# Making Transformers Memory-Bound Again

Karl Stratos

See the appendix for a background on the roofline model (Appendix A) and analyses of transformer kernels (Appendix B–E).

## 1 Token-Time

To reduce training "token-time" (i.e., time to process $n_{\text{tokens}}$ tokens), we can increase the MFU of linear layers by increasing the batch size $B$—almost linearly until $B \approx d$ (Appendix C)—until we hit some sustained empirical FLOP/s roof. At this point, training is compute-bound and there is no benefit of considering a larger batch size. A single update with a batch size $2B$ will take as long as two updates with batch size $B$:

$$t(2B) \approx 2t(B) \qquad\qquad B \gg d$$

However, the story changes radically if the batch is partitioned into $K$ shards (sequence-level). The *per-device* batch size $B_{\text{device}} = B/K$ is cut down by a factor of $K$, thus the device becomes memory-bound again, e.g.,

$$t(2B_{\text{device}}) \approx t(B_{\text{device}}) \qquad\qquad B_{\text{device}} \ll d$$

Hence we can keep reducing token-time by increasing $K$ and $B$ jointly.

## 2 Loss-Time

Although the above strategy reduces token-time, it may not reduce "loss-time" (i.e., time to reach a target loss, this is what really matters). The bottleneck becomes the optimizer. For reasonable batch sizes, using the same amount of data results in a similar final loss:

$$\text{Loss}\,(B, n_{\text{tokens}}) \approx \text{Loss}(B', n_{\text{tokens}}) \qquad\qquad \forall B_{\text{min}} \leq B, B' \leq B^\star \tag{1}$$

$B^\star$ is known as the "critical batch size". However, for large (post-critical) batch sizes, the optimizer's data efficiency diminishes:

$$\text{Loss}\,(B, n_{\text{tokens}}) > \text{Loss}(B', n_{\text{tokens}}) \qquad\qquad B > \max(B', B^\star)$$

Thus loss-time may not necessarily decrease even if we keep increasing the batch size and the number of devices jointly (thus increasing MFU). This is why an optimizer that preserves data efficiency for large batch sizes is appealing for large-scale training.

# A Background

A **kernel** is a function implemented for hardware accelerators like GPUs and TPUs. Given an input with some size $N$, it performs $n_F(N)$ FLOPs (i.e., $+, -, \times, /$) and moves $n_B(N)$ bytes (between HBM and SRAM). We assume a "sustained" setting in which the kernel processes a stream of inputs. Then the time to perform $n_F(N)$ FLOPs and move $n_B(N)$ bytes on average can be viewed as the sustained compute/communication time per input, which we will denote by $t_F(N)$ and $t_B(N)$ (in seconds). The sustained effective runtime per input is then

$$t(N) = \max\left(t_F(N), t_B(N)\right) \tag{2}$$

assuming perfect overlap between computation and memory traffic.[1] (2) is thus bottlenecked by whichever of compute/communication that takes longer.

**Definition A.1.** The kernel is **compute-bound** if $t_F(N) \geq t_B(N)$ and **memory-bound** if $t_F(N) < t_B(N)$.

Intuitively, a memory-bound kernel wastes time by waiting around for data, making the effective runtime longer than "necessary". Therefore we are motivated to make $t_F(N)$ as large as possible relative to $t_B(N)$ (i.e., spend more time in compute than communication). This is clearly meaningful only if all operations are meaningful.[2]

## A.1 Roofline Model

Let $\pi^\star$ and $\beta^\star$ denote the peak FLOP/s and bytes/s (aka. HBM bandwidth) the device can handle in an ideal setting. For example, NVIDIA A100 has $\pi^\star = 3.12 \times 10^{14}$ (312 TFLOP/s) and $\beta^\star = 1.6 \times 10^{12}$ (1.6 TB/s). We want to make the kernel's **effective FLOP/s**

$$P(N) := \frac{n_F(N)}{t(N)} = \frac{n_F(N)}{\max\left(t_F(N), t_B(N)\right)} \leq \pi^\star \tag{3}$$

as close to $\pi^\star$ as possible. Note that even if the kernel is optimally implemented so that it truly performs $\pi^\star$ FLOP/s and transfers $\beta^\star$ bytes/s, (3) is not necessarily tight. It depends on the compute vs communcation workloads, more specifically

$$P(N) = \frac{n_F(N)}{\max\left(t_F(N), t_B(N)\right)} = \frac{n_F(N)}{\max\left(\frac{n_F(N)}{\pi^\star}, \frac{n_B(N)}{\beta^\star}\right)} = \min\left(\pi^\star, \beta^\star \frac{n_F(N)}{n_B(N)}\right) = \min\left(\pi^\star, \beta^\star I(N)\right) \tag{4}$$

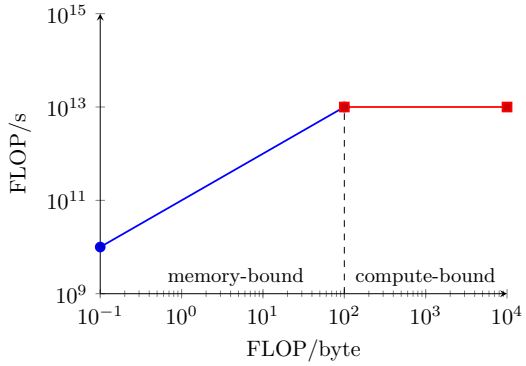
where $I(N) := \frac{n_F(N)}{n_B(N)}$ (FLOP/byte) is known as the **arithmetic intensity (AI)** of the kernel. Thus the effective FLOP/s linearly increases in AI (with slope $\beta^\star$), until it reaches $\pi^\star$ and remains there. The transition point is characterized by

$$\underbrace{\beta^\star I(N) = \pi^\star}_{\text{effective FLOP/s matches the peak}} \quad \Leftrightarrow \quad \underbrace{I(N) = \frac{\pi^\star}{\beta^\star} =: I^\star}_{\text{AI becomes "critical"}} \quad \Leftrightarrow \quad \underbrace{t_F(N) = t_B(N)}_{\text{kernel becomes compute-bound}}$$

where $I^\star$ (critical AI) represents the FLOP-byte ratio when the device is optimally functioning (e.g., 195 FLOP/byte for A100). This is typically depicted by plotting (4) as a piecewise-linear function of AI in log-log scale, e.g.,

[1] Without overlap, $t(N) = t_F(N) + t_B(N)$. Real kernels often overlap to some extent using techniques like double-buffered/tiled schedule, though falling short of perfect overlap for various reasons.

[2] For instance, we can trivially make a kernel compute-bound by repeatedly adding and subtracting 1 on SRAM, since it increase the compute time without increasing the communication time, but the benefit of such a compute-bound kernel is meaningless (i.e., fully exploiting the computational faculty of the device to do meaningless work).




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

The *empirical* FLOP/s $\pi(N) = \frac{n_F(N)}{t_F(N)} \leq \pi^\star$ and bandwidth $\beta(N) = \frac{n_B(N)}{t_B(N)} \leq \beta^\star$ depend on many factors.[3] In all cases, it holds that $P(N) \leq \min(\pi^\star, \beta^\star I(N))$ where $\pi^\star, \beta^\star$ upper bound the effective FLOP/s (hence the name **roofline model**). How close we approach the peak FLOP/s is measured by **MFU** (model FLOPs utilization):

$$\text{MFU}(N) := \frac{P(N)}{\pi^\star} \leq \min\left(1, \frac{I(N)}{I^\star}\right) \tag{5}$$

which is 100% iff the kernel's AI is critical. In real-world large-scale distributed training, MFU is typically not higher than 40% and as low as 20% due to various inefficiencies in small per-device batch sizes, layers with low AI (e.g., attention), and inter-device communication (e.g., all-reduce to average gradients in data parallelism).

**Takeaway.** A kernel is compute-bound on a device (i.e., it performs peak FLOP/s) iff its arithemtic intensity $I(N)$ passes a device-specific threshold for some sufficiently large input size $N$. Otherwise, the kernel is memory-bound.

# B 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**$(Z, XY)$

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

(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.

## B.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:

[3] A primary factor is the input size $N$, which leaves tensor-core pipelines and HBM links idle if too small. Other factors include non-FLOP operations like masking (reducing $\pi(N)$) and cache misses (reducing $\beta(N)$),

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

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

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

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.

## B.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.

### B.2.1 Batching

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

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

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.[4]

**Takeaway.** Perfect GEMM has arithmetic intensity $O(d)$ where $d$ is any matrix dimension, thus easily compute-bound for nontrivial matrices (e.g., $d > 3I^\star \approx 600$ on A100 with BF16). But batching does not increase arithmetic intensity.

# C 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).

**Forward.**

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

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

where $d_F = \frac{F}{F+1} d \in (0, d)$ and $H(B, d_F)$ is the harmonic mean of $B, d_F > 0$. Thus

$$\frac{1}{2} \min(B, d_F) \le I(B, d, F) < \min(B, d_F)$$

scaling *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 $B \to \infty$.[5]

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

$$\underbrace{z_X}_{B \times d} = \underbrace{z_Y}_{B \times Fd} \underbrace{W^\top}_{Fd \times d} \qquad \underbrace{z_W}_{d \times Fd} = \underbrace{X^\top}_{d \times B} \underbrace{z_Y}_{B \times Fd}$$

$$\begin{aligned} n_F(B, d, F) &= 4BFd^2 \\ n_B(B, d, F) &= 4(Bd + Fd^2 + BFd) \\ I(B, d, F) &= \text{same as forward} \end{aligned}$$

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).

# D 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).

[4] 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 $t_{\text{launch}} \gg t$).

[5] Do not confuse this with batching independent matmuls (Section B.2.1). Here, $X \in \mathbb{R}^{B \times d}$ grows in $B$, thus we are actually scaling a matrix dimension.

**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 $O(NHT^2 + NTd)$ bytes, resulting in an AI of

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

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 $O(NTd)$ bytes. This results in an AI of

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

**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

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

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

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

where $A = \frac{QK^\top}{\sqrt{d_H}}$ and $\Pi = A.\text{softmax(dim=–1)}$. It involves four batched GEMMs each requiring $2NT^2d$ FLOPs, thus $8NT^2d$ FLOPs in total. As in the forward pass, AI is $O(T)$ 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 $O(NT^2d)$ FLOPs and moves $O(NTd)$ bytes, with an AI of $O(T)$. 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).[6]

# E Transformer FLOPs

Given a batch $X \in \mathbb{R}^{N\times T\times d}$ of $B = NT$ token embeddings, the forward pass computes:

- For layer $l = 1 \dots L$:
  - Attention
    - 4 linear layers ($d \to d$) on $B$ tokens to compute the QKV and output
    - 1 attention layer on $NH$ sequences
  - Gated feedforward
    - 3 linear layers (two $d \to Fd$, one $Fd \to d$) on $B$ tokens
- 1 linear layer ($d \to V$) on $B$ tokens to compute the logits

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

The total number of FLOPs in the forward and backward pass on one batch is thus the sum of (with $F = 4$)

$$\begin{aligned} n_{F,G}(N,T,d) &= 6(16Ld^2 + dV)B && \text{(GEMM)} \\ n_{F,A}(N,T,d) &= (6LTd)B && \text{(attention)} \end{aligned} \tag{6}$$

The attention overhead over GEMM is[7]

$$\frac{n_{F,A}(N,T,d)}{n_{F,G}(N,T,d)} = \frac{T}{16d + \frac{V}{L}} \approx \frac{T}{17d}$$

which can be significant (e.g., a quarter if $T = 4d$). In practice, attention does cause a large overhead in training time.

## E.1 Shortcut

Even so, it is common to approximate the total FLOPs by parameter GEMM FLOPs only, since they are still the majority. Given $n_{\text{params}}$ parameters and $n_{\text{tokens}}$ training tokens, this is[8]

$$n_F \approx (6 + [[\text{remat}]]2) \times (n_{\text{params}} - [[\text{unemb}]]n_{\text{emb}}) \times n_{\text{tokens}} \tag{7}$$

Note that (6) is consistent with (7). If the cluster performs $P$ FLOP/s, we can predict that training will take

$$t_{\text{train}} \approx \frac{n_F}{P} \tag{8}$$

seconds. Since this excludes various communication costs (e.g., inter-device in distributed training) in addition to attention, it tends to be a lower bound, but often surprisingly accurate. For instance, training a 500M model on 12.5B tokens performing 1.4 PFLOP/s will take about 10.2 hours (with rematerialization); training a 8.3B model on 6T tokens performing 238 PFLOP/s will take about 20 days.

[7] We assume $V \approx Ld$ to be concrete (e.g., 30 layers for $V = 30d$).

[8] $n_{\text{params}}$ can include elementwise parameters such as layer normalization scales and biases. Technically, these incur only 3 FLOPs per parameter (i.e., each op causes only $*$). But they take up $< 0.1\%$ of the model size, so this counting error makes little difference.