} \approx 2.18 \times 10^{22} \text{ FI}$$

  - **8 H100s** (historical baseline, 989.5×10^{12} FLOP/s each):

    $$8 \times 14 \times 86{,}400 \text{ s} \times 989.5 \times 10^{12} \text{ FLOP/s} \approx 9.58 \times 10^{21} \text{ FLOPs}$$

Even running 8 B200s at an idealized 100% utilization for two solid weeks produces 2.18×10^{22} FLOPs—less than one-tenth of GPT-3's total training compute.

Counting FLOPs for Matrix Multiplication

In every Transformer and deep neural network, the overwhelming majority of arithmetic work comes from **matrix multiplication** (often called **GEMM**, General Matrix Multiply).

Picture multiplying two matrices:

- Left input matrix X of shape $B \times D$ (B rows, D columns).
- Right weight matrix W of shape $D \times K$ (D rows, K columns).
- Output matrix $Y = XW$ of shape $B \times K$ (B rows, K columns).

Every single output entry $Y_{i,k}$ (for $1 \le i \le B$ and $1 \le k \le K$) is the inner product of row i of X with column k of W:

$$Y_{i,k} = \sum_{j=1}^{D} X_{i,j}\, W_{j,k}$$

How many floating-point operations does this require?

- For each triple (i, j, k), we perform **1 multiplication** ($X_{i,j} \cdot W_{j,k}$) and **1 addition** (accumulating that product into the running sum for $Y_{i,k}$).
- Strictly speaking, summing D terms takes D multiplications and $D - 1$ additions, giving $BK(2D - 1) = 2BDK - BK$ FLOPs. Because Tensor Core hardware executes fused multiply-add instructions into a zero-initialized accumulator (and because $D \gg 1$ in practice), the standard formula counts $2D$ FLOPs per output element across all $B \cdot K$ output elements:

$$\text{FLOPs}(XW) = 2BDK$$

KEY IDEA

Breaking the formula apart: $B \cdot K$ is the total number of output elements, D is the inner contraction length for each output element, and the factor of 2 accounts for one multiply plus one add per contraction step. Taken as a single conceptual unit, $2BDK$ **is the exact compute cost of multiplying a** (B, D) **matrix by a** (D, K) **matrix**.

DEFINITION

Once we time that matrix multiplication on a GPU (using the synchronized `benchmark` pattern from Section 4) and obtain a wall-clock runtime t_{actual} in seconds, the **measured throughput** is:

$$\text{actual FLOP/s} = \frac{2BDK}{t_{\text{actual}}}$$

WORKED EXAMPLE

Worked Example 1 (Large GPU Matrix Multiplication): Let $B = 16{,}384$ (2^{14}), $D = 32{,}768$ (2^{15}), and $K = 8{,}192$ (2^{13}).

1. **Total FLOPs:**

   $$\text{FLOPs} = 2 \times 16{,}384 \times 32{,}768 \times 8{,}192 = 2^1 \times 2^{14} \times 2^{15} \times 2^{13} = 2^{43} = 8{,}796{,}093{,}02$$

   (roughly 8.80 TFLOPs of work).

2. **Measured Throughput:** Suppose our synchronized benchmark measures $t_{\text{actual}} = 5.5 \text{ ms} = 5.5 \times 10^{-3}$ s on a single B200 in BF16. Then:

$$\text{actual FLOP/s} = \frac{8.80 \times 10^{12} \text{ FLOPs}}{5.5 \times 10^{-3} \text{ s}} = 1.60 \times 10^{15} \text{ FLOP/s} = 1{,}600 \text{ TFLOP/s}$$

Blackwell Peak Compute: Precision and Sparsity

To judge whether $1{,}600 \text{ TFLOP/s}$ is impressive or disappointing, we must compare it against the hardware's ceiling. Comparing three generations of NVIDIA data-center GPUs reveals two defining trends in AI hardware design:

GPU (Architecture)	FP32 Peak	BF16 / FP16 Dense	BF16 / FP16 Sparse (2:4)	BF16 Dense / FP32 Ratio
A100 (Ampere, *historical*)	19.5 TFLOP/s	312 TFLOP/s	—	$16\times$
H100 (Hopper, *historical*)	67.5 TFLOP/s	989.5 TFLOP/s	$1{,}979 \text{ TFLOP/s}$	$14.7\times$
B200 (Blackwell, current)	75 TFLOP/s	$2{,}250 \text{ TFLOP/s}$	$4{,}500 \text{ TFLOP/s}$	$30\times$

Three engineering lessons jump straight out of this table:

1. **Read datasheets carefully:** Marketing headlines quote the **2:4 structured sparsity** number ($4{,}500 \text{ TFLOP/s}$ on B200, $1{,}979 \text{ TFLOP/s}$ on H100), which only applies if at least 2 out of every 4 contiguous weights are zero. Standard dense GEMMs achieve exactly **half** ($2{,}250 \text{ TFLOP/s}$ on B200, 989.5 TFLOP/s on H100). This justifies the `/ 2` factor we used in Napkin Calculation I in Section 1.
2. **FP32 compute has stalled while Tensor Core compute exploded:** From H100 to B200, FP32 peak barely moved ($67.5 \to 75 \text{ TFLOP/s}$), whereas dense BF16 more than doubled ($989.5 \to 2{,}250 \text{ TFLOP/s}$), widening the BF16-to-FP32 gap to $30\times$. Running matmuls in FP32 on Blackwell throws away 97% of the GPU's compute capability.
3. **Blackwell pushes the low-precision bet even further** with native **FP8** and **NVFP4** Tensor Cores, pushing peak FLOP/s even higher than BF16.

Model FLOPs Utilization (MFU)

DEFINITION

We normalize measured throughput against hardware peak using **Model FLOPs Utilization (MFU)**:

$$\text{MFU} = \frac{\text{actual FLOP/s}}{\text{promised dense peak FLOP/s (for active dtype)}}$$

Breaking this ratio apart:

- The numerator (actual FLOP/s) counts *only* the useful model FLOPs divided by total wall-clock time—meaning any time spent waiting on memory, launching kernels, or communicating across GPUs enlarges the time denominator inside actual FLOP/s and pulls MFU down.
- The denominator is the hardware's **dense peak** for the active precision (e.g., $2{,}250\ \text{TFLOP/s}$ for B200 BF16).
- As a whole, MFU is a dimensionless fraction in $[0, 1]$ measuring what fraction of the Tensor Cores' theoretical ceiling actually turned into useful model arithmetic.

WORKED EXAMPLE

Worked Example 2 (Computing MFU): Continuing Worked Example 1, our large BF16 matmul achieved $1{,}600\ \text{TFLOP/s}$ on a B200 (dense BF16 peak $= 2{,}250\ \text{TFLOP/s}$):

$$\text{MFU} = \frac{1{,}600\ \text{TFLOP/s}}{2{,}250\ \text{TFLOP/s}} \approx 0.711\ (71.1\%)$$

In end-to-end training (which includes communication, non-matmul layers, and optimizer updates, not just one isolated huge GEMM), a rule of thumb is that $\text{MFU} \geq 0.5$ is considered strong.

KEY IDEA

Why is MFU rarely near 1.0? Because non-matmul operations (elementwise activations, normalizations, MoE routing) stall waiting on HBM, small batch or tile shapes under-utilize the SMs, and distributed collectives and kernel launches add overhead.

This raises the central diagnostic question of GPU performance engineering: given *any* operation—an activation function, a dot product, a small matmul, or a giant GEMM—how can we predict mathematically whether it will saturate Blackwell's Tensor Cores or stall waiting on HBM? Answering that requires the **two-resource bottleneck model**.

✎ Check Your Understanding

1. **Quick Computation:** Compute the total FLOPs for multiplying a matrix X of shape $(1{,}024, 4{,}096)$ by a weight matrix W of shape $(4{,}096, 2{,}048)$.
   Answer: $\text{FLOPs} = 2BDK = 2 \times 1{,}024 \times 4{,}096 \times 2{,}048 = 2 \times 2^{10} \times 2^{12} \times 2^{11} = 2^{34} \approx 1.72 \times 10^{10}$ FLOPs (17.2 GFLOPs).
2. **What Would Happen If:** An engineer accidentally switches a B200 BF16 dense matrix multiplication to run in FP32 instead, keeping the same matrix dimensions. By what maximum factor does the hardware's peak compute ceiling drop?
   Answer: It drops by a factor of $30\times$, from $2{,}250$ TFLOP/s (BF16 dense) down to 75 TFLOP/s (FP32 peak).

Section 7: The Two-Resource Execution Model and Arithmetic Intensity

KEY IDEA

In Section 6, we noted that some kernels run close to peak Tensor Core speed while others crawl with tiny MFU. Simply counting FLOPs cannot explain why—an operation with *fewer* FLOPs can easily take just as long as an operation with $20\times$ more FLOPs if both are waiting for bytes to arrive from HBM. We need a unified model that accounts for **both** arithmetic speed and memory bandwidth simultaneously.

The Two-Resource Execution Model

Recall from Section 4 that every PyTorch tensor lives in GPU HBM, while arithmetic executes on the Blackwell SMs. Any GPU kernel therefore performs three logical steps between two hardware blocks:

- **Left block — GPU HBM:** characterized by its memory bandwidth B_{HBM} (measured in bytes/s).
- **Right block — Blackwell SMs (Tensor Cores + CUDA Cores):** characterized by its peak compute speed F_{peak} (measured in FLOP/s).
- **Dataflow arrows between them:**
  1. Top arrow pointing right (HBM → SMs): **Read inputs** (bytes).
  2. Inside the SMs box: **Compute** (FLOPs).
  3. Bottom arrow pointing left (SMs → HBM): **Write outputs** (bytes).

DEFINITION

Two independent durations govern how long the kernel takes:

$$T_{\text{comm}} = \frac{\text{bytes moved}}{B_{\text{HBM}}}, \qquad T_{\text{comp}} = \frac{\text{FLOPs}}{F_{\text{peak}}}$$

Let's examine each expression:

- T_{comm} (communication or memory-transfer time, in seconds) is the total number of bytes read from *plus* written to HBM divided by the HBM bandwidth B_{HBM} (bytes/s). It is the minimum time required just to move the data across the HBM bus, even if arithmetic were infinitely fast.
- T_{comp} (compute time, in seconds) is the total FLOPs divided by the peak arithmetic speed F_{peak} (FLOP/s). It is the minimum time required to perform the math, even if memory bandwidth were infinite.

KEY IDEA

Modern GPUs overlap memory transfers and arithmetic—for example, using asynchronous Tensor Memory Accelerator (TMA) loads to prefetch the next tile from HBM while Tensor Cores compute on the current tile. Under ideal overlap, the faster resource finishes completely hidden behind the slower resource, so total execution time is dictated solely by the bottleneck:

$$T_{\text{total}} = \max(T_{\text{comm}},\ T_{\text{comp}})$$

COMMON MISCONCEPTION

You might think that total kernel time is the *sum* $T_{\text{comm}} + T_{\text{comp}}$, assuming the GPU finishes reading every byte of an entire tensor before performing a single multiply. That breaks on real GPUs because streaming execution pipelines loads and math simultaneously across thousands of threads and tiles; the $\max$ model represents the tightest idealized lower bound on runtime.

Arithmetic Intensity vs. Accelerator Intensity

Comparing T_{comm} and T_{comp} directly requires plugging in specific hardware numbers every time. By rearranging the inequality $T_{\text{comp}} > T_{\text{comm}}$, we cleanly separate the **workload's** intrinsic property from the **hardware's** intrinsic property:

$$\frac{\text{FLOPs}}{F_{\text{peak}}} > \frac{\text{bytes moved}}{B_{\text{HBM}}} \quad \Longleftrightarrow \quad \frac{\text{FLOPs}}{\text{bytes moved}} > \frac{F_{\text{peak}}}{B_{\text{HBM}}}$$

Each side of this rearranged inequality defines a fundamental ratio with units of FLOP/byte:

DEFINITION

- **Arithmetic Intensity (a property of the workload/operation alone):**

$$I_{\text{op}} = \frac{\text{FLOPs}}{\text{bytes moved}} \quad \left[\frac{\text{FLOP}}{\text{byte}}\right]$$

This measures how many floating-point operations the algorithm extracts from every single byte fetched from or written to HBM.

DEFINITION

- **Accelerator Intensity (a property of the GPU hardware and precision alone, also called the "ridge point"):**

$$I_{\text{hw}} = \frac{F_{\text{peak}}}{B_{\text{HBM}}} \quad \left[\frac{\text{FLOP}}{\text{byte}}\right]$$

This measures how many FLOPs the GPU *can* execute in the time it takes to transfer one byte across HBM. It is the exact threshold of work-per-byte required to keep the compute units fed.

Comparing I_{op} against I_{hw} immediately classifies any kernel into one of two regimes:

DEFINITION

- **Memory-bound ($T_{\text{comm}} > T_{\text{comp}}$, equivalently $I_{\text{op}} < I_{\text{hw}}$):** The kernel does too little math per byte transferred. Tensor Cores sit idle waiting on HBM. Speeding up the math (or deleting arithmetic ops) barely changes runtime; only moving fewer bytes speeds it up.

DEFINITION

- **Compute-bound ($T_{\text{comp}} > T_{\text{comm}}$, equivalently $I_{\text{op}} > I_{\text{hw}}$):** The kernel performs enough math per byte that HBM transfers finish well before the arithmetic does. Arithmetic saturates the GPU, and Tensor Cores run at full tilt.

WORKED EXAMPLE

Worked Example (Hardware Ridge Point on H100 vs. Blackwell Trend): Compute the BF16 accelerator intensity I_{hw} for the historical H100 GPU, which has dense BF16 peak $F_{\text{peak}} = 989.5 \text{ TFLOP/s} = 989.5 \times 10^{12} \text{ FLOP/s}$ and HBM bandwidth $B_{\text{HBM}} = 3.35 \text{ TB/s} = 3.35 \times 10^{12} \text{ bytes/s}$:

$$I_{\text{hw, H100 BF16}} = \frac{989.5 \times 10^{12} \text{ FLOP/s}}{3.35 \times 10^{12} \text{ bytes/s}} \approx 295 \text{ FLOP/byte}$$

> Interpretation: to saturate an H100's BF16 Tensor Cores, a kernel must perform **at least** 295 **floating-point operations for every single byte** moved to or from HBM. Anything below $295\ \text{FLOP}/\text{byte}$ leaves the Tensor Cores starved.
>
> Now consider what happens on Blackwell: across GPU generations, Tensor Core throughput (F_{peak}) scales even faster than HBM bandwidth (B_{HBM})—especially when stepping down to **FP8** and **NVFP4**, where F_{peak} doubles or quadruples while B_{HBM} stays fixed. That pushes the ridge point I_{hw} even higher, making it *harder* for low-reuse operations to keep Blackwell's Tensor Cores fed.

To see what this means in practice, we now compute the exact arithmetic intensity I_{op} of every core neural network operation—from ReLU and GELU to dot products, matrix–vector products, and tiled matrix–matrix multiplications—and plot them together on the **Roofline Model**.

> **✎ Check Your Understanding**
>
> 1. **Fill in the Blank:** A kernel whose arithmetic intensity I_{op} is strictly less than the hardware's accelerator intensity I_{hw} is called _____, meaning its wall-clock time is governed by T_{comm} rather than T_{comp}.
>   **Answer:** memory-bound (or HBM-bandwidth bound). When $I_{\text{op}} < I_{\text{hw}}$, $T_{\text{comm}} > T_{\text{comp}}$, so HBM transfers dominate total runtime.
> 2. **Yes/No with Justification:** Suppose a GPU upgrade doubles peak Tensor Core throughput F_{peak} while keeping HBM bandwidth B_{HBM} constant. Does a kernel with fixed I_{op} become easier to make compute-bound?
>   **Answer:** No—it becomes *harder*. Doubling F_{peak} at constant B_{HBM} doubles the ridge point $I_{\text{hw}} = F_{\text{peak}}/B_{\text{HBM}}$, requiring twice as many FLOPs per byte to reach the compute-bound regime.

Section 8: Diagnosing Real Operations on the Roofline

With I_{op} and I_{hw} defined in Section 7, we can test every core building block of a language model and immediately expose several counterintuitive GPU behaviors.

Elementwise Operations: Why ReLU Is Not Faster Than GELU

Consider applying an elementwise activation function to a BF16 tensor $x \in \mathbb{R}^n$ (for example, $n = 1{,}024 \times 1{,}024 = 1{,}048{,}576$ elements). Because BF16 uses $2\ \text{bytes}/\text{element}$, the kernel must read x from HBM ($2n$ bytes) and write the output y back to HBM ($2n$ bytes), moving a total of $4n$ bytes regardless of which activation function is computed.

Now compare **ReLU** against **GELU**:

- **ReLU ($y = \max(0, x)$):** Requires 1 comparison per element, so $\text{FLOPs} = n$.

$$I_{\text{op, ReLU}} = \frac{n \text{ FLOPs}}{4n \text{ bytes}} = 0.25 \text{ FLOP/byte}$$

- **GELU ($y = 0.5\,x\,(1 + \tanh(\sqrt{2/\pi}(x + 0.044715x^3)))$):** Requires roughly 20 operations per element, so $\text{FLOPs} \approx 20n$.

$$I_{\text{op, GELU}} = \frac{20n \text{ FLOPs}}{4n \text{ bytes}} = 5 \text{ FLOP/byte}$$

WORKED EXAMPLE

Worked Example 1 (Wall-Clock Time of ReLU vs. GELU): Evaluate both kernels on $n = 1{,}048{,}576$ BF16 elements using our H100 baseline ($B_{\text{HBM}} = 3.35$ TB/s, $F_{\text{peak}} = 989.5$ TFLOP/s, $I_{\text{hw}} \approx 295$ FLOP/byte):

- Total bytes moved for *both* kernels: $4n = 4 \times 1{,}048{,}576 \approx 4.194 \times 10^6$ bytes.
- T_{comm} for *both*: $\dfrac{4.194 \times 10^6 \text{ bytes}}{3.35 \times 10^{12} \text{ bytes/s}} \approx 1.25\,\mu\text{s}$.
- T_{comp} for ReLU: $\dfrac{1.049 \times 10^6 \text{ FLOPs}}{989.5 \times 10^{12} \text{ FLOP/s}} \approx 0.00106\,\mu\text{s}$.
- T_{comp} for GELU: $\dfrac{20.97 \times 10^6 \text{ FLOPs}}{989.5 \times 10^{12} \text{ FLOP/s}} \approx 0.0212\,\mu\text{s}$.
- Applying $T_{\text{total}} = \max(T_{\text{comm}}, T_{\text{comp}})$: ReLU takes $1.25\,\mu$s and GELU takes $1.25\,\mu$s!

COMMON MISCONCEPTION

You might think that because GELU performs $20\times$ more arithmetic than ReLU, ReLU must run $20\times$ faster on a GPU. That intuition breaks completely because both 0.25 and 5 FLOP/byte are orders of magnitude below the GPU's ridge point I_{hw}. Both kernels are deeply **memory-bound**: the HBM transfer time completely hides the extra $\tanh$ arithmetic. Speeding up elementwise ops requires **kernel fusion**—combining the activation directly into the preceding or following kernel while data is still on-chip in registers/SMEM so that the extra $4n$-byte round-trip to HBM never happens at all.

Dot Products and Matrix–Vector Products: Why LLM Decode Is Memory-Bound

Next, test linear algebra operations on vectors in BF16 (2 bytes/element):

1. **Dot product ($x \cdot w$, vectors of length n):**
   - Bytes moved: read x ($2n$ B) + read w ($2n$ B) + write scalar output (2 B) $= 4n + 2 \approx 4n$ bytes.
   - Arithmetic: n multiplies + $(n-1)$ adds $= 2n - 1 \approx 2n$ FLOPs.

- Arithmetic intensity:

$$I_{\text{op}} = \frac{2n-1}{4n+2} \approx 0.5\ \text{FLOP/byte} \implies \textbf{memory-bound}$$

2. **Matrix–vector product (xW, where $x \in \mathbb{R}^{1\times n}$ and $W \in \mathbb{R}^{n\times n}$):**
   - Bytes moved: read x ($2n$ B) + read W ($2n^2$ B) + write output ($2n$ B) $= 2n^2 + 4n \approx 2n^2$ bytes.
   - Arithmetic: n dot products of length n, giving $n(2n-1) = 2n^2 - n \approx 2n^2$ FLOPs.
   - Arithmetic intensity:

$$I_{\text{op}} = \frac{2n^2-n}{2n^2+4n} \approx 1\ \text{FLOP/byte} \implies \textbf{memory-bound}$$

Why is matrix–vector multiplication stuck at just $1\ \text{FLOP/byte}$ no matter how large n grows? Because **every weight W_{ij} (2 bytes) is fetched all the way from HBM to perform just 2 FLOPs** (one multiply and one add), and is immediately discarded—zero weight reuse across tokens!

> **KEY IDEA**
> This explains a defining bottleneck of modern LLM deployment: **autoregressive inference decoding** generates one new token per sequence at a time. Passing a single token vector through every linear layer is a chain of **matrix–vector products** (plus reading the entire past KV cache from HBM). At batch size 1, LLM decode sits around $1\ \text{FLOP/byte}$—fundamentally **HBM-bandwidth bound**, leaving over 99% of Blackwell's Tensor Cores idle unless multiple user requests are **batched** together.

Matrix–Matrix Multiply: On-Chip Tiling Unlocks Compute-Bound Speed

How do we escape the $1\ \text{FLOP/byte}$ ceiling? By multiplying a **batch** of vectors (a matrix X) against W, and staging tiles in on-chip memory so each loaded weight is reused across many rows of X.

Picture the on-chip tiling diagram on a Blackwell SM:

- At the top sits **GPU HBM**, holding three $n \times n$ BF16 matrices: X, W, and output Y. Total HBM traffic if each matrix is moved once is $2n^2 + 2n^2 + 2n^2 = 6n^2$ bytes.
- Downward and upward arrows connect HBM to the **Blackwell SM (SMEM + TMEM + Tensor Cores)** below, labeled `load tiles` and `store tile`.
- Inside the SM, small tiles (`X tile` $\times$ `W tile` $\rightarrow$ `Y tile`) are loaded into SMEM, multiplied repeatedly on Tensor Cores while accumulating partial sums into TMEM, and written back to HBM only once a `Y tile` is complete.

Now compute the arithmetic intensity of a **square BF16 matrix–matrix multiply** ($n \times n$):

- Bytes moved: $2n^2 + 2n^2 + 2n^2 = 6n^2$ bytes.
- Arithmetic: $2n^3 - n^2 \approx 2n^3$ FLOPs.

- Arithmetic intensity:

$$I_{\text{op}} = \frac{2n^3}{6n^2} = \frac{n}{3}\ \text{FLOP/byte}$$

Unlike ReLU (0.25), dot product (0.5), and MatVec (1), which stay flat constants as n grows, **matrix–matrix multiplication's arithmetic intensity grows linearly with n!** At $n = 1{,}024$:

$$I_{\text{op}} = \frac{1{,}024}{3} \approx 341\ \text{FLOP/byte}$$

which clears the H100's $295\ \text{FLOP/byte}$ ridge point and enters the **compute-bound** regime.

For a general **rectangular GEMM** multiplying $X \in \mathbb{R}^{B\times D}$ by $W \in \mathbb{R}^{D\times K}$ stored in b bytes/element (e.g., $b = 2$ for BF16, $b = 1$ for FP8):

$$I_{\text{op}} = \frac{2BDK}{b\,(BD + DK + BK)}$$

Let's break this equation down term by term:

- Numerator $2BDK$: total FLOPs (derived in Section 6).
- Inner sum $BD + DK + BK$: total number of scalar elements across input X (BD), weight W (DK), and output Y (BK).
- Multiplying by b converts total elements into total bytes moved across HBM.
- Dividing numerator and denominator by BDK reveals an illuminating equivalent form:

$$I_{\text{op}} = \frac{2}{b\left(\frac{1}{K} + \frac{1}{B} + \frac{1}{D}\right)}$$

  This reciprocal sum behaves like a harmonic mean: **it is dominated by the smallest of the three dimensions B, D, K.**

WORKED EXAMPLE

Worked Example 2 (Why Batch Size B Flips Decode Into Compute-Bound Prefill/Training): Let $D = K = 4{,}096$ in BF16 ($b = 2$). Compare $B = 1$ (unbatched decode) against $B = 512$ (training or batched prefill):

- **At $B = 1$:**

$$I_{\text{op}} = \frac{2}{2\left(\frac{1}{4096} + \frac{1}{1} + \frac{1}{4096}\right)} = \frac{1}{1 + \frac{2}{4096}} \approx 1.0\ \text{FLOP/byte} \quad \text{(severely memory-bound}$$

- **At $B = 512$:**

$$I_{\text{op}} = \frac{2}{2\left(\frac{1}{4096} + \frac{1}{512} + \frac{1}{4096}\right)} = \frac{1}{\frac{10}{4096}} = 409.6\ \text{FLOP/byte} \quad \text{(compute-bound)}$$

> Notice the dramatic jump: because training and inference **prefill** process many tokens in parallel ($B \gg 1$), staging tiles in SMEM/TMEM reuses each fetched weight across B tokens, turning $1\ \text{FLOP/byte}$ matvecs into compute-bound GEMMs. We will return to this exact B-dependence in Section 11 when analyzing how small a micro-batch can safely get during gradient accumulation.

The Roofline Model and MFU Ceiling

DEFINITION

All of these regimes come together on a single log–log plot known as the **Roofline Model**:

- **Horizontal axis (log scale):** Arithmetic Intensity I_{op} in FLOP/byte.
- **Vertical axis (log scale):** Attainable throughput in FLOP/s.
- **The Roofline Curve:** For any GPU, attainable throughput is bounded by both memory bandwidth ($B_{\text{HBM}} \cdot I_{\text{op}}$) and peak compute ($F_{\text{peak}}$):

$$\text{Attainable FLOP/s} = \min(F_{\text{peak}},\ B_{\text{HBM}} \cdot I_{\text{op}})$$

  On a log–log plot, this produces a two-segment roof:

  1. **Left of the ridge point (Memory-Bound Region):** A sloped ramp rising linearly with I_{op}, whose height is set by HBM bandwidth ($B_{\text{HBM}} \cdot I_{\text{op}}$).
  2. **Right of the ridge point (Compute-Bound Region):** A flat horizontal ceiling capped at the Tensor Core peak (F_{peak}).
  3. **The corner ("ridge point"):** Occurs precisely at $I_{\text{op}} = I_{\text{hw}} = F_{\text{peak}}/B_{\text{HBM}}$, where the sloped ramp hits the horizontal ceiling.

- **Comparing GPUs on the plot:** The historical **H100 BF16** roofline (dashed grey line, ridge $\approx 295\ \text{FLOP/byte}$) sits below and to the left of the **Blackwell B200 BF16** roofline (solid blue line, flat ceiling at $2{,}250\ \text{TFLOP/s}$ dense, higher ramp from faster HBM, and a ridge point shifted further right).
- **Placing our workloads along the x-axis:** Reading left to right along the ramp, **ReLU** (0.25) sits near the bottom-left, **MatVec** (1) sits slightly higher, and **GELU** (5) sits further up the ramp—all three pinned firmly inside the red memory-bound region far below peak compute. Far to the right of I_{hw}, **Large GEMM** hits the green flat compute-bound ceiling.

DEFINITION

Dividing the roofline equation $\min(F_{\text{peak}}, B_{\text{HBM}} \cdot I_{\text{op}})$ by F_{peak} and recalling that $F_{\text{peak}}/B_{\text{HBM}} = I_{\text{hw}}$ gives the theoretical **Roofline MFU ceiling**:

$$\text{MFU}_{\text{roofline}} = \min\left(1,\ \frac{I_{\text{op}}}{I_{\text{hw}}}\right)$$

KEY IDEA

Let's read this compact formula as a single conceptual unit: **an operation's best possible MFU is capped by the ratio of its arithmetic intensity to the hardware's ridge point, saturated at 1 once it crosses the ridge.** To raise MFU for any operation stranded to the left of the ridge, you must push it rightward via **batching** (increasing B), **tiling** (keeping reuse on-chip), **quantization** (cutting b), or **kernel fusion** (eliminating intermediate HBM round-trips).

Summary Across Core BF16 Operations

Operation (BF16)	Bytes Moved	FLOPs	I_{op} (FLOP/byte)	Regime
ReLU (n)	$4n$	n	0.25	Memory-bound
Dot product (n)	$4n + 2$	$2n - 1$	≈ 0.5	Memory-bound
Matrix–vector ($n \times n$)	$2n^2 + 4n$	$2n^2 - n$	≈ 1	Memory-bound
GELU (n)	$4n$	$\approx 20n$	≈ 5	Memory-bound
Matrix–matrix ($n \times n$)	$6n^2$	$2n^3 - n^2$	$\approx n/3$ (341 at $n = 1024$)	**Compute-bound**

One subtle effect of **lower precision (FP8 / NVFP4)** deserves attention: shrinking element size b halves or quarters bytes moved (pushing I_{op} *upward*), while simultaneously unlocking faster FP8/NVFP4 Tensor Core peaks (pushing I_{hw} *upward* too). Thus *both* I_{op} and I_{hw} shift upward together; for memory-bound decode, cutting bytes directly boosts wall-clock tokens/second, while for compute-bound GEMMs, higher F_{peak} directly accelerates training!

We now understand single-operation compute and memory completely. Next, we assemble multiple layers into a full neural network and account for the forward pass, PyTorch autograd, and the backward pass—finally deriving the $6BP$ (and $6ND$) rule we used back in Section 1.

✎ Check Your Understanding

1. **What Would Happen If:** You replace every isolated ReLU activation in a BF16 model with GELU (which uses $20\times$ more FLOPs per element), without fusing either activation into neighboring kernels. How much slower will the activation kernels run?
   Answer: Virtually no slower (the same wall-clock speed). Both ReLU ($0.25\ \text{FLOP}/\text{byte}$) and GELU ($5\ \text{FLOP}/\text{byte}$) sit far below I_{hw} and move the exact same $4n$ bytes across HBM, so HBM transfer time hides the extra GELU arithmetic.
2. **Quick Computation:** On an H100 ($I_{\text{hw}} \approx 295\ \text{FLOP}/\text{byte}$), what is the theoretical maximum $\text{MFU}_{\text{roofline}}$ attainable by a BF16 matrix–vector product ($I_{\text{op}} \approx 1\ \text{FLOP}/\text{byte}$)?
   Answer: $\text{MFU}_{\text{roofline}} = \min(1, 1/295) \approx 0.0034$ (about 0.34%), illustrating why unbatched LLM decoding leaves almost the entire Tensor Core array idle.

Section 9: Autograd, Backward Pass Anatomy, and the $6BP$ Compute Rule

So far we have counted FLOPs for a single forward matrix multiplication ($2BDK$). Training a neural network, however, requires not just computing the model's output (the **forward pass**), but also computing the gradient of the scalar loss with respect to every trainable parameter (the **backward pass**). Where does the factor of 6 in Section 1's $6ND$ formula come from, and what does backpropagation force us to keep in GPU HBM?

Deep Network Model and PyTorch Autograd

To perform exact accounting without unnecessary clutter, consider an **L-layer network of uniform width D**, where each layer is a `Block` applying a $D \times D$ matrix multiply followed by a ReLU activation:

```python
class Block(nn.Module):
    def __init__(self, dim: int):
        super().__init__()
        self.weight = nn.Parameter(
            torch.randn(dim, dim) / math.sqrt(dim))

    def forward(self, x):
        return F.relu(x @ self.weight)

# Forward + backward
loss = F.mse_loss(model(x).mean(), y)
loss.backward()
```

Each layer has $D \times D = D^2$ parameters, so across L layers the total parameter count is:

$$P = LD^2$$

(Note that `torch.randn(dim, dim) / math.sqrt(dim)` scales initial weights by $1/\sqrt{D}$ so that activation variance stays bounded across layers rather than exploding with D.)

DEFINITION

During the **forward pass**, PyTorch evaluates the sequence of activations $h_1, h_2, \ldots, h_L$ while dynamically recording every operation into a directed computation graph (**autograd**). Calling `loss.backward()` walks that graph in reverse, applying the chain rule at each node and populating the `.grad` attribute of every `nn.Parameter`.

WORKED EXAMPLE

Worked Example 1 (Toy Autograd Check): Verify the chain rule on a single linear unit with scalar loss $L(w) = \frac{1}{2}(x \cdot w - 5)^2$, given input $x = [1, 2, 3]$ and weight $w = [1, 1, 1]$:

1. **Forward pass:**
   - Inner product: $z = x \cdot w = 1(1) + 2(1) + 3(1) = 6$.
   - Residual: $r = z - 5 = 6 - 5 = 1$.
   - Loss: $L = \frac{1}{2}r^2 = \frac{1}{2}(1)^2 = 0.5$.
2. **Backward pass (reverse chain rule):**
   - Gradient with respect to z: $\dfrac{\partial L}{\partial z} = r = (x \cdot w - 5) = 1$.
   - Gradient with respect to w: since $\dfrac{\partial z}{\partial w_i} = x_i$,

$$\nabla_w L = \frac{\partial L}{\partial z} \cdot x = 1 \cdot [1, 2, 3] = [1, 2, 3]$$

Look closely at the last line: computing $\nabla_w L$ required the **forward input** x. That exact dependency scales up to entire activation matrices in deep networks.

Backward Pass Anatomy: Two GEMMs per Layer

Now zoom into a single linear layer inside our L-layer network taking a batch of B input vectors:

$$h_2 = h_1 W_2$$

where $h_1 \in \mathbb{R}^{B \times D}$, $W_2 \in \mathbb{R}^{D \times D}$, and $h_2 \in \mathbb{R}^{B \times D}$.

Picture the two-row forward/backward flow diagram for this layer:

- **Top row (Forward, blue):** `h1 (B, D)` feeds rightward into the operation box `h2 = h1 W2 (2 B D^2 FLOPs)`, producing `h2 (B, D)`.

- **Connecting dashed arrow dropping straight down from `h1`:** labeled **`saved h1 in HBM!`**—this arrow routes `h1` all the way down into the backward weight-gradient box.
- **Bottom row (Backward, red):** Upstream gradient `∇_{h2} L` (shape $B \times D$) enters from the right and **splits into two separate GEMM boxes**:
  1. **Activation gradient (passed to the previous layer):** $\nabla_{h_1} L = (\nabla_{h_2} L)\, W_2^\top$ ($2BD^2$ FLOPs).
  2. **Weight gradient (used to update W_2):** $\nabla_{W_2} L = h_1^\top (\nabla_{h_2} L)$ ($2BD^2$ FLOPs).

Let's derive both backward matrix formulas cleanly using index notation so no step is taken on faith. Writing the forward product entry-by-entry:

$$(h_2)_{b,j} = \sum_{i=1}^{D} (h_1)_{b,i}\, (W_2)_{i,j}$$

By the multivariable chain rule:

1. **Gradient with respect to input activations h_1:** Entry $(h_1)_{b,i}$ influences $(h_2)_{b,j}$ across all output columns $j = 1, \ldots, D$ with partial derivative $(W_2)_{i,j}$. Summing over j:

   $$\frac{\partial L}{\partial (h_1)_{b,i}} = \sum_{j=1}^{D} \frac{\partial L}{\partial (h_2)_{b,j}}\, (W_2)_{i,j} = \Big((\nabla_{h_2} L)\, W_2^\top \Big)_{b,i}$$

   This multiplies a (B, D) matrix by a (D, D) matrix $\implies$ $\mathbf{2BD^2}$ FLOPs.

2. **Gradient with respect to weights W_2:** Entry $(W_2)_{i,j}$ influences $(h_2)_{b,j}$ across all batch examples $b = 1, \ldots, B$ with partial derivative $(h_1)_{b,i}$. Summing over b:

   $$\frac{\partial L}{\partial (W_2)_{i,j}} = \sum_{b=1}^{B} (h_1)_{b,i}\, \frac{\partial L}{\partial (h_2)_{b,j}} = \Big(h_1^\top (\nabla_{h_2} L) \Big)_{i,j}$$

   This multiplies a (D, B) matrix by a (B, D) matrix $\implies$ $\mathbf{2BD^2}$ FLOPs.

Written in `einops` (using the named-axis convention from Section 5), both backward GEMMs are completely self-documenting:

```
h1_grad = einsum(h2.grad, w2, "batch out, in out -> batch in")  # 2 B D^2 FLOPs
w2_grad = einsum(h2.grad, h1, "batch out, batch in -> in out")  # 2 B D^2 FLOPs
```

KEY IDEA

Two fundamental consequences follow immediately:

1. **The backward pass takes $2\times$ the FLOPs of the forward pass:** $2BD^2 + 2BD^2 = 4BD^2$ FLOPs per layer, compared to $2BD^2$ FLOPs in forward.
2. **Memory retention:** Computing `w2_grad` requires `h1`, which was produced during the forward pass. Therefore, autograd *cannot* discard h_1 after computing h_2—it must keep h_1 alive in HBM all the way until the backward pass reaches this layer.

COMMON MISCONCEPTION

You might think that the backward pass takes the *same* amount of compute as the forward pass ("one trip forward, one trip back"). That breaks because each layer has *two* inputs—activations h_1 and weights W_2—and backpropagation needs gradients with respect to *both* (one to keep propagating backward, one to update the weights), requiring two full GEMMs instead of one.

The $6BP$ Rule — Dense Models vs. 2026 Architectures

KEY IDEA

Summing across all L layers of our network (recalling $P = LD^2$):

- **Forward pass:** $L \times (2BD^2) = 2B(LD^2) = \mathbf{2BP}$ FLOPs
- **Backward pass:** $L \times (4BD^2) = 4B(LD^2) = \mathbf{4BP}$ FLOPs
- **Total per training step:**

$$\text{Forward } (2BP) + \text{Backward } (4BP) = \mathbf{6BP} \text{ FLOPs per step}$$

WORKED EXAMPLE

Worked Example 2 (Recovering Section 1's $6ND$ Formula): Suppose we train a dense Transformer with $P = N = 70 \times 10^9$ parameters over an entire dataset of $D_{\text{tokens}} = 15 \times 10^{12}$ tokens, split into T training steps of B tokens per step (so $T \cdot B = D_{\text{tokens}}$).

- Each step costs $6BN$ FLOPs.
- Summing over all T steps gives:

$$\text{Total FLOPs} = \sum_{t=1}^{T} 6BN = 6(TB)N = 6ND_{\text{tokens}}$$

This is the exact derivation of the $6ND$ formula we used in Napkin Calculation I!

While $6BP$ is exact for our L-layer MLP (ignoring tiny elementwise ReLU FLOPs) and serves as the standard approximation for dense Transformers at short context lengths, two architectural refinements are essential for modern 2026 models:

1. **Sparse Mixture-of-Experts (MoE)** (e.g., **DeepSeek-V4.1-Flash**): Each token activates only a small subset of expert feedforward layers. Therefore, step compute scales with active parameters as $6BP_{\text{active}}$, while weight and optimizer memory still scale with P_{total}.
2. **Long context and hybrid/sparse attention:** Standard dense self-attention computes pairwise dot products between all S tokens in a sequence, adding an $O(BS^2DL)$ compute term. At short sequence lengths S, $6BP$ dominates; at very long context lengths S, the quadratic S^2 attention term overtakes the linear projections—which directly motivates 2026 hybrid and sparse attention architectures.

We have now accounted for all training FLOPs and discovered why activations must linger in HBM. Yet backpropagation only produces gradients `.grad`; it does not update the weights. To account for the remaining $8\ \text{bytes/parameter}$ of our $12\ \text{B/param}$ memory budget from Section 1—and to assemble the complete HBM equation—we examine **optimizers** and CUDA memory hygiene next.

> **✎ Check Your Understanding**
>
> 1. **Quick Computation:** For a single linear layer $h_2 = h_1W_2$ with batch size $B = 256$ and width $D = 1{,}024$, how many FLOPs does the **backward pass** alone require?
> **Answer:** $4BD^2 = 4 \times 256 \times 1{,}024^2 = 2^{30} \approx 1.07 \times 10^9$ FLOPs (two backward GEMMs of $2BD^2$ each—one for $\nabla_{h_1} L$ and one for $\nabla_{W_2} L$).
>
> 2. **Yes/No with Justification:** During the forward pass of training, can PyTorch immediately overwrite h_1 in HBM as soon as $h_2 = h_1W_2$ finishes computing?
> **Answer:** No. The backward pass needs h_1 to compute the weight gradient $\nabla_{W_2} L = h_1^\top(\nabla_{h_2} L)$, so autograd must retain h_1 in HBM until backpropagation reaches that layer.

Section 10: Optimizers, Memory Hygiene, and the Complete Training Memory Breakdown

Once `loss.backward()` finishes populating `p.grad` for every parameter `p`, an **optimizer** uses those gradients to update the model weights. Simple gradient descent (`SGD`) requires zero extra memory beyond the weights and gradients themselves—so why does modern LLM training pay $8\ \text{bytes/parameter}$ for AdamW?

> **KEY IDEA**
> Raw SGD applies the same learning rate η to every parameter regardless of whether its gradient has been consistently large, consistently tiny, or oscillating violently. In deep neural networks with heterogeneous parameter scales, raw SGD converges slowly or diverges. Adaptive optimizers fix this by tracking per-parameter gradient history—at the direct cost of storing persistent state tensors in GPU HBM.

Adaptive Optimizers: From SGD to AdamW

Comparing the progression of optimizers highlights the exact tradeoff between optimization quality and HBM state per parameter:

Optimizer	Update Idea	FP32 State / Parameter
SGD	Step along raw gradient: $-\eta g$	0 B
Momentum	SGD + exponential moving average of g	4 B
AdaGrad (2011)	SGD scaled by $1/\sqrt{\sum g^2 + \epsilon}$	4 B
RMSProp	AdaGrad with exponential moving average of g^2	4 B
Adam / AdamW	**RMSProp** + **Momentum** (first & second moments)	8 B $(4 + 4)$

Two systems properties apply to all adaptive optimizers:

1. **Optimizer states are stored in FP32 (4 bytes per state tensor):** As we proved in Section 3, storing running moment averages in BF16 causes small updates to swamp and stall over long training runs. Because Adam and AdamW maintain *two* running averages per parameter—the first moment (smoothed mean of g) and the second moment (smoothed variance/uncentered second moment of g^2)—they require $4 + 4 = \mathbf{8}$ bytes per parameter.
2. **Roofline behavior of the optimizer step:** Every optimizer update is purely elementwise (reading parameters, gradients, and moments, updating each entry independently, and writing them back). From Section 8, elementwise operations have very low arithmetic intensity and are therefore **memory-bound**. However, because `optimizer.step()` executes only **once per global step** (whereas forward and backward execute huge GEMMs across every token in the batch), its wall-clock time is small compared to the forward and backward passes when the batch size is large.

Optimizer Implementation and CUDA Memory Hygiene

Implementing **AdaGrad** as a PyTorch `Optimizer` subclass makes every tensor allocation and in-place operation explicit:

```
class AdaGrad(torch.optim.Optimizer):
    def step(self):
        for group in self.param_groups:
            lr = group["lr"]
            for p in group["params"]:
                state = self.state[p]
                grad = p.grad.data
                g2 = state.get("g2", torch.zeros_like(grad))
                g2 += torch.square(grad)
                state["g2"] = g2
                p.data -= lr * grad / torch.sqrt(g2 + 1e-5)
```

Notice what happens on the first call to `step()`: `torch.zeros_like(grad)` allocates a brand-new HBM buffer `g2` the same size as parameter `p`, and persists it in `self.state[p]`. On every step, `g2` accumulates g^2, and `p.data` updates in-place by $-\mathrm{lr} \cdot g / \sqrt{g_2 + 10^{-5}}$.

WORKED EXAMPLE

Worked Example 1 (Tracing Two Steps of AdaGrad on a Single Parameter): Let a parameter start at $p = 1.0$ with learning rate $\eta = 0.1$, $\epsilon = 10^{-5}$, and initial state $g_2 = 0$.

- **Step 1 (gradient $g = 2.0$):**
  - Accumulate squared gradient: $g_2 = 0 + 2.0^2 = 4.0$.
  - Scaled step: $\Delta p = 0.1 \times \dfrac{2.0}{\sqrt{4.0 + 10^{-5}}} \approx 0.1 \times \dfrac{2.0}{2.0} = 0.1$.
  - New parameter: $p = 1.0 - 0.1 = 0.9$.
- **Step 2 (gradient $g = 2.0$ again):**
  - Accumulate squared gradient: $g_2 = 4.0 + 2.0^2 = 8.0$.
  - Scaled step: $\Delta p = 0.1 \times \dfrac{2.0}{\sqrt{8.0 + 10^{-5}}} \approx 0.1 \times \dfrac{2.0}{2.828} \approx 0.0707$.
  - New parameter: $p = 0.9 - 0.0707 = 0.8293$. Because g_2 grows monotonically, parameters that consistently receive large gradients automatically have their effective step size damped.

Now consider **CUDA memory hygiene** after `optimizer.step()` finishes. PyTorch does *not* automatically erase `p.grad` between steps; instead, calling `loss.backward()` **adds** new gradients onto whatever already sits in `p.grad`. Failing to clear gradients silently sums them across steps (a bug if accidental, though we will deliberately exploit this exact behavior for gradient accumulation in Section 11).

When clearing gradients, always pass `set_to_none=True`:

```
optimizer.zero_grad(set_to_none=True)
```

Why does `set_to_none=True` matter compared to `set_to_none=False` (filling every gradient tensor with zeros)?

- Setting `.grad = None` releases the HBM buffers immediately instead of keeping $2P$ bytes of zeroed gradient tensors sitting occupied in HBM (and avoids launching a memory-bound kernel just to write zeros across $2P$ bytes).

COMMON MISCONCEPTION

- You might worry that freeing and re-allocating gigabytes of gradient buffers on every step will trigger slow OS-level `cudaMalloc` and `cudaFree` calls. PyTorch avoids that penalty via its **CUDA caching allocator**, which holds onto freed HBM blocks in an internal pool and recycles them almost instantaneously to the next allocation request without talking to the OS driver.

Complete Training Memory Breakdown

We can now assemble the complete HBM memory budget for training our L-layer network ($P = LD^2$ parameters) in BF16/FP32 AdamW mixed precision. Picture a horizontal segmented bar divided into two main groups—**Static Model State** on the left (fixed solely by model size P) and **Dynamic Activations** on the right (growing with batch size B, depth L, and sequence length S):

1. **Static Model State ($12P$ bytes):**
   - **Parameters (blue block):** $2P$ bytes (BF16)
   - **Gradients (red block):** $2P$ bytes (BF16)
   - **Optimizer State (yellow block):** $8P$ bytes (FP32 AdamW moments; $4P$ for AdaGrad)
2. **Dynamic Activations (green block):**
   - In our L-layer MLP, each of the L layers saves an activation matrix h_ℓ of shape (B, D) in BF16 (2 bytes/element), totaling $2BDL$ bytes.

KEY IDEA

Combining all four blocks produces the complete training HBM equation:

$$\text{Total HBM} = \underbrace{2P}_{\text{params}} + \underbrace{2P}_{\text{grads}} + \underbrace{8P}_{\text{AdamW}} + \underbrace{2BDL}_{\text{activations}}$$

Let's read every term cleanly:

- $2P + 2P + 8P = 12P$ bytes is the exact static footprint we used in Napkin Calculation II in Section 1. Note that if a training framework also keeps a separate **FP32 master copy of parameters** (to avoid losing precision during small weight updates), that adds another $4P$ bytes on top of this baseline (bringing static state to $16P$ bytes).
- $2BDL$ bytes is the dynamic activation memory required by autograd during the forward pass to support backpropagation (derived in Section 9). In full **Transformers**, activations also store attention intermediates (such as query/key/value projections, attention score matrices, and softmax outputs) and therefore scale with **sequence length** S as well.

WORKED EXAMPLE

Worked Example 2 (Full HBM Audit of an L-Layer MLP): Consider a 32-layer MLP ($L = 32$) of width $D = 4{,}096$ trained with batch size $B = 2{,}048$ in BF16/FP32 AdamW (no separate FP32 master copy):

1. **Parameter count:** $P = LD^2 = 32 \times 4{,}096^2 = 32 \times 16{,}777{,}216 = 536{,}870{,}912 \approx 537\text{M}$ parameters.
2. **Static Model State ($12P$):**
   - Parameters: $2P = 1.074 \times 10^9$ bytes $= 1.00$ GiB
   - Gradients: $2P = 1.074 \times 10^9$ bytes $= 1.00$ GiB
   - AdamW state: $8P = 4.295 \times 10^9$ bytes $= 4.00$ GiB
   - Static subtotal: $12P = \mathbf{6.00}$ GiB.
3. **Dynamic Activations ($2BDL$):**

$$2 \times 2{,}048 \times 4{,}096 \times 32 = 536{,}870{,}912 \text{ bytes} = \mathbf{0.50} \text{ GiB}$$

4. **Total HBM:** $6.00 + 0.50 = \mathbf{6.50}$ GiB.

Now observe what happens if we scale this setup to language-model batch sizes (for instance, $B = 65{,}536$ tokens in flight): activation memory jumps $32\times$ from 0.50 GiB to 16.0 GiB, becoming nearly $3\times$ **larger than the entire model, gradients, and optimizer combined!**

This exposes a fundamental collision between our compute goal (large batches B and deep networks L) and our memory ceiling (finite HBM). How do we shrink $2BDL$ without sacrificing either large-batch optimization or model depth? That brings us to the two workhorse memory-saving techniques of LLM training: **gradient accumulation** and **activation checkpointing**.

✎ Check Your Understanding

1. **Fill in the Blank:** Calling `optimizer.zero_grad(set_to_none=True)` releases gradient HBM buffers immediately rather than writing zeros, and PyTorch avoids costly OS `cudaMalloc` overhead on the next step by recycling those freed blocks through its _____.
   Answer: CUDA caching allocator.
2. **What Would Happen If:** You switch a BF16 training run from AdamW to AdaGrad (keeping no separate FP32 master copy of weights). How does the static memory footprint per parameter change?
   Answer: It drops from 12 bytes/parameter to 8 bytes/parameter (2 B params $+$ 2 B grads $+$ 4 B for AdaGrad's single FP32 squared-gradient state instead of AdamW's 8 B for two moments).

Section 11: Breaking the Activation Memory Ceiling: Gradient Accumulation and Activation Checkpointing

Comparing **inference** and **training** exposes why activation memory explodes specifically during training:

- **Inference:** Once layer $\ell + 1$ finishes computing from layer ℓ's output, layer ℓ's activation is never needed again and is immediately discarded. Inference therefore holds only $O(1)$ activation layers in HBM at any instant (plus the KV cache for autoregressive decoding).
- **Training:** Because backpropagation needs every layer's input h_ℓ to compute $\nabla_{W_\ell} L$ (Section 9), forward must keep *every* layer's activation alive in HBM simultaneously—holding $O(L)$ activation layers.

KEY IDEA

This creates a painful tension. From Section 8's roofline analysis, **large batch sizes** B are essential: they improve gradient estimates (optimization stability) *and* push rectangular GEMMs past Blackwell's roofline ridge point I_{hw} to achieve high MFU. Yet activation memory ($2BDL$) grows linearly with both B and L, rapidly triggering out-of-memory (OOM) errors.

KEY IDEA

Because activation memory is proportional to the product $B \cdot L$, we have **two orthogonal levers** to break this ceiling:

- **Gradient accumulation:** shrinks the B factor in activation memory.
- **Activation checkpointing (rematerialization):** shrinks the L factor in activation memory.

Gradient Accumulation: Shrinking the Batch Factor

Suppose we want the exact mathematical weight update of a global batch of size B, but activations for B examples do not fit in HBM. Recall from Section 10 that calling `loss.backward()` **adds** newly computed gradients into the existing `.grad` buffer rather than overwriting it.

DEFINITION

Gradient accumulation splits the global batch B into k sequential **micro-batches**, each of size B/k. Visualize a left-to-right pipeline with $k = 4$ micro-batches:

- Four blue boxes in sequence: `Micro-batch 1 (B/4): Fwd → Bwd (accum)` $\longrightarrow$ `Micro-batch 2 (B/4): Fwd → Bwd (accum)` $\longrightarrow$ `Micro-batch 3 (B/4): Fwd → Bwd (accum)` $\longrightarrow$ `Micro-batch 4 (B/4): Fwd → Bwd (accum)`.
- One final yellow box on the right: `step()` followed by `zero_grad()`.

Here is why this shrinks memory while preserving exact mathematical equivalence:

1. For Micro-batch 1, forward allocates activations only for $B/4$ examples. Immediately afterward, backward consumes those activations, **frees them from HBM**, and stores the micro-batch's gradient in the $2P$-byte `.grad` buffer.
2. For Micro-batches 2, 3, and 4, the peak activation footprint is again only $B/4$ examples—and each backward pass sums its gradients directly into the **same** $2P$-byte `.grad` buffer (zero additional gradient memory!).
3. Because the mean loss over B examples is linear—$\frac{1}{B}\sum_{i=1}^{B}\ell_i = \frac{1}{k}\sum_{m=1}^{k}\left(\frac{1}{B/k}\sum_{i\in\text{micro }m}\ell_i\right)$—dividing each micro-batch loss by k before calling `backward()` produces the **exact same gradient** (up to floating-point associativity) as running one giant batch of size B.

WORKED EXAMPLE

Worked Example 1 (4× Activation Reduction via Gradient Accumulation): Consider our MLP with $B = 64$, $D = 1{,}024$, and $L = 16$ in BF16 (2 bytes/element):

- **Full batch ($B = 64$):**

$$\text{Activation HBM} = 2 \times 64 \times 1{,}024 \times 16 = 2{,}097{,}152 \text{ bytes} = 2 \text{ MiB}$$

- **Gradient accumulation with $k = 4$ (micro-batch $B/4 = 16$):**

$$\text{Peak Activation HBM} = 2 \times 16 \times 1{,}024 \times 16 = 524{,}288 \text{ bytes} = 512 \text{ KiB} \quad (4\times \text{ r}$$

COMMON MISCONCEPTION

The Roofline Pitfall: You might think gradient accumulation is completely free—just set $k = B$ so micro-batch size is 1 and activation memory nearly disappears! That breaks because of Section 8's rectangular GEMM arithmetic intensity formula $I_{\text{op}} = \frac{2}{b(1/K+1/B_{\text{micro}}+1/D)}$. Shrinking the micro-batch size B_{micro} too far drags your GEMMs left of Blackwell's roofline ridge point I_{hw}, turning compute-bound training into memory-bound MatVecs and crashing MFU. You should shrink B_{micro} only as far as HBM forces you—and never below the hardware's compute-bound threshold if avoidable.

Activation Checkpointing (Rematerialization)

DEFINITION

What if even the smallest compute-bound micro-batch still exceeds HBM because the network has many layers L (or long context S)? We attack the L factor using **activation checkpointing** (also called **rematerialization**), whose core philosophy is simple: **trade extra compute for lower HBM usage**.

Instead of saving every intermediate tensor during the forward pass, we save only designated **checkpoint** tensors and discard all intermediate tensors between them. When the backward pass reaches a region whose intermediates were discarded, it runs a small local forward pass starting from the nearest saved checkpoint to **recompute** the missing tensors on demand, uses them to compute gradients, and immediately frees them again.

Visualize a 4-block network where each block computes an intermediate g_ℓ and an output h_ℓ, producing the sequence `x, g1, h1, g2, h2, g3, h3, g4, h4`:

- **Store all (top row):** Every single box—all 9 tensors (`x`, `g1`, `h1`, `g2`, `h2`, `g3`, `h3`, `g4`, `h4`)—is shaded solid and stays pinned in HBM throughout forward.
- **Checkpointed (bottom row):** Only the block boundaries (`x`, `h1`, `h2`, `h3`, `h4`) are shaded yellow as stored checkpoints; all internal intermediates (`g1`, `g2`, `g3`, `g4`) are drawn as empty dashed boxes (discarded during forward). During backward through block 4, a curved red arrow loops forward from checkpoint `h3` into `g4`, labeled **`recompute g4 in backward from checkpoint h3`**.

In PyTorch, wrapping a layer with `torch.utils.checkpoint.checkpoint` implements this automatically:

```
for layer in self.layers:
    x = torch.utils.checkpoint.checkpoint(
        layer, x, use_reentrant=False)
```

How Often to Checkpoint? Deriving the $\sqrt{L}$ Tradeoff

Across a deep network of L layers, how far apart should we space the checkpoints? Let k be the number of layers between consecutive checkpoints (so we store one checkpoint every k layers).

Let's derive the optimal spacing k from first principles:

1. **Persistent checkpoint memory:** Dividing L layers into segments of length k requires storing L/k checkpoint tensors throughout the forward pass.
2. **Temporary recomputation memory during backward:** When backpropagation enters a segment of length k, it must recompute and briefly hold all k activations inside that single segment in order to backpropagate through it (freeing them before moving to the previous segment).

3. **Peak activation memory (in units of layer-sized tensors):**

$$M(k) = \frac{L}{k} + k$$

4. **Minimize $M(k)$ with respect to k:** Differentiate and set to zero:

$$\frac{dM}{dk} = -\frac{L}{k^2} + 1 = 0 \implies k^2 = L \implies k = \sqrt{L}$$

   Substituting $k = \sqrt{L}$ back into $M(k)$ gives a peak activation footprint of $2\sqrt{L} = \mathbf{O}(\sqrt{\mathbf{L}})$ layer tensors!

5. **Recomputation FLOP cost at $k = \sqrt{L}$:** During the backward pass, each of the $\sqrt{L}$ segments recomputes its $\sqrt{L}$ layers **exactly once**. The total number of recomputed layers across the entire backward pass is therefore $\sqrt{L} \times \sqrt{L} = L$ layers—precisely **one extra forward pass** ($2BP$ FLOPs)! Adding this to the normal $6BP$ step raises total step compute to at most $8BP$ **FLOPs** (at most a 33% compute increase).

Compare all three strategies on an $L = 9$ layer network (visualizing three rows of 9 boxes `h1` through `h9`):

- **Every layer ($k = 1$):** All 9 boxes `h1..h9` are solid blue (stored).
- **Every $\sqrt{L}$ ($k = 3$):** Only `h3`, `h6`, and `h9` are highlighted yellow checkpoints; `h1, h2`, `h4, h5`, `h7, h8` are dashed/discarded and recomputed segment by segment in backward.
- **No layers ($k = L$):** Keep only input `x`; to backprop through layer L, recompute layers $1 \dots L-1$; to backprop through layer $L-1$, recompute layers $1 \dots L-2$; and so on—total recomputed layers $= \sum_{\ell=1}^{L-1} \ell = \frac{L(L-1)}{2} = \mathbf{O}(\mathbf{L^2})$, which is computationally prohibitive!

Strategy (L layers)	Activation Memory	Recomputation Cost	Total Step FLOPs
Store every layer	$O(L)$	0	$6BP$
Checkpoint every $\sqrt{L}$	$O(\sqrt{L})$	$O(L)$ **(1 extra fwd)**	$\leq 8BP$
Store no layers	$O(1)$	$O(L^2)$	prohibitive

WORKED EXAMPLE

Worked Example 2 (Quantifying $\sqrt{L}$ Checkpointing on a 64-Layer Model): Consider an $L = 64$ layer network:

- **Without checkpointing:** Must store 64 activation layers; step compute is $6BP$.
- **Checkpointing every $k = \sqrt{64} = 8$ layers:** Stores $64/8 = 8$ checkpoints plus up to 8 temporary activations inside the active backward segment—at most 16 layer tensors total (a $4\times$ **reduction** in peak activation memory, and for $L = 100$ a $5\times$ reduction, growing larger as models deepen). Meanwhile, total compute increases from $6BP$ to at most $8BP$ (a $1.33\times$ factor).

- **Storing no layers:** Recomputes $\frac{64\times 63}{2} = 2{,}016$ layer forward passes—over $31\times$ the work of a normal forward pass!

DEFINITION

On Blackwell GPUs—where compute-to-memory ratios (I_{hw}) are extraordinarily high—**selective activation checkpointing** is even more effective: rather than checkpointing expensive GEMMs, you discard and recompute the *memory-heavy, low-FLOP* operations (activations, layer normalizations, attention intermediates), saving substantial HBM capacity for minimal wall-clock cost.

We now possess every single conceptual and mathematical piece of the lecture. In our final section, we assemble them into one complete Blackwell training loop and map every line of code to its exact hardware cost.

✎ Check Your Understanding

1. **Quick Computation:** For an $L = 100$ layer network using $\sqrt{L}$ activation checkpointing, how many layers apart should checkpoints be placed, and what is the upper bound on total step FLOPs (in terms of B and P)?
   Answer: Checkpoints should be placed every $k = \sqrt{100} = 10$ layers, and total step compute is at most $8BP$ FLOPs ($6BP$ standard forward+backward plus at most $2BP$ for one extra recomputed forward pass).
2. **Yes/No with Justification:** Does using gradient accumulation with $k = 8$ micro-batches change the total amount of HBM required to store the model's gradients (`.grad`)?
   Answer: No. Gradient accumulation shrinks peak *activation* memory by $8\times$, while gradients from all 8 micro-batches sum into the exact same $2P$-byte `.grad` buffer.

Section 12: End-to-End Anatomy of a Blackwell Training Step

Every principle developed across Sections 1 through 11—precision formats, CUDA device placement, mixed precision, roofline regimes, backpropagation's $6BP$ work, optimizer state, memory hygiene, gradient accumulation, and activation checkpointing—converges inside a single eight-line training loop:

```
for step in range(num_steps):
    for micro in range(accum_steps):
        x, y = get_batch()                  # directly on CUDA
        with torch.amp.autocast("cuda", dtype=torch.bfloat16):
            loss = F.mse_loss(model(x).mean(), y) / accum_steps
        loss.backward()                     # accumulates .grad
    optimizer.step()                        # FP32 state update
    optimizer.zero_grad(set_to_none=True)  # free grad HBM
```

Every single line of this loop maps directly to a concrete **HBM allocation**, **HBM transfer**, or **Tensor Core GEMM**:

1. **`x, y = get_batch()` (directly on CUDA):** Allocates or streams the micro-batch of size $B_{\text{micro}} = B/\text{accum_steps}$ straight into GPU HBM, avoiding slow host–device stalls across the CPU–GPU interconnect (Section 4).
2. **`with torch.amp.autocast("cuda", dtype=torch.bfloat16):`:** Configures automatic mixed precision (Section 3) so matrix multiplications run in BF16 on Blackwell Tensor Cores (at $2{,}250\ \text{TFLOP/s}$ dense peak rather than FP32's $75\ \text{TFLOP/s}$, Section 6) while keeping sensitive reductions and nonlinearities in FP32.
3. **`loss = F.mse_loss(model(x).mean(), y) / accum_steps` (Forward pass):**
   - Executes $2B_{\text{micro}}P$ FLOPs of compute-bound Tensor Core GEMMs (Sections 8–9), totaling $2BP$ FLOPs across all `accum_steps` micro-batches.
   - Allocates (or selectively checkpoints, Section 11) the micro-batch's activations in HBM ($2B_{\text{micro}}DL$ bytes in our MLP baseline).
   - Dividing the loss by `accum_steps` ensures the summed gradients across micro-batches match the mean loss over the full global batch B.
4. **`loss.backward()` (Backward pass):**
   - Executes $4B_{\text{micro}}P$ FLOPs of compute-bound Tensor Core GEMMs ($2\times$ forward, Section 9), totaling $4BP$ FLOPs across the full step (or up to $6BP$ if full $\sqrt{L}$ activation checkpointing recomputes one extra forward pass).
   - Reads the saved activations from HBM, immediately frees them back to the CUDA caching allocator once each layer finishes, and accumulates parameter gradients into the $2P$-byte BF16 `.grad` buffers.
5. **`optimizer.step()` (FP32 state update):**
   - Runs once per global step, executing memory-bound elementwise updates across the $2P$-byte BF16 parameters, $2P$-byte BF16 gradients, and $8P$-byte FP32 AdamW moment buffers (Section 10).
6. **`optimizer.zero_grad(set_to_none=True)` (Free grad HBM):**
   - Drops all `.grad` tensor references so the $2P$-byte gradient buffers return immediately to PyTorch's CUDA caching allocator before the next global step begins (Section 10).

Worked End-to-End Resource Audit

WORKED EXAMPLE

To tie every formula from the lecture together in one concrete calculation, audit a single global training step of an $L = 36$ layer MLP of width $D = 4{,}096$ trained on **1 Blackwell B200 GPU** in BF16/FP32 AdamW, with global batch size $B = 8{,}192$, gradient accumulation `accum_steps = 4` (so $B_{\text{micro}} = 2{,}048$), and $\sqrt{L}$ activation checkpointing enabled:

1. **Model size (P):**

$$P = LD^2 = 36 \times 4{,}096^2 = 36 \times 16{,}777{,}216 = 603{,}979{,}776 \approx 604\text{M parameters}$$

2. **Static HBM ($12P$):**

$$12 \times 603{,}979{,}776 \text{ bytes} = 7.248 \times 10^9 \text{ bytes} \approx \mathbf{6.75}\ \text{GiB}$$

(1.125 GiB BF16 params $+$ 1.125 GiB BF16 grads $+$ 4.50 GiB FP32 AdamW state).

3. **Peak Activation HBM:**
   - Without gradient accumulation or checkpointing: $2BDL = 2 \times 8{,}192 \times 4{,}096 \times 36 = 2.416 \times 10^9$ bytes $= \mathbf{2.25}$ GiB.
   - With `accum_steps = 4` ($B_{\text{micro}} = 2{,}048$): drops $4\times$ to 0.5625 GiB (across 36 layers).
   - Adding $\sqrt{L}$ checkpointing ($k = \sqrt{36} = 6$): holds 6 checkpoints plus at most 6 in-segment recomputed layers $= 12$ layer tensors instead of 36 (another $3\times$ drop), shrinking peak activations to:

$$2 \times 2{,}048 \times 4{,}096 \times 12 = 201{,}326{,}592 \text{ bytes} = \mathbf{0.1875}\ \text{GiB}\ (192\ \text{MiB})$$

   - Total HBM is $6.75 + 0.1875 \approx \mathbf{6.94}$ GiB.

4. **Roofline Check on Micro-Batch GEMM ($B_{\text{micro}} = 2{,}048, D = K = 4{,}096, b = 2$):**

$$I_{\text{op}} = \frac{2}{2\left(\frac{1}{4096} + \frac{1}{2048} + \frac{1}{4096}\right)} = \frac{1}{\frac{4}{4096}} = \mathbf{1{,}024}\ \text{FLOP/byte}$$

   At $1{,}024$ FLOP/byte, every micro-batch GEMM sits well to the right of Blackwell's BF16 ridge point—fully compute-bound!

5. **Step Compute and Expected Runtime (at $\text{MFU} = 0.5$ on 1 B200):**
   - Standard forward $+$ backward: $6BP = 6 \times 8{,}192 \times 603{,}979{,}776 \approx 2.97 \times 10^{13}$ FLOPs.
   - Including $\sqrt{L}$ recomputation ($\leq 8BP$): $8 \times 8{,}192 \times 603{,}979{,}776 \approx \mathbf{3.96 \times 10^{13}}$ FLOPs.
   - Effective B200 throughput at $\text{MFU} = 0.5$: $0.5 \times 2.25 \times 10^{15} = 1.125 \times 10^{15}$ FLOP/s.

- Expected step time: $\frac{3.96 \times 10^{13}}{1.125 \times 10^{15}} \approx \mathbf{35.2}\ \text{ms/step}$.

The Complete Resource Accounting Checklist

KEY IDEA

Every system decision in language model training reduces to five verifiable checks:

- **Tensors & Precision:** Track footprint via $\text{numel} \times \text{bytes/elem}$; train in BF16 mixed precision with FP32 optimizer state, and leverage FP8 or microscaled Blackwell NVFP4 for next-generation throughput and memory savings.
- **CUDA & Blackwell:** Allocate and keep tensors in GPU HBM, wrap benchmarks with `torch.cuda.synchronize()` before and after timed blocks, always divide sparse datasheet peaks by 2 for dense GEMMs, and rely on on-chip SMEM/TMEM tiling for reuse.
- **Expressivity:** Write tensor contractions, reductions, and reshapes with `einops` (`einsum`, `reduce`, `rearrange`) so contraction axes, memory-view contiguity, and FLOP counts stay explicit.
- **Compute & Roofline:** Count $2BDK$ per GEMM and $6BP$ per dense training step ($6BP_{\text{active}}$ for sparse MoE, plus $O(BS^2DL)$ for dense long-context attention); diagnose bottlenecks with $\text{MFU}_{\text{roofline}} = \min(1, I_{\text{op}}/I_{\text{hw}})$—large GEMMs are compute-bound, while elementwise ops, optimizer updates, and unbatched autoregressive decode are memory-bound.
- **Fitting Large Runs:** Break the activation memory wall orthogonally via **gradient accumulation** (scaling activation memory by $1/k$, while watching micro-batch arithmetic intensity) and **activation checkpointing** ($O(\sqrt{L})$ memory for $O(L)$ recompute, raising step FLOPs from $6BP$ to at most $8BP$).

✎ Check Your Understanding

1. **What Would Happen If:** In the eight-line Blackwell training loop above, you forgot to divide the micro-batch loss by `accum_steps` (writing `loss = F.mse_loss(model(x).mean(), y)` instead of dividing by `accum_steps`)?
   Answer: Because `loss.backward()` sums gradients across all `accum_steps` micro-batches, the accumulated `.grad` tensor passed to `optimizer.step()` would be scaled up by a factor of `accum_steps` compared to the true mean gradient of a full batch of size B.

2. **Yes/No with Justification:** In the worked audit above, if we kept increasing `accum_steps` until $B_{\text{micro}} = 4$ (still using $D = K = 4{,}096$ in BF16), would the micro-batch GEMMs stay compute-bound on a Blackwell GPU?
   Answer: No. Using $I_{\text{op}} = \frac{2}{2(1/4096+1/4+1/4096)} \approx 4\ \text{FLOP/byte}$, the GEMM's arithmetic intensity would plummet far below Blackwell's ridge point I_{hw}, pushing the kernel deep into the memory-bound regime and collapsing MFU.