Compute-Bound Training

Karl Stratos

Last updated: January, 2026

Contents

1 On-Device Performance

1.1 Compte- vs Memory-Bound

Consider a kernel (i.e., algorithm) processing a stream of inputs. The sustained effective runtime per input is

$$t(N) = \max\left(t_F(N), t_B(N)\right) \tag{1}$$

where N is some measurement of the input size, $t_F(N)$ is the average compute time, and $t_B(N)$ is the average within-device communication time (i.e., moving data between HBM and SRAM).[1] We say the kernel is **compute-bound** if $t_F(N) \geq t_B(N)$, otherwise **memory-bound**. We always want to make the kernel as compute-bound as possible, assuming all operations are meaningful.[2] To analyze when a kernel becomes compute-bound as a function of device properties, let $n_F(N)$ and $n_B(N)$ count FLOPs $(+, -, \times, /)$ and bytes per input. In an idealized setting,

$$t_F(N) = \frac{n_F(N)}{\pi^\star} \qquad\qquad t_B(N) = \frac{n_B(N)}{\beta^\star}$$

where $\pi^\star$ and $\beta^\star$ denote the peak FLOPs/s and bytes/s the device can handle. For example, NVIDIA A100 has $\pi^\star = 3.12 \times 10^{14}$ (BF16) and $\beta^\star = 1.6 \times 10^{12}$. The kernel becomes compute-bound when

$$\underbrace{\frac{n_F(N)}{n_B(N)}}_{I(N)} \geq \underbrace{\frac{\pi^\star}{\beta^\star}}_{I^\star}$$

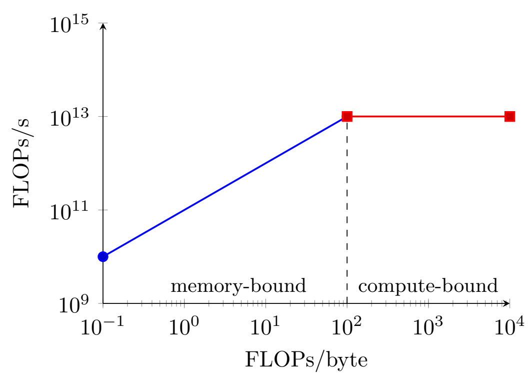
namely the kernel's **arithmetic intensity (AI)** exceeds the critical AI of the device (e.g., $I^\star = 195$ for A100).

1.2 Roofline Model

The compute-bound condition can be stated in $P(N) := n_F(N)/t(N)$ (i.e., the kernel's effective FLOPs/s). We can easily check that

$$P(N) = \min\left(\pi^\star, \beta^\star I(N)\right) \tag{2}$$

which increases linearly in AI with slope $\beta^\star$, until it saturates to $\pi^\star$ at $I(N) \geq I^\star$. This is typically depicted by plotting (2) as a piecewise-linear function of AI in log-log scale, e.g.,



$$\pi^\star = 10^{13} \text{ (10 TFLOPs/s)}$$
$$\beta^\star = 10^{11} \text{ (100 GB/s)}$$
$$I^\star = 100$$

The curve is always an upper bound on empirical measurement, hence the name "roofline model". How close we approach the peak FLOPs/s is measured by **MFU**:

$$\text{MFU}(N) := \frac{P(N)}{\pi^\star} \leq \min\left(1, \frac{I(N)}{I^\star}\right) \tag{3}$$

which is 100% iff the kernel's AI is critical. In real-world training, MFU is typically 20–40% due to various inefficiencies (e.g., small per-device batch size, layers with low AI like attention, interconnect communication).

[1] This assume perfect overlap between computation and memory traffic. In reality, $\max(t_F(N), t_B(N)) \leq t(N) \leq t_F(N) + t_B(N)$.

[2] We can meaninglessly make any kernel compute-bound by, e.g., repeatedly adding and subtracting 1 on SRAM, since it increase the compute time without increasing the communication time.

kernel	mode	FLOP operation	n_F	n_B (BF16)	kernel AI
GEMM (App. A.1)		$A_1 A_2$	$2d_1d_2d_3$	$2(d_1d_2 + d_2d_3 + d_1d_3)$	$(d_1^{-1} + d_2^{-1} + d_3^{-1})^{-1}$
Linear (App. A.2)	forward	XW	$2BFd^2$	$2(Bd + Fd^2 + BFd)$	$(B^{-1} + d_F^{-1})^{-1}$
	backward	—	$4BFd^2$	$4(Bd + Fd^2 + BFd)$	$(B^{-1} + d_F^{-1})^{-1}$
MHA (App. A.3)	forward	$QK^\top V$	$2NT^2d$	$\Theta(NTd)$	$\Theta(T)$
	backward	—	$4NT^2d$	$\Theta(NTd)$	$\Theta(T)$

Table 1: Transformer kernels. GEMM: $A_1 \in \mathbb{R}^{d_1 \times d_2}$, $A_2 \in \mathbb{R}^{d_2 \times d_3}$. Linear: $X \in \mathbb{R}^{B \times d}$, $W \in \mathbb{R}^{d \times Fd}$ for B tokens and assymetry factor $F > 0$ ($d_F := \frac{F}{F+1}d$). MHA: $Q, K, V \in \mathbb{R}^{N \times T \times d}$ for N sequences of T tokens, assuming $d = Hd_H$ for H heads (no GQA) and FlashAttention with causal masking (halving FLOPs).

1.3 Transformer Kernels

Table 1 summarizes the main kernels used in transformer training. The table does not include rematting, which increases the backward n_F, n_B by those of the forward. Some key takeaways are as follows:

- GEMM (with perfect tiling) has AI scaling like $\Theta(\min_i d_i)$ and easily compute-bound (∼600 on A100). But batching does not increase AI (since it increase n_F, n_B equally).
- Linear layer has AI scaling like $\Theta(\min(B, d))$, thus growing linearly in device token load when $B < d$. So we have a longer stretch of this linear scaling for larger models (if training is not already compute-bound).
- FlashAttention improves AI from $\Theta(\min(T, d_H))$ to $\Theta(T)$ by reducing byte traffic; FLOPs remain the same. So MHA is compute-bound for moderate sequence lengths.

1.4 Training FLOPs

Suppose we train a typical transformer with L blocks each containing an attention layer and a gated feedforward layer. We can read off the total training FLOPs from Table 1. For FLOPs caused by parameter-bound GEMM,

$$n_{F,\text{GEMM}} = L(\underbrace{4 \cdot 6Bd^2}_{\text{QKV/output in pre-/post-attention}} + \underbrace{3 \cdot 6BFd^2}_{\text{gated feedforward}}) + \underbrace{6BdV}_{\text{logits}}$$

where V is the vocab size. Attention causes $n_{F,\text{ATTN}} = 6LBTd$ FLOPs (using $B = NT$) without model parameters. Attention dominates GEMM when $n_{F,\text{ATTN}} \geq n_{F,\text{GEMM}}$ or

$$T \geq d(4 + 3F) + \frac{V}{L}$$

While the overhead $\frac{T}{d(4+3F)+V/L}$ can be nontrivial even for short sequences (e.g., 3–4% for Llama-3 70B training with $T = 4096$), GEMM tends to be clearly dominant in most settings, especially with scale (e.g., we would need $T \geq 223\text{K}$ to make Llama-3 405B training attention-bound). This yields a delightfully quick way to approximate the total training FLOPs:

$$n_{F,\text{train}} \approx n_{F,\text{GEMM}} \approx 6n_{\text{params}}n_{\text{tokens}} \tag{4}$$

where n_{params} is the model size and n_{tokens} is the training data size. This is because each parameter in GEMM causes two FLOPs $(+, *)$, once in the forward pass and twice in the backward pass, on each training token.[3] (4) in turn provides a quick way to estimate the training time:

$$t_{\text{train}} \approx \frac{n_{F,\text{train}}}{K \cdot \pi^\star \cdot \text{MFU}} \tag{5}$$

where the denominator computes the total effective FLOPs/s in a cluster of K devices. Since this excludes interconnect communication costs in addition to attention, it tends to be a lower bound, but often surprisingly accurate. For instance, training a 70B model on 15T tokens will cause $\sim 6.4 \times 10^{24}$ FLOPs and take ∼45 days on a pod of 8960 v5p chips at 40% MFU.

[3] With rematting, the factor becomes 8 since the backward pass will generally involve two additional FLOPs. Also, when the embedding layer is not tied to the readout layer, technically we must subtract Vd from n_{params} since it does not participate in GEMM, but for deep models even this error is relatively harmless. Finally, we have ignored elementwise-interacting parameters causing 3 FLOPs each (e.g., layer normalization, biases), but they take up $< 0.1\%$ of the model size, so this counting error makes little difference.

	all-gather	reduce-scatter	all-reduce	all-to-all
initial data condition/chip	$\mathcal{B}/K$	$\mathcal{B}$ [partial]	$\mathcal{B}$ [partial]	$\mathcal{B}/K$
end data condition/chip	$\mathcal{B}$	$\mathcal{B}/K$	$\mathcal{B}$	$\mathcal{B}/K$ [transposed]
total bytes sent/chip (bidir. ring schedule)	$\mathcal{B}$	$\mathcal{B}$	$2\mathcal{B}$	$\mathcal{B}/4$
total comm. time	$\mathcal{B}/(m\beta_I^\star)$	$\mathcal{B}/(m\beta_I^\star)$	$2\mathcal{B}/(m\beta_I^\star)$	$\mathcal{B}/(4m\beta_I^\star)$

Table 2: Collectives (Appendix C). $\mathcal{B}$ is the bytes of the full logical object (what you would have after all-gather on each chip). K is the number of chips (even). $\beta_I^\star$ is the bidirectional interconnect bandwidth of a chip on a physical axis. $m \leq 3$ is the number of physical axes being used.

	sharding	toks/chip	FLOPs	bytes	interconnect AI
DP	batch	B/K_{DP}	BdD/K_{DP}	dD	B/K_{DP}
FSDP	batch, state	B/K_{FSDP}	BdD/K_{FSDP}	dD	B/K_{FSDP}
TP	layout	B	BdD/K_{TP}	Bd	D/K_{TP}
(FSDP+TP)*	batch, state, layout	B/K_{FSDP}	BdD/K	$d\sqrt{BD/K}$	$\sqrt{BD/K}$

Table 3: Distributed training strategies. We write the FLOPs/bytes (fw+bw) and interconnect AI as asymptotic functions of the batch size B, number of chips K_s used for strategy s, and model/feedforward dimensions d, D. (FSDP+TP)* assumes FSDP/TP over independent axes with $K = K_{\mathrm{FSDP}}K_{\mathrm{TP}}$ chips where $K_{\mathrm{FSDP}} \asymp \sqrt{BK/D}$.

2 Interconnect Communication

For crosscheckability, we adopt the setting in Austin *et al.* (2025) and assume a TPU v5p pod (Appendix B), a 3D physical mesh of size $16 \times 20 \times 28$ totalling $K = 8960$ "chips" (i.e., devices). Each chip can compute $\pi^\star = 4.59 \times 10^{14}$ FLOPs/s (BF16) and send $\beta_I^\star = 1.8 \times 10^{11}$ bytes/s to its two neighbors in a single physical axis (thus up to $3\beta_I^\star = 5.4 \times 10^{11}$ bytes/s to its six neighbors across all three physical axes). Table 2 summarizes the fundamental collective operations, where a main takeaway is that all-reduce is $\sim$2$\times$ more expensive and all-to-all is $\sim$4$\times$ cheaper than all-gather/reduce-scatter.

2.1 Compte- vs Interconnect-Bound

In distributed training, the sustained effective runtime per training step is roughly

$$t_{\mathrm{step}} = \max\left(\max\left(t_{\mathrm{fw},F}, t_{\mathrm{fw},I}\right), \max\left(t_{\mathrm{bw},F}, t_{\mathrm{bw},I}\right)\right) \tag{6}$$

where $t_{p,F}, t_{p,I}$ are compute and interconnect communication times for phase $p \in \{\mathrm{fw}, \mathrm{bw}\}$, assuming that the forward and backward passes overlap across batches.[4] This implies training is compute-bound if $t_{p,F} \geq t_{p,I}$ for all phases, otherwise **interconnect-bound**. Since $t_{p,F} = n_{p,F}/\pi^\star$ and $t_{p,I} = n_{p,I}/(m\beta_I^\star)$ where $n_{p,F}, n_{p,I}$ count the FLOPs and bytes in phase p and m is the number of physical axes involved, the compute-bound condition can be written as

$$I_I := \min\left(\frac{n_{\mathrm{fw},F}}{n_{\mathrm{fw},I}}, \frac{n_{\mathrm{bw},F}}{n_{\mathrm{bw},I}}\right) \geq \frac{I_I^\star}{m} \tag{7}$$

where I_I is the "interconnect AI" of the chosen distributed training strategy, and $I_I^\star := \pi^\star/\beta_I^\star$ is the critical interconnect AI of the device (e.g., $I_I^\star = 2550$ for v5p). Table 3 summarizes the AI of various strategies. Note that DP/FSDP are token-bound while TP is width-bound; judiciously combining FSDP and TP can achieve higher AI.

2.2 Data Parallel (DP)

DP distributes B tokens to K_{DP} chips each holding $B_{\mathrm{chip}} = B/K_{\mathrm{DP}}$ tokens. In the backward pass, DP all-reduces the weight gradients (BF16). The all-reduce does not block backpropagation and can be done asynchronously. We

[4] We ignore the within-device communication time here since the bottleneck is often the interconnect rather than HBM bandwidth in a GEMM-dominated regime (unless the per-device token load is extremely small).

have $n_{\mathrm{fw},I} = 0$ and $n_{\mathrm{bw},I} = 4n_{\mathrm{params}}$ ($2\times$ overhead comes from all-reduce, see Table 2), so the compute-bound condition (7) becomes

$$I_I = \min\left(\frac{2n_{\mathrm{params}}B_{\mathrm{chip}}}{0}, \frac{4n_{\mathrm{params}}B_{\mathrm{chip}}}{4n_{\mathrm{params}}}\right) = B_{\mathrm{chip}} \geq \frac{I_I^\star}{m_{\mathrm{DP}}} \tag{8}$$

(e.g., $B_{\mathrm{chip}} \geq 850$ on a v5p pod). *When compute-bound*, DP slashes the step time by K_{DP} times. Unfortunately, pure DP replicates the entire model on every chip, so it is not feasible if the model is too large.

2.2.1 Critical batch size

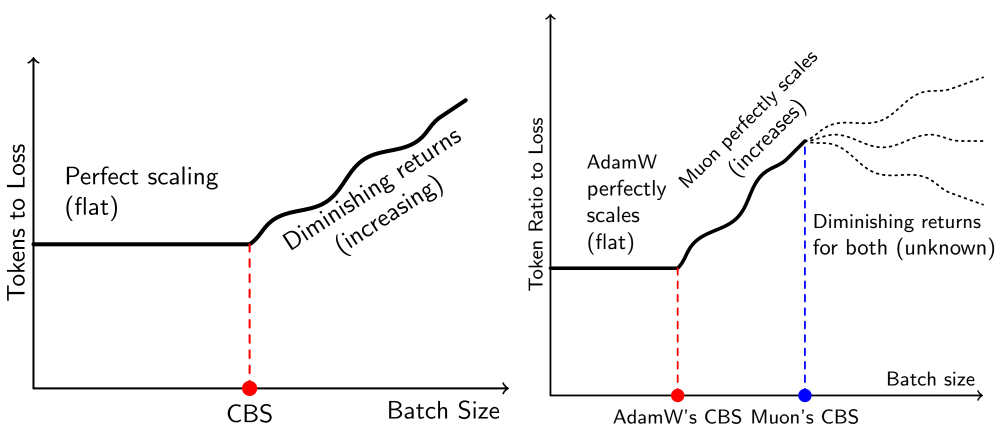
More explicitly, the empirical step time (6) using pure DP is

$$\hat{t}_{\mathrm{step}} = \max\left(\frac{4n_{\mathrm{params}}B}{\hat{\pi}(B_{\mathrm{chip}})K_{\mathrm{DP}}}, \frac{4n_{\mathrm{params}}}{m_{\mathrm{DP}}\hat{\beta}_I}\right) \tag{9}$$

where $\hat{\pi}(B_{\mathrm{chip}})$ is the empirical FLOPs/s per chip and $\hat{\beta}_I$ is the empirical bidirectional interconnect bandwidth. For a sufficiently large B, training is compute-bound with $\hat{\pi}(B_{\mathrm{chip}}) \approx \pi^\star$ roughly constant; the step time decreases in K_{DP} but also increases in B (at fixed K_{DP}). For a sufficiently small B, training is interconnect- or memory-bound. In either case, the step time no longer decreases in K_{DP} *but also no longer increases in* B.[5] This means we can almost indefinitely halve the "token-time" (i.e., time to cover a fixed number n_{train} of training tokens) by

- If already compute-bound, double B and K_{DP} together.
- If memory-/interconnect-bound, double B.

However, the utility of token-time reduction is limited if we end up with a worse model. Then we may have to train on more tokens to match the previous quality. In this case, training may end up taking the same "loss-time" (i.e., time to reach a target loss)—only less data efficient! To preserve the loss-time while reducing the step-time, doubling B must double the per-step loss reduction. This is empirically true up to a certain point $B \leq B^\star$ where $B^\star$ is known as the **critical batch size**. This happens because a bigger batch allows the optimizer to estimate its gradient-based update more accurately, but the returns diminish when B is too large. The relative location of $B^\star$ and the post-$B^\star$ behavior are thus properties of the optimizer. Some curvature-aware optimizers can remain more data efficient at large B than plain first-order methods (e.g., Muon vs Adam, figures from Shah *et al.* (2025)):



Despite the clear empirical identity, the critical batch size requires substantial work to formalize (Appendix E).

2.3 Fully-Sharded Data Parallel (FSDP)

FSDP combines DP with state sharding (i.e., model parameters and optimizer states) to reduce memory load per chip. In the forward pass at each layer, it all-gathers the layer weights (BF16), implying $n_{\mathrm{fw},I} = 2n_{\mathrm{params}}$. The weights block the forward computation but may be prefetched in the previous layer. In the backward pass, it

[5] When interconnect-bound, $\hat{t}_{\mathrm{step}} = \frac{4n_{\mathrm{params}}}{m_{\mathrm{DP}}\hat{\beta}_I} = \Theta(1)$. When memory-bound, $\hat{t}_{\mathrm{step}} = \frac{4n_{\mathrm{params}}B_{\mathrm{chip}}}{\hat{\pi}(B_{\mathrm{chip}})} = \Theta(1)$ since $\hat{\pi}(B_{\mathrm{chip}}) \propto B_{\mathrm{chip}}$ for $B_{\mathrm{chip}} \ll d$ (Appendix A.2).

similarly all-gathers the weights (to compute activation gradients). The weight gradients, however, are now *reduce-scattered* (cheaper than all-reduce) since each chip needs just its own gradient shard. This implies $n_{\mathrm{bw},I} = 4n_{\mathrm{params}}$. Each chip performs the usual FLOPs over $B_{\mathrm{chip}} = B/K_{\mathrm{FSDP}}$ tokens, so the compute-bound condition (7) becomes

$$I_I = \min\left(\frac{2n_{\mathrm{params}}B_{\mathrm{chip}}}{2n_{\mathrm{params}}}, \frac{4n_{\mathrm{params}}B_{\mathrm{chip}}}{4n_{\mathrm{params}}}\right) = B_{\mathrm{chip}} \geq \frac{I_I^\star}{m_{\mathrm{FSDP}}} \tag{10}$$

which is identical to DP! Thus FSDP slashes both per-chip compute and memory at a similar payload as DP, which only slashes compute. In practice, however, FSDP is noticeably slower than DP since it overlaps compute and communcation less effectively and performs more collectives.

- The memory-saving quality of FSDP clearly depends how the model is sharded. If all-gathering the layer weight W implies all-gathering the whole model, there is no memory saving from parameter sharding—only we pay the communication cost (e.g., PyTorch example). But if the model is sharded so that all-gathering W does not involve other weights, the peak model memory during training is the largest layer.
- The optimizer states do not need to be communicated for first-order methods like Adam (since each parameter evolves independently), but this may change for second-order optimizers.

2.4 Tensor Parallel (TP)

TP (aka. model parallel) shards layouts so that each chip only needs to perform partial contraction. In transformers, TP over K_{TP} chips can be viewed as sharding the dominant feedforward layer receiving input activations $X \in \mathbb{R}^{B\times d}$ as[6]

$$Y = \underbrace{X}_{B\times d}\underbrace{W_{\mathrm{up}}}_{d\times D}\underbrace{W_{\mathrm{down}}}_{D\times d} = \sum_{i=1}^{K_{\mathrm{TP}}} \underbrace{X}_{B\times d}\underbrace{W_{\mathrm{up},i}}_{d\times \frac{D}{K_{\mathrm{TP}}}}\underbrace{W_{\mathrm{down},i}}_{\frac{D}{K_{\mathrm{TP}}}\times d} \tag{11}$$

This slashes the per-chip FLOPs by a factor of K_{TP}, but $Y \in \mathbb{R}^{B\times d}$ needs to be all-reduced. Alternatively, we may decompose the all-reduce into all-gathering/reduce-scattering $X_i, Y_i \in \mathbb{R}^{B\times d/K_{\mathrm{TP}}}$, which has the benefit of keeping activations sharded between layers (assuming all layers have matching sharded layouts) and skipping a half of the collectives when the next layer only needs a shard (Appendix D). In either case, the peak per-chip memory is $\Theta(Bd)$ and the compute-bound condition (7) is

$$I_I = \min\left(\frac{4BdD/K_{\mathrm{TP}}}{4Bd}, \frac{8BdD/K_{\mathrm{TP}}}{4Bd}\right) = \frac{D}{K_{\mathrm{TP}}} \geq \frac{I_I^\star}{m_{\mathrm{TP}}} \tag{12}$$

We see that pure TP is severely width-bound, e.g., it would take $D > 54$k to be compute-bound on a $4\times4\times4$ v5p pod (7.6m on the whole pod!). In practice, K_{TP} is kept modest, e.g., 4 chips would make $D \geq 10.2$k sufficient.[7]

2.5 FSDP+TP

We may use FSDP and TP in conjunction over m_{FSDP} and m_{TP} independent physical axes each using K_{FSDP} and K_{TP} chips (thus $m = m_{\mathrm{FSDP}} + m_{\mathrm{TP}} \leq 3$ axes and $K = K_{\mathrm{FSDP}}K_{\mathrm{TP}}$ chips total). Then a chip corresponds to some logical coordinate (i, j) in the FSDP/TP space computing the i-th data shard $Y_i \in \mathbb{R}^{B/K_{\mathrm{FSDP}}\times d}$ by[8]

$$Y_i = \sum_{j=1}^{K_{\mathrm{TP}}} \underbrace{X_i}_{\frac{B}{K_{\mathrm{FSDP}}}\times d}\underbrace{W_{\mathrm{up},j}}_{d\times\frac{D}{K_{\mathrm{TP}}}}\underbrace{W_{\mathrm{down},j}}_{\frac{D}{K_{\mathrm{TP}}}\times d} \tag{13}$$

This means TP only communicates $n_{\mathrm{fw},I}^{\mathrm{TP}} = \frac{4Bd}{K_{\mathrm{FSDP}}}$ bytes (sharded activations) and FSDP only all-gathers $n_{\mathrm{fw},I}^{\mathrm{FSDP}} = \frac{4dD}{K_{\mathrm{TP}}}$ bytes (sharded weights). Since TP and FSDP are independent, the communication time is bottlenecked by

[6]The gated feedforward layer $Y = (XW_{\mathrm{gate}}) \odot (XW_{\mathrm{up}})W_{\mathrm{down}}$ works with the same sharding layouts, only increasing FLOPs without changing the communication cost (i.e., it only makes compute-bound easier).

[7]TP group is placed contiguously along one physical axis, so it effectively uses $m_{\mathrm{TP}} = 1$ axis of bandwidth.

[8]Again, we focus on the dominant feedforward layer WLOG for the purpose of computing the interconnect AI of transformer training. Furthermore, we focus on the forward pass which is the bottleneck (i.e., the backward pass is more compute-bound, see (12)).

whichever is slower. It is easy to verify that

$$K^\star_{\text{FSDP}} = \mathop{\arg\min}_{\substack{K_{\text{FSDP}}>0: \\ K_{\text{TP}}=K/K_{\text{FSDP}}}} \max\left(\underbrace{\frac{n^{\text{FSDP}}_{\text{fw},I}}{\beta^\star_I m_{\text{FSDP}}}}_{t_{I,\text{FSDP}}}, \underbrace{\frac{n^{\text{TP}}_{\text{fw},I}}{\beta^\star_I m_{\text{TP}}}}_{t_{I,\text{TP}}}\right) = \sqrt{\frac{BKm_{\text{FSDP}}}{Dm_{\text{TP}}}} \tag{14}$$

minimizes the communication time at $t_I = t_{I,\text{FSDP}} = t_{I,\text{TP}} = \frac{4d}{\beta^\star_I}\sqrt{\frac{BD}{Km_{\text{FSDP}}m_{\text{TP}}}}$. The per-chip compute of (13) remains at $t_F = \frac{4BdD}{K\pi^\star}$. The compute-bound condition $t_F \geq t^\star_I$ can be written as

$$\frac{B}{K} \geq \frac{(I^\star_I)^2}{Dm_{\text{FSDP}}m_{\text{TP}}} \tag{15}$$

The idea is that (15) is easier to satisfy than (10) when D is sufficiently large. For instance with a batch size of 4.19m, pure DP/FSDP on the whole v5p pod (i.e., using 8960 chips in 3 axes) is never compute-bound since there are not enough tokens per chip ($468 < 850$). However, suppose $D = 28672$ and consider 2240-way FSDP and 4-way TP (which is put together from $K^\star_{\text{FSDP}} \approx 1619$ in (14) using $m_{\text{FSDP}} = 2$ and $m_{\text{TP}} = 1$).[9] This satisfies the condition (15) ($468 > 113$).

References

Austin, J., Douglas, S., Frostig, R., Levskaya, A., Chen, C., Vikram, S., Lebron, F., Choy, P., Ramasesh, V., Webson, A., and Pope, R. (2025). How to scale your model. Retrieved from https://jax-ml.github.io/scaling-book/.

McCandlish, S., Kaplan, J., Amodei, D., and Team, O. D. (2018). An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*.

Shah, I., Polloreno, A. M., Stratos, K., Monk, P., Chaluvaraju, A., Hojel, A., Ma, A., Thomas, A., Tanwer, A., Shah, D. J., *et al.* (2025). Practical efficiency of muon for pretraining. *arXiv preprint arXiv:2505.02222*.

[9] The split is logically valid but breaks the independent axes assumption: an FSDP group of size 2240 must span all three physical dimensions, so TP and FSDP will contend on at least one physical dimension. We maintain the independent-axes model as a simplifying throughput proxy. In practice, JAX can build topology-aware meshes that attempt an efficient mapping from logical axes to the physical topology.

A Kernels

A.1 GEMM

General matrix multiplication (GEMM) computes:

$$\underbrace{Z}_{d\times D} = \underbrace{X}_{d\times K}\underbrace{Y}_{K\times D}$$

A perfect kernel will load the entire X, Y to SRAM once:

PerfectGEMM$(X, Y, \text{output}{=}Z)$

- $X \leftarrow$ **READ**(X)
- $Y \leftarrow$ **READ**(Y)
- **WRITE**(Z, XY)

$$\begin{aligned} n_F(d,K,D) &= 2dDK \\ n_B(d,K,D) &= 2(dK + KD + dD) \\ I(d,K,D) &= \frac{1}{\frac{1}{D} + \frac{1}{d} + \frac{1}{K}} \end{aligned}$$

(the factor of 2 in n_B comes from the BF16 precision). The AI is optimal and strictly increasing in all input dimensions (e.g., $I(d) = \frac{1}{3}d$ with $d = D = K$). But SRAM is severely limited (e.g., 164–192KB on A100), so the perfect kernel only works on trivially small matrices.

A.1.1 Naive GEMM

A naive kernel will "give up" on optimizing the SRAM usage and compute each output cell $Z_{i,j} \in \mathbb{R}$ independently (e.g., separate CUDA thread), loading each input element every time it is needed:

NaiveGEMM$(X, Y, \text{output}{=}Z)$

- For $i = 1 \dots d$ and $j = 1 \dots D$ in parallel (independent threads):
  - $z \leftarrow 0$
  - For $k = 1 \dots K$:
    - $x_k \leftarrow$ **READ**$(X_{i,k})$
    - $y_k \leftarrow$ **READ**$(Y_{k,j})$
    - $z \leftarrow z + x_k \times y_k$
  - **WRITE**$(Z_{i,j}, z)$

$$\begin{aligned} n_F(d,K,D) &= 2dDK \\ n_B(d,K,D) &= 2(2dDK + dD) \\ I(d,K,D) &= \frac{1}{2 + \frac{1}{K}} \in [0.33, 0.5) \end{aligned}$$

Even though computation of all output cells is fully parallelized, AI is dismal (< 0.5) because we now read the input many times: each $X_{i,k}$ is read D times and each $Y_{k,j}$ is read d times.

A.1.2 Tiled GEMM

We can avoid re-reading while conserving memory by low-rank decomposition of $Z = XY$. Specifically, we partition X, Y into X_k, Y_k along the contracting dimension and compute $Z = \sum_k X_k Y_k$. In this way, X_k, Y_k can be loaded exactly once to SRAM to calculate $X_k Y_k \in \mathbb{R}^{d\times D}$ then evicted. However, this still requires fitting X_k, Y_k on SRAM. We may further partition Z into "tiles". For instance, if $d = 2m$, $D = 3n$, and $K = 4l$, we may chop up the output matrix into 6 tiles ($m \times n$) and compute each independently (e.g., seperate thread block):

$$\begin{bmatrix} X_{1,1} & X_{1,2} & X_{1,3} & X_{1,4} \\ X_{2,1} & X_{2,2} & X_{2,3} & X_{2,4} \end{bmatrix} \begin{bmatrix} Y_{1,1} & Y_{1,2} & Y_{1,3} \\ Y_{2,1} & Y_{2,2} & Y_{2,3} \\ Y_{3,1} & Y_{3,2} & Y_{3,3} \\ Y_{4,1} & Y_{4,2} & Y_{4,3} \end{bmatrix} = \begin{bmatrix} Z_{1,1} & Z_{1,2} & Z_{1,3} \\ Z_{2,1} & Z_{2,2} & Z_{2,3} \end{bmatrix} \qquad \underbrace{Z_{1,2}}_{m\times n} = \sum_{t=1}^{4} \underbrace{X_{1,t}}_{m\times l}\underbrace{Y_{t,2}}_{l\times n}$$

An input tile now may need to be loaded multiple times (e.g., $X_{1,1}$ is required for 3 output tiles), but even so many times fewer than the naive kernel. A device further optimizes communication overhead by storing recently used input tiles in an intermediate cache (L2) that is much larger than SRAM (e.g., 40MB on A100), where the bandwidth of L2↔SRAM is many times faster than HBM↔L2:

TiledGEMM$(X, Y, \text{output}{=}Z)$

- For each output tile $Z_{\text{tile}} \in \mathbb{R}^{m \times n}$ in parallel (thread blocks):
  - $T_Z \leftarrow 0_{m \times n}$
  - For each $X_{\text{tile}} \in \mathbb{R}^{m \times l}$ and $Y_{\text{tile}} \in \mathbb{R}^{l \times n}$ serially (cooperative threads):
    - $T_X \leftarrow \textbf{READ}_{\text{L2}}(X_{\text{tile}})$ # if cache miss, load from HBM to L2
    - $T_Y \leftarrow \textbf{READ}_{\text{L2}}(Y_{\text{tile}})$ # if cache miss, load from HBM to L2
    - `sync()` # ensure tiles are available in SRAM to all threads
    - $T_Z \leftarrow T_Z + T_X T_Y$
    - `sync()` # ensure SRAM is not overwritten by any thread yet
  - $\textbf{WRITE}(Z_{\text{tile}}, T_Z)$

$$n_F(d, K, D) = 2dDK$$
$$n_B(d, K, D) \geq 2(dK + KD + dD)$$
$$I(d, K, D) \leq \frac{1}{\frac{1}{D} + \frac{1}{d} + \frac{1}{K}}$$

While it may not quite reach the optimal AI (e.g., due to L2 overhead and cache misses), the kernel is now practical, making GEMM one of the most scalable operations on hardware accelerators. In this note, we assume GEMM is perfect and reads the input matrices exactly once for simplicity.

A.1.3 Batching

More generally, the kernel can batch M independent matrix multiplications into a single GEMM:

$$\underbrace{Z}_{M \times d \times D} = \underbrace{X}_{M \times d \times K} \underbrace{Y}_{M \times K \times D} \tag{16}$$

The kernel remains the same except that it operates on "cubes" $Z_{\text{cube}} \in \mathbb{R}^{p \times m \times n}$ instead of tiles (i.e., it schedules $(M/p) \times (d/m) \times (D/n)$ such cubes to be processed by independent thread blocks). Since this increases *both* n_F, n_B by a factor of M, AI remains constant in the batch size. The effective runtime of batched GEMM thus satisfies

$$t(M, d, K, D) = M \times t(1, d, K, D)$$

assuming d, K, D are large enough to saturate the device by themselves. For instance, one batched GEMM of 100 giant matrices takes roughly the same time as serially computing the 100 giant matrix multiplications.[10]

Takeaway. Perfect GEMM has arithmetic intensity $\Theta(\min(d, D, K))$, thus easily compute-bound for nontrivial matrices (e.g., $\min(d, D, K) > 3I^\star \approx 600$ on A100 with BF16).[11] But batching does not increase arithmetic intensity.

A.2 Linear Layer

The linear layer has the parameter $W \in \mathbb{R}^{d \times Fd}$ (omitting the bias) for some assymetry factor $F > 0$. The input is a batch $X \in \mathbb{R}^{B \times d}$ of B vectors (e.g., token embeddings or activations).

Forward.

$$\underbrace{Y}_{B \times Fd} = \underbrace{X}_{B \times d} \underbrace{W}_{d \times Fd}$$

$$n_F(B, d, F) = 2BFd^2$$
$$n_B(B, d, F) = 2(Bd + Fd^2 + BFd)$$
$$I(B, d, F) = \frac{1}{\frac{1}{B} + \frac{1}{d_F}} = \frac{1}{2} H(B, d_F)$$

where $d_F = \frac{F}{F+1} d \in (0, d)$ and $H(B, d_F)$ is the harmonic mean. Thus $\frac{1}{2} \min(B, d_F) \leq I(B, d, F) \leq \min(B, d_F)$ scales jointly in B and d_F. However, it also continuously scales in one even if the other is fixed. For instance, holding d_F fixed, we have $I(B, d, F) = \left(\frac{B}{B + d_F}\right) d_F \to d_F$ as $B \to \infty$.

[10]This omits the kernel-launch overhead time $t_{\text{launch}} \approx 5\mu$s. With it, the unbatched overhead is $\frac{M(t_{\text{launch}} + t(1, d, K, D))}{t_{\text{launch}} + t(M, d, K, D)}$, which is $\approx$1 assuming $t_{\text{launch}} \ll t$ (but M if $t_{\text{launch}} \gg t$).

[11]To see this, note that if $m = \min(d, D, K)$, we have $1/m \leq 1/d + 1/D + 1/K \leq 3/m$ and thus $m/3 \leq (1/d + 1/D + 1/K)^{-1} \leq m$.

Backward. Given the gradient $z_Y \in \mathbb{R}^{B \times Fd}$ of the loss with respect to Y, the backward pass computes

$$\underbrace{z_X}_{B\times d} = \underbrace{z_Y}_{B\times Fd}\underbrace{W^\top}_{Fd\times d} \qquad \underbrace{z_W}_{d\times Fd} = \underbrace{X^\top}_{d\times B}\underbrace{z_Y}_{B\times Fd} \qquad \begin{aligned} n_F(B,d,F) &= 4BFd^2 \\ n_B(B,d,F) &= 4(Bd + Fd^2 + BFd) \\ I(B,d,F) &= \text{same as forward} \end{aligned}$$

Note that z_Y is read twice. One may consider a fused backward kernel that reads z_Y once to double the AI, but we will avoid complications that do not change the asymptotic behavior.

Backward with remat. Often to save memory, we "rematerialize" the input X in the backward pass instead of saving it from the forward pass (i.e., activation checkpointing). Specifically, we compute

$$X = X_{\text{prev}} W_{\text{prev}}$$

which is used as input to the backward matmul $z_W = X^\top z_Y$. Assuming the previous matrices have the same sizes, the backward pass with remat increases the factor from 4 to 6 in n_F, n_B (but the AI remains the same).

A.3 Attention Layer

The **multi-head attention (MHA)** layer has no learnable parameters. The input is a tuple of query, key, and value tensors $Q, K, V \in \mathbb{R}^{NH \times T \times d_H}$ where N is the number of sequences, H is the number of heads, T is the sequence length, and d_H is the head dimension. We assume the typical case $d = d_H H$ (i.e., the model dimension is partitioned across heads).

Forward. The forward pass computes

$$\underbrace{O}_{NH\times T\times d_H} = \underbrace{\left(\frac{QK^\top}{\sqrt{d_H}}\right)\text{.softmax(dim=–1)}}_{NH\times T\times T}\underbrace{V}_{NH\times T\times d_H}$$

It involves two batched GEMMs (batch size NH): one between $T \times d_H$ and $d_H \times T$, and the other between $T \times T$ and $T \times d_H$. Each requires $2NT^2d$ FLOPs, thus $4NT^2d$ FLOPs in total. Naively constructing the probability tensor $\Pi \in \mathbb{R}^{NH\times T\times T}$ requires moving $\Theta(NHT^2 + NTd)$ bytes, resulting in an AI of

$$I_{\text{naive}}(N,H,T,d_H) = \Theta\left(\frac{Td_H}{T+d_H}\right) = \Theta\left(\min(T,d_H)\right)$$

Instead, it is now standard to use tiled attention that loads the $NH \times T \times d$ input matrices only once (e.g., FlashAttention), moving only $\Theta(NTd)$ bytes. This results in an AI of

$$I_{\text{tiled}}(N,H,T,d_H) = \Theta(T)$$

Backward. Given the gradient $z_O \in \mathbb{R}^{NH\times T\times d_H}$ of the loss with respect to O, the backward pass computes

$$z_\Pi = z_O V^\top \qquad\qquad z_K = \frac{z_A^\top Q}{\sqrt{d_H}}$$

$$z_V = \Pi^\top z_O \qquad\qquad z_Q = \frac{z_A K}{\sqrt{d_H}}$$

$$z_A = \Pi \odot z_\Pi - (\Pi \odot z_\Pi)\text{.sum(dim=–1)} \odot \Pi$$

where $A = \frac{QK^\top}{\sqrt{d_H}}$ and $\Pi = A$.softmax(dim=–1). It involves four batched GEMMs each requiring $2NT^2d$ FLOPs, thus $8NT^2d$ FLOPs in total. As in the forward pass, AI is $\Theta(T)$ with a tiled kernel.

Backward with remat. We may save memory by rematerializing O in the backward pass. This requires re-running the entire forward pass to obtain $(Q,K,V) \mapsto O$, increasing the FLOPs to $12NT^2d$.

Summary. With tiled attention, forward/backward performs $\Theta(NT^2d)$ FLOPs and moves $\Theta(NTd)$ bytes, with an AI of $\Theta(T)$. Furthermore, it considers only $\sum_{t=1}^{T} t = T(T+1)/2$ of the attention scores under causal masking, so the number of FLOPs is $\approx \frac{1}{2}$ of the total count above, resulting in $\approx 2NT^2d$ for forward and $\approx 4NT^2d$ for backward (or $\approx 6NT^2d$ with remat).[12]

B TPU v5p Pod

A TPU v5p pod is a 3D physical mesh of size $16 \times 20 \times 28$ tallying to $K = 8960$ chips total. Each chip is connected to 6 neighbors with wraparounds, so that a single physical axis yields a "ring". Let $\beta_I^\star$ denote the peak *bidirectional* interconnect bandwidth of a chip on the ring (i.e., using its two links on that axis, both directions).[13] A chip can then send up to $3\beta_I^\star$ bytes/s across all three physical axes, assuming all links are simultaneously saturated.

B.1 Slices and logical mesh

A v5p pod comes in reconfigurable $4 \times 4 \times 4$ cubes. This allows us to request any multiple of a cube as a mini 3D torus containing $K_{\text{slice}} \leq K$ chips (with wraparounds) called a **slice**. Even though there are three physical axes $P_x \times P_y \times P_z = K_{\text{slice}}$, we can use a higher-dimensional **logical mesh** with named axes $M_1 \times \cdots \times M_n = K_{\text{slice}}$. This works simply by bijectively mapping logical coordinates onto the physical 3D torus, but has a practical implication that later axes tend to be more fragmented due to larger strides (under standard row-major ordering) and thus with lower effective bandwidth. So we often use the first axis for a collective with the heaviest bandwidth consumption (e.g., data parallel).

C Collectives

A **collective** is an operation that changes the initial data condition of a set of devices/chips to some end condition (agnostic to the type of hardware accelerator, e.g., GPUs vs TPUs). Let $\mathcal{B}$ denote the total bytes to communicate between K (even) chips in a ring. We will assume a bidirectional ring schedule to calculate the total bytes moved per chip (n_I) and time (t_I) for each collective operation.

C.1 All-Gather

- **Initial condition**: Each chip holds a single shard of data (of size $\mathcal{B}/K$).
- **End condition**: Each chip holds all data (of size $\mathcal{B}$).
- **Communication load/time**: For $K/2$ steps, each chip forwards one shard-sized chunk to each neighbor, which is to be retained.[14] Thus the total outgoing volume per chip is $n_I = (K/2)(2\mathcal{B}/K) = \mathcal{B}$. The total communication time is $t_I = \mathcal{B}/\beta_I^\star$ using the fact that all bytes are sent bidirectionally.

C.2 Reduce-Scatter

- **Initial condition**: Each chip holds all data, but partially reduced (of size $\mathcal{B}$).
- **End condition**: Each chip holds a single shard of completely reduced data (of size $\mathcal{B}/K$).
- **Communication load/time**: Same as all-gather (instead of retaining, reduce and keep forwarding).

C.3 All-Reduce

- **Initial condition**: Each chip holds all data, but partially reduced (of size $\mathcal{B}$).
- **End condition**: Each chip holds all data, completely reduced (of size $\mathcal{B}$).
- **Communication load/time**: Reduce-scatter + all-gather, thus $n_I = 2\mathcal{B}$ and $t_I = 2\mathcal{B}/\beta_I^\star$.

[12] Causal masking does not reduce the byte traffic, so AI also falls by $\approx \frac{1}{2}$ (but remains $\Theta(T)$).

[13] The inter-chip interconnects are referred to as **ICI** and distinguished from data-center networking (**DCN**) which connects a set of ICI-connected TPUs (much more slowly).

[14] The last step only has one chunk left to send for even K, but we ignore this effect for simplicity (e.g., assume dummy chunk).

C.4 All-to-All

- **Initial condition**: Each chip holds a single shard of data (of size $\mathcal{B}/K$), further logically partitioned into K chunks ("sharded shards" of size $\mathcal{B}/K^2$).
- **End condition**: Each chip holds one chunk from every chip, i.e., the sharding axes are transposed (of size $\mathcal{B}/K$).
- **Communication load/time (idealized)**: A crude accounting suggests a total outgoing volume of

$$n_I = \Big(\underbrace{K/2 + (K/2-1) + \cdots + 1}_{\text{num chunks sent to left}} + \underbrace{(K/2-1) + (K/2-2) + \cdots + 1}_{\text{num chunks sent to right}}\Big)\frac{\mathcal{B}}{K^2} = \frac{\mathcal{B}}{4}$$

per chip, yielding $t_I \approx \mathcal{B}/(4\beta_I^\star)$. Thus all-to-all may achieve a smaller effective communication volume than all-gather/reduce-scatter by streaming smaller chunks. In practice the effective all-to-all bandwidth is often lower due to routing/contention.

D Tensor Parallel Algorithms

Reminder: we wish to compute the feedforward layer

$$Y = X \underbrace{W_{\text{up}}}_{d\times D} \underbrace{W_{\text{down}}}_{D\times d} = \sum_{i=1}^{K_{\text{TP}}} X \underbrace{W_{\text{up},i}}_{d\times \frac{D}{K_{\text{TP}}}} \underbrace{W_{\text{down},i}}_{\frac{D}{K_{\text{TP}}}\times d}$$

D.1 All-Reduce Version

- Forward: each chip holds the input $X \in \mathbb{R}^{B\times d}$

$$\begin{aligned}
H_i &\leftarrow XW_{\text{up},i} \in \mathbb{R}^{B\times D/K_{\text{TP}}} && \text{\# } 2BdD/K_{\text{TP}} \text{ FLOPs}\\
\widehat{Y}_i &\leftarrow H_iW_{\text{down},i} \in \mathbb{R}^{B\times d} && \text{\# } 2BdD/K_{\text{TP}} \text{ FLOPs}\\
Y &\leftarrow \textbf{AllReduce}(\widehat{Y}_i) \in \mathbb{R}^{B\times d} && \text{\# } 4Bd \text{ bytes (blocking)}
\end{aligned}$$

- Backward: each chip holds the gradient $z_Y \in \mathbb{R}^{B\times d}$

$$\begin{aligned}
z_{H_i} &\leftarrow z_Y W_{\text{down},i}^\top \in \mathbb{R}^{B\times D/K_{\text{TP}}} && \text{\# } 2BdD/K_{\text{TP}} \text{ FLOPs}\\
z_{W_{\text{down},i}} &\leftarrow H_i^\top z_Y \in \mathbb{R}^{D/K_{\text{TP}}\times d} && \text{\# } 2BdD/K_{\text{TP}} \text{ FLOPs}\\
\hat{z}_{X_i} &\leftarrow z_{H_i} W_{\text{up},i}^\top \in \mathbb{R}^{B\times d} && \text{\# } 2BdD/K_{\text{TP}} \text{ FLOPs}\\
z_X &\leftarrow \textbf{AllReduce}(\hat{z}_{X_i}) \in \mathbb{R}^{B\times d} && \text{\# } 4Bd \text{ bytes (non-blocking)}\\
z_{W_{\text{up},i}} &\leftarrow X^\top z_{H_i} \in \mathbb{R}^{d\times D/K_{\text{TP}}} && \text{\# } 2BdD/K_{\text{TP}} \text{ FLOPs}
\end{aligned}$$

D.2 All-Gather/Reduce-Scatter Version

- Forward: each chip holds an input shard $X_i \in \mathbb{R}^{B\times d/K_{\text{TP}}}$

$$\begin{aligned}
X &\leftarrow \textbf{AllGather}(X_i) \in \mathbb{R}^{B\times d} && \text{\# } 2Bd \text{ bytes (blocking)}\\
H_i &\leftarrow XW_{\text{up},i} \in \mathbb{R}^{B\times D/K_{\text{TP}}} && \text{\# } 2BdD/K_{\text{TP}} \text{ FLOPs}\\
\widehat{Y}_i &\leftarrow H_iW_{\text{down},i} \in \mathbb{R}^{B\times d} && \text{\# } 2BdD/K_{\text{TP}} \text{ FLOPs}\\
Y_i &\leftarrow \textbf{ReduceScatter}(\widehat{Y}_i) \in \mathbb{R}^{B\times d/K_{\text{TP}}} && \text{\# } 2Bd \text{ bytes (blocking)}
\end{aligned}$$

- Backward: each chip holds a gradient shard $z_{Y_i} \in \mathbb{R}^{B \times d/K_{\mathrm{TP}}}$

$$\begin{aligned}
z_Y &\leftarrow \mathbf{AllGather}(z_{Y_i}) \in \mathbb{R}^{B \times d} && \text{\# } 2Bd \text{ bytes (blocking)} \\
z_{H_i} &\leftarrow z_Y W_{\mathrm{down},i}^\top \in \mathbb{R}^{B \times D/K_{\mathrm{TP}}} && \text{\# } 2BdD/K_{\mathrm{TP}} \text{ FLOPs} \\
z_{W_{\mathrm{down},i}} &\leftarrow H_i^\top z_Y \in \mathbb{R}^{D/K_{\mathrm{TP}} \times d} && \text{\# } 2BdD/K_{\mathrm{TP}} \text{ FLOPs} \\
\hat{z}_{X_i} &\leftarrow z_{H_i} W_{\mathrm{up},i}^\top \in \mathbb{R}^{B \times d} && \text{\# } 2BdD/K_{\mathrm{TP}} \text{ FLOPs} \\
z_{X_i} &\leftarrow \mathbf{ReduceScatter}(\hat{z}_{X_i}) \in \mathbb{R}^{B \times d/K_{\mathrm{TP}}} && \text{\# } 2Bd \text{ bytes (non-blocking)} \\
z_{W_{\mathrm{up},i}} &\leftarrow X^\top z_{H_i} \in \mathbb{R}^{d \times D/K_{\mathrm{TP}}} && \text{\# } 2BdD/K_{\mathrm{TP}} \text{ FLOPs}
\end{aligned}$$

Note that X may need to be all-gathered again if not kept from the forward pass.

E Critical Batch Size Theory

We review the seminal work of McCandlish *et al.* (2018) who derive a few ways to think about critical batch sizes.

E.1 Gradient Noise Scale

Consider SGD on $J(\theta) = \mathbf{E}_x[J_x(\theta)]$ by the gradient estimator $\hat{g}(B) := (1/B)\sum_{i=1}^B \nabla J_{x_i}(\theta) \in \mathbb{R}^d$ from iid samples, then ask:

How should we choose the batch size B to maximize the loss reduction in this step?

An impractical answer is "as large as possible" to compute $g := \nabla J(\theta) = \lim_{B\to\infty} \hat{g}(B)$.[15] A more practical answer is "just large enough". Since $\mathbf{E}[\hat{g}(B)] = g$ and $\mathrm{Cov}(\hat{g}(B)) = \frac{\Sigma}{B}$ where $\Sigma := \mathrm{Cov}_x(\nabla J_x(\theta))$, the required B depends on Σ relative to $||g||$ (i.e., local signal-to-noise ratio). Define

$$\Delta^\star(B) := \max_{\eta \geq 0} \; J(\theta) - \mathbf{E}[J^{(2)}(\theta - \eta \hat{g}(B))] \tag{17}$$

as the loss reduction with batch size B after quadratic-model line search (the expectation is wrt. the B iid samples). $J^{(2)}$ is the second-order approximation to coax the closed-form solution

$$\Delta^\star(B) = \frac{\Delta J_{\max}}{1 + B_{\mathrm{noise}}/B} \qquad \Delta J_{\max} := \frac{||g||^4}{2g^\top H g} \quad \text{("infinite-batch loss reduction")} \tag{18}$$

$$B_{\mathrm{noise}} := \frac{\mathrm{tr}\,(H\Sigma)}{g^\top H g} \quad \text{("gradient noise scale")}$$

where $H := \nabla^2 J(\theta)$ is the local Hessian (assuming $g^\top H g > 0$).[16] Thus (17) grows linearly for $B < B_{\mathrm{noise}}$ but stays $\Theta(1)$ for $B > B_{\mathrm{noise}}$ (saturating to $\Delta J_{\max}$ as $B \to \infty$). Two insights:

1. The noise scale is small early in training (high signal) and large late in training (low signal).
2. The improvement is reduced by the multiplicative factor $1 + \frac{B_{\mathrm{noise}}}{B}$.

E.1.1 Learning rate

The optimal LR accompanying the gradient scale (i.e., the argmax of (17)) is

$$\eta^\star(B) = \frac{\eta_{\max}}{1 + B_{\mathrm{noise}}/B} \qquad \eta_{\max} := \frac{||g||^2}{g^\top H g} \quad \text{("infinite-batch optimal LR")} \tag{19}$$

which likewise grows linearly for sub-scale batches but stays $\Theta(1)$ for post-scale. However, $\eta^\star(B)$ is a local optimum of the quadratic model and should be read as a scaling law, not a plug-in recipe. We mention a couple heuristics.

[15] This follows from the usual steepest descent argument that an update v with fixed $||v||$ which maximizes the first-order decrease must be $v \propto g$. But note that we must optimize the learning rate for each batch size to make this hold empirically.

[16] Use $\mathbf{E}[J^{(2)}(\theta - \eta\hat{g}(B))] = J(\theta) - \eta\,||g||^2 + \frac{\eta^2}{2}\mathrm{tr}\left(H\mathbf{E}[\hat{g}(B)\hat{g}(B)^\top]\right)$ with $\mathbf{E}[\hat{g}(B)\hat{g}(B)^\top] = \mathrm{Cov}(\hat{g}(B)) - gg^\top = \frac{\Sigma}{B} - gg^\top$. The name "gradient noise scale" is inspired by the classical signal-to-noise framework where the optimal regression *error* has the form $\sigma^2/(1+\sigma^2/\nu^2)$ that goes up in noise $\nu^2 > 0$. Here we have an optimal *improvement* of form $\Delta J_{\max}/(1 + B_{\mathrm{noise}}/B)$ that goes up in batch size B, so the direction is flipped.

The first is $\eta_{\text{cov}}(B) = \sqrt{rB}$ where $r = \eta_0^2/B_0$ is the base ratio. This choice preserves the covariance of the SGD update (or any optimizer whose empirical covariance shrinks by $1/B$) regardless of the batch size since $\text{Cov}(\theta - \eta\hat{g}(B)) = (\eta^2/B)\Sigma = r\Sigma$. The second is simply $\eta_{\text{lin}}(B) \propto B$ (e.g., double the LR when doubling B), which aims to achieve a similar pace of learning regardless of the batch size. This choice is justified only under certain assumptions, e.g., $B \ll B_{\text{noise}}$ or the function is locally linear where the current gradient is the same as the adjacent gradients. In practice, increasing the batch size generally necessiates some degree of LR search (although LR should generally go up with B).

E.2 Pareto Frontier

(18) does not tell us how to choose B for the whole training run, only one particular step. At a macro level, B causes a tradeoff between $E(B)$ and $S(B)$—number of examples vs steps needed to reach a fixed target loss. More specifically, $E(B) = BS(B)$ and the tradeoff comes from the fact that $S(B)$ should decrease like $S_{\text{min}} + O(1/B)$ where S_{min} denotes the irreducible number of necessary steps. Assuming the per-step slowdown factor $1 + \frac{B_{\text{noise}}}{B}$ in (18) composes over training, and B_{noise} is constant throughout training, we have the precise relationships

$$E_{\text{min}} = B_{\text{noise}} S_{\text{min}} \tag{20}$$

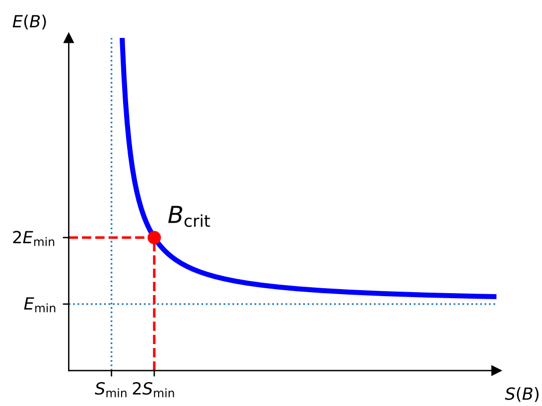
$$S(B) = S_{\text{min}} \left(1 + \frac{B_{\text{noise}}}{B}\right) = S_{\text{min}} + \frac{E_{\text{min}}}{B} \tag{21}$$

$$E(B) = S_{\text{min}} \left(B + B_{\text{noise}}\right) = BS_{\text{min}} + E_{\text{min}} \tag{22}$$

where E_{min} denotes the irreducible number of necessary examples. Thus a balanced macro-level batch size is

$$B_{\text{crit}} := \frac{E_{\text{min}}}{S_{\text{min}}} \tag{23}$$

It is located at the "knee" of the curve:



"Knee"
$S(B_{\text{crit}}) = 2S_{\text{min}}$
$E(B_{\text{crit}}) = 2E_{\text{min}}$

Estimation. Naively, we will estimate (23) by directly gauging the asymptotes $E_{\text{min}}, S_{\text{min}}$ with extreme batch sizes. Instead, McCandlish *et al.* (2018) note that eliminating B in (21) and (22) yields the following law:

$$\frac{S(B)}{S_{\text{min}}} - 1 = \left(\frac{E(B)}{E_{\text{min}}} - 1\right)^{-1} \tag{24}$$

which allows us to empirically generate the pairs $(S(B), E(B))$ for different batch sizes B and fit (24) (i.e., nonlinear regression) to get estimates of $E_{\text{min}}, S_{\text{min}}$.

E.3 Dynamic Batch Size

A limitation of B_{crit} (23) is that it assumes B_{noise} is constant. Let $B_{\text{noise}}(s)$ denote the noise scale at a unit of training run s and $B(s)$ our choice of batch size. The number of steps and examples to reach the target loss is

$$S(B(\cdot)) := \int_s 1 + \frac{B_{\text{noise}}(s)}{B(s)}\, ds \qquad\qquad E(B(\cdot)) := \int_s B(s) + B_{\text{noise}}(s)\, ds \tag{25}$$

$$S_{\text{min}} := \int_s 1\, ds \qquad\qquad E_{\text{min}} := \int_s B_{\text{noise}}(s)\, ds$$

(the continuous relaxation is not necessary but simplifies the math). It is clear that (25) subsumes (20–22) as a special case. Now we pose the following optimization problem:

$$B^\star(\cdot) = \operatorname*{arg\,min}_{B(\cdot):\; S(B(\cdot))=S_0} E(B(\cdot))$$

where S_0 is our "time budget". We can easily verify by Lagrangian relaxation

$$B^\star(s) = \sqrt{\lambda B_{\text{noise}}(s)} \tag{26}$$

for some $\lambda \geq 0$. This suggests a dynamic batch size schedule where we refresh B by $\sqrt{\lambda B_{\text{noise}}}$ once in a while where we estimate B_{noise} and choose a knob λ to select a point on the data-time Pareto frontier. Also note that (26) is consistent with the "knee" optimality under constant B_{noise} since we have $B^\star = \sqrt{\lambda B_{\text{noise}}} = B_{\text{noise}}$ for $\lambda = B_{\text{noise}}$.

Estimation. McCandlish *et al.* (2018) propose a no-overhead estimator for B_{noise} by approximating

$$B_{\text{noise}} \approx \frac{\operatorname{tr}(\Sigma)}{||g||^2} =: B_{\text{simple}}$$

(i.e., isotropic curvature), developing unbiased estimators for $\operatorname{tr}(\Sigma)$ and $||g||^2$ separately, then finally taking their ratio.[17] The estimators are

$$\widehat{\operatorname{tr}(\Sigma)} = \frac{||\hat{g}(B_2)||^2 - ||\hat{g}(B_1)||^2}{\frac{1}{B_2} - \frac{1}{B_1}} \qquad\qquad \widehat{||g||^2} = \frac{B_1\,||\hat{g}(B_1)||^2 - B_2\,||\hat{g}(B_2)||^2}{B_1 - B_2}$$

where $B_1 = KB_2 > B_2$ correspond to global (K chips) and per-chip batch sizes so that we just need to compute the gradient norms before and after averaging. To derive these estimators, use the key identity $\mathbf{E}[||\hat{g}(B)||^2] = ||g||^2 + \frac{\operatorname{tr}(\Sigma)}{B}$ and consider the difference between B_1 and B_2 to isolate $\operatorname{tr}(\Sigma)$ and $||g||^2$; their unbiasedness follows from the linearity of expectation.

[17] The resulting estimator is noisy and generally biased, so they recommend using a moving average to reduce variance.