All the Transformer Math You Need to Know

Part 4 of How To Scale Your Model (Part 3: Sharding | Part 5: Training)

Here we'll do a quick review of the Transformer architecture, specifically how to calculate FLOPs, bytes, and other quantities of interest.

AUTHORS

Jacob Austin
Sholto Douglas
Roy Frostig
Anselm Levskaya
Charlie Chen
Sharad Vikram
Federico Lebron
Peter Choy
Vinay Ramasesh
Albert Webson
Reiner Pope*

AFFILIATION

Google DeepMind

PUBLISHED

Feb. 4, 2025

Counting Dots

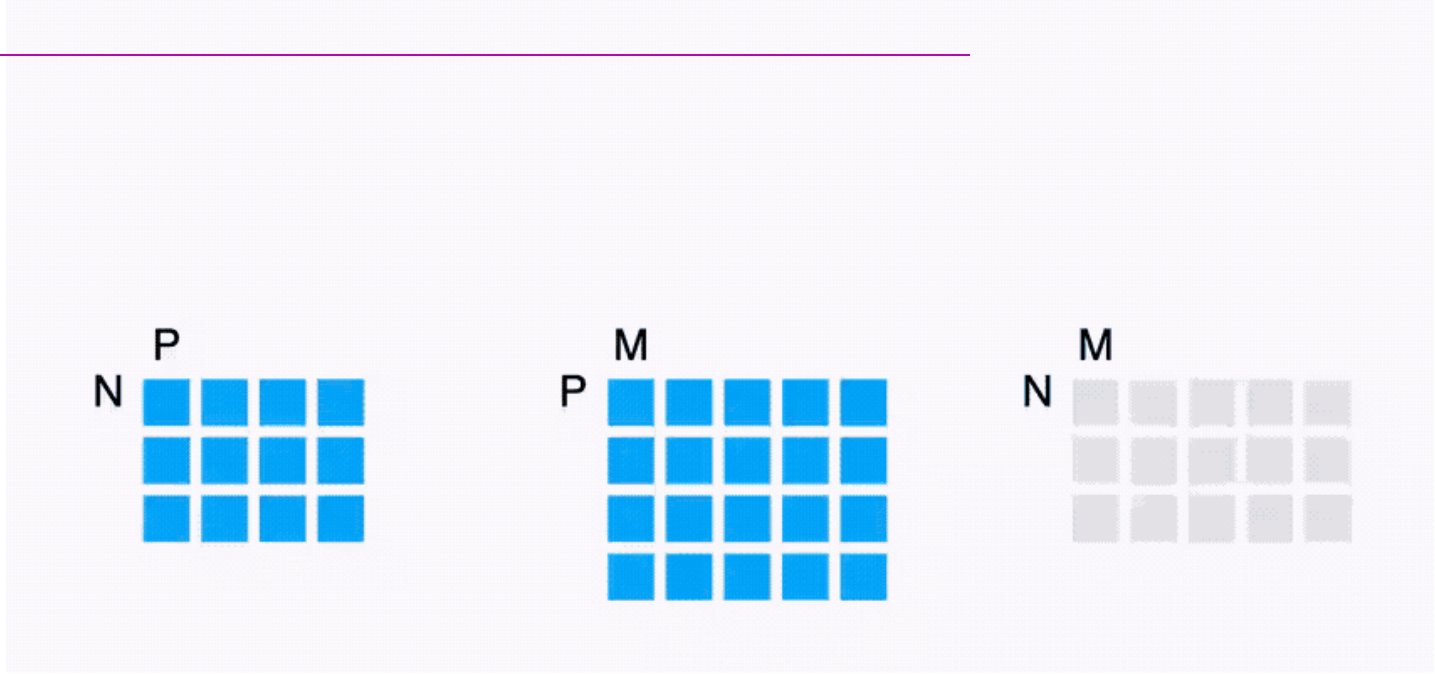
Let's start with vectors x, y and matrices A, B of the following shapes:

array	shape
x	$[\mathrm{P}]$
y	$[\mathrm{P}]$
A	$[\mathrm{N\ P}]$
B	$[\mathrm{P\ M}]$

- A dot product of $x \cdot y$ requires P *adds* and *multiplies*, or $2P$ floating-point operations total.
- A matrix-vector product Ax does N dot-products along the rows of A, for $2NP$ FLOPs.
- A matrix-matrix product AB does a matrix-vector product for each of the M columns of B, for $2NPM$ FLOPs total.
- In general, if we have two higher dimensional arrays C and D, where some dimensions are CONTRACTING and some are BATCHING. (e.g. $C[GHIJKL]$, $D[GHMNKL]$) then the FLOPs cost of this contraction is two times the product of all of the C and D dimensions where the batch and contraction dimensions are only counted once, (e.g. $2GHIJMNKL$). Note that a dimension is only batching if it occurs in both multiplicands. (Note also that the factor of 2 won't apply if there are no contracting dimensions and this is just an elementwise product.)

Operation	FLOPs	Data
$x \cdot y$	$2P$	$2P$
Ax	$2NP$	$NP + P$
AB	$2NPM$	$NP + PM$
$[c_0, \ldots, c_N] \cdot [d_0, \ldots, d_N]$	$2\prod c_i \times \prod_{\substack{d_j \notin BATCH \\ d_j \notin CONTRACT}} d_j$	$\prod c_i + \prod d_j$

Make note of the fact that for a matrix-matrix multiply, the *compute* scales cubically $O(N^3)$ while the data transfer only scales quadratically $O(N^2)$ - this means that as we scale up our matmul size, it becomes *easier* to hit the compute-saturated limit. This is extremely unusual, and explains in large part why we use architectures dominated by matrix multiplication - they're amenable to being scaled!



Forward and reverse FLOPs

During training, we don't particularly care about the result of a given matrix multiply; we really care about its derivative. That means we do significantly more FLOPs during backpropagation.

If we imagine **B** is just one matrix in a larger network and **A** are our input activations with **C = A B**, the derivative of the loss **L** with respect to **B** is given by the chain rule:

$$\frac{\partial L}{\partial B} = \frac{\partial L}{\partial C}\frac{\partial C}{\partial B} = A^T\left(\frac{\partial L}{\partial C}\right)$$

which is an outer product and requires $2NPM$ FLOPs to compute (since it contracts over the N dimension). Likewise, the derivative of the loss with respect to **A** is

$$\frac{\partial L}{\partial A} = \frac{\partial L}{\partial C}\frac{\partial C}{\partial A} = \left(\frac{\partial L}{\partial C}\right)B^T$$

is again $2NPM$ FLOPs since **dL/dC** is a (co-)vector of size $[N, M]$. While this quantity isn't the derivative w.r.t. a parameter, it's used to compute derivatives for previous layers of the network (e.g. just as dL/dC is used to compute dL/dB above).

Adding these up, we see that **during training, we have a total of 6NPM FLOPs**, compared to 2NPM during inference: 2NPM in the forward pass, 4NPM in the backward pass. Since PM is the number of parameters in the matrix, this is the simplest form of the famous $6 * \text{num parameters} * \text{num tokens}$ approximation of Transformer FLOPs during training: each token requires $6 * \text{num parameters}$ FLOPs. We'll show a more correct derivation below.

Transformer Accounting

Transformers are the future. Well, they're the present at least. Maybe a few years ago, they were one of many architectures. But today, it's worth knowing pretty much every detail of the architecture. We won't reintroduce the architecture but this blog and the original Transformer paper may be helpful references.

Here's a basic diagram of the Transformer decoder architecture:

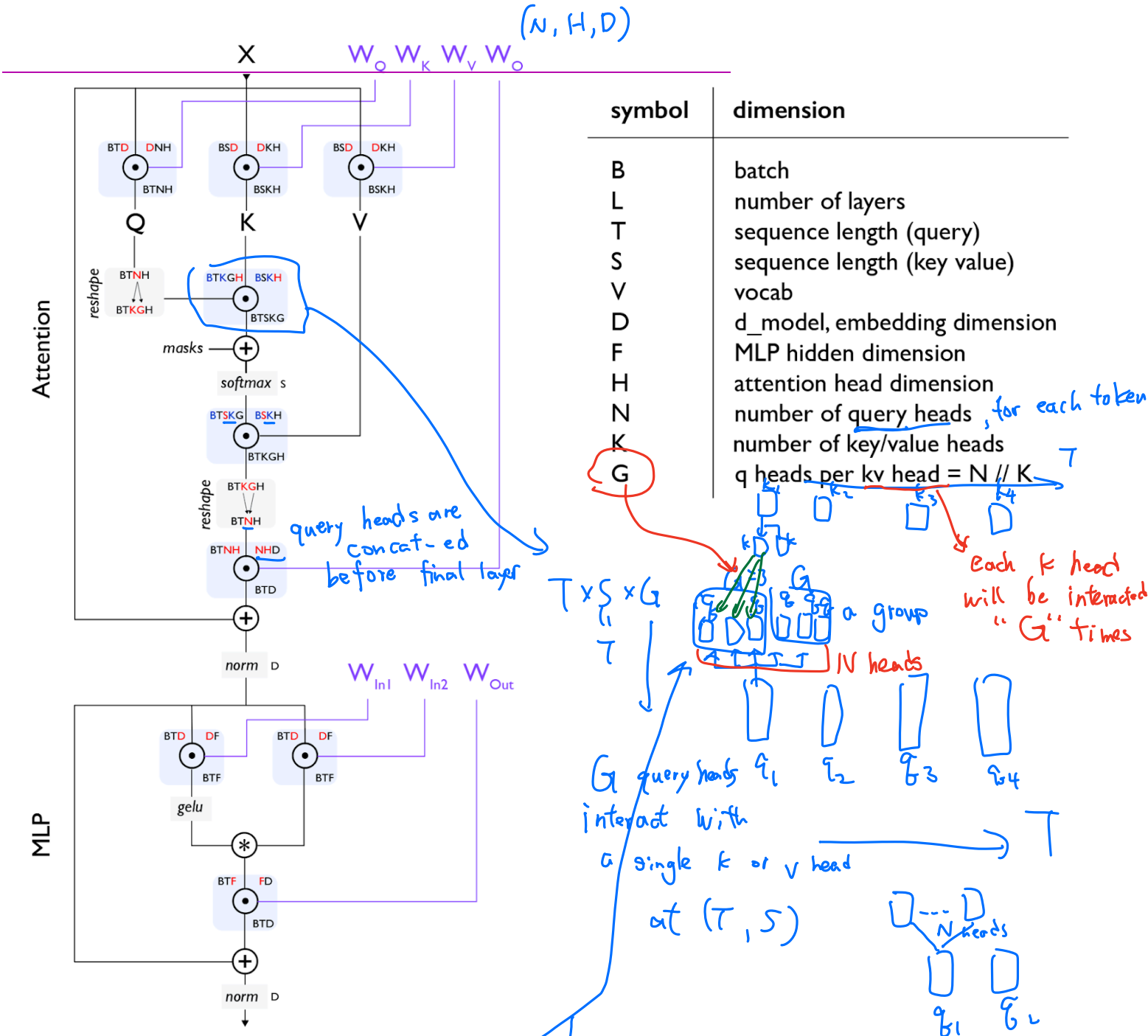


Figure: this diagram shows one layer of a standard Transformer and flows from top-to-bottom. We use a single-letter convention to describe the shapes and layouts of arrays in a Transformer, again showing contracting dimensions in red, and batched dimensions in blue. In a given operation, the input shape is given on top-left and the parameter shape is given on the top-right, with the resulting shape below, e.g. BTD is the input shape for the gating einsum and DF is the weight shape.

Note [gating einsum]: The diagram above uses a "gating einsums" [1] where we split the up-projection matrix into two matrices (W_{In1} and W_{In2} above) whose outputs are elementwise multiplied as a kind of "gating function". Not all LLMs use this, so you will sometimes see a single W_{In} matrix and a total MLP parameter count of 2DF instead of 3DF. Typically in this case, D and F will be scaled up to keep the parameter count the same as the 3 matrix case. With that said, some form of gating einsum is used by LLAMA, DeepSeek, and many other models.

Note 2 [MHA attention]: With self-attention, T and S are the same but for cross-attention they may be different. With vanilla Multi-Head Attention (MHA), N and K are the same while for Multi-Query Attention (MQA) [2] K=1 and for Grouped MQA (GMQA) [3] K merely has to divide N.

Global FLOPs and Params Calculation

For the below we're going to compute per-layer FLOPs to avoid having to stick factors of **L** everywhere.

MLPs

The MLPs of a Transformer typically consist of 2 input matmuls that are element-wise combined and a single output matmul:

operation	train FLOPs	params
$A[B,T,D] \cdot W_{in1}[D,F]$	$6BTDF$	DF
$A[B,T,D] \cdot W_{in2}[D,F]$	$6BTDF$	DF
$\sigma(A_{in1})[B,T,F] * A_{in2}[B,T,F]$	$O(BTF)$	
$A[B,T,F] \cdot W_{out}[F,D]$	$6BTDF$	DF
	$\approx 18BTDF$	$3DF$

Attention

For the generic grouped-query attention case with different **Q** and **KV** head numbers, let us assume equal head dimension H for **Q**,**K**,**V** projections, and estimate the cost of the **QKVO** matmuls:

operation	train FLOPs	params
$A[B,T,D] \cdot W_Q[D,N,H]$	$6BTDNH$	DNH
$A[B,T,D] \cdot W_K[D,K,H]$	$6BTDKH$	DKH
$A[B,T,D] \cdot W_V[D,K,H]$	$6BTDKH$	DKH
$A[B,T,N,H] \cdot W_O[N,H,D]$	$6BTDNH$	DNH
	$12BTD(N+K)H$	$2D(N+K)H$

The dot-product attention operation is more subtle, effectively being a $TH \cdot HS$ matmul batched over the B, K dimensions, a softmax, and a $TS \cdot SH$ matmul again batched over the B, K dimensions. We highlight the batched dims in blue:

operation	train FLOPs
$Q[B,T,K,G,H] \cdot K[B,S,K,H]$	$6BTSKGH = 6BTSNH$
$\text{softmax}_S \;\; L[B,T,S,K,G]$	$O(BTSKG) = O(BTSN)$
$S[B,T,S,K,G] \cdot V[B,S,K,H]$	$6BTSKGH = 6BTSNH$
	$\approx 12BTSNH = 12BT^2NH$

Note [causal masking]: Most recent transformers use a causal mask as opposed to full bidirectional attention. In this case the useful FLOPs of the dot product operations are reduced by a factor of 1/2. To achieve this reduction in practice we need to make use of an attention kernel, rather than a naive einsum.

Other Operations

There are several other operations happening in a Transformer. Layernorms are comparatively cheap and can be ignored for first-order cost estimates. There is also the final enormous (though not per-layer) unembedding matrix multiply.

operation	train FLOPs	params
$\text{layernorm}_D \;\; A[B,T,D]$	$O(BTD)$	D
$A[B,T,D] \cdot W_{unembed}[D,V]$	$6BTDV$	DV

General rule of thumb for Transformer FLOPs

If we neglect the cost of dot-product attention for shorter-context training, then the total FLOPs across all layers is

$$(18BTDF + 12BTD(N+K)H)L = 6 * BT * (3DF + 2D(N+K)H)L$$
$$= 6 * \text{num tokens} * \text{parameter count}$$

Leading to a famous rule of thumb for estimating dense Transformer FLOP count, ignoring the attention FLOPs. (Unembedding is another simple matmul with $6BTDV$ FLOPs and DV params, and follows the same rule of thumb.)

Fractional cost of attention with context length

If we do account for dot-product attention above and assume $F = 4D$, $D = NH$ (as is typical) and $N = K$:

$$\frac{\text{attention FLOPs}}{\text{matmul FLOPs}} = \frac{12BT^2NH}{18BTDF + 24BTDNH} = \frac{12BT^2D}{4 * 18BTD^2 + 24BTD^2} = \frac{12BT^2D}{96BTD^2} = \frac{T}{8D}$$

So the takeaway is that **dot-product attention FLOPs only become dominant during training once T>8D**. For D ~ 8k, this would be ~64K tokens. This makes some sense, since it means as the MLP size increases, the attention FLOPs become less critical. For large models, the quadratic cost of attention is not actually a huge obstacle to longer context training. However, for smaller models, even e.g. Gemma-27B, D=4608 which means attention becomes dominant around 32k sequence lengths. Flash Attention also helps alleviate the cost of long-context, which we discuss briefly in Appendix A.

Miscellaneous Math

Sparsity and Mixture-of-Experts

We'd be remiss not to briefly discuss Mixture of Experts (MoE) models [4], which replace the single dense MLP blocks in a standard Transformer with a set of independent MLPs that can be dynamically routed between. To a first approximation, **an MoE is just a normal dense model with E MLP blocks per layer**, instead of just one. Each token activates k of these experts, typically $k \ll E$. The ratio E/k is called the sparsity and is usually between 8 and 64 (e.g. DeepSeek v3 has effectively $k = 8$, $E = 256$). This increases the parameter count by $O(E)$, while multiplying the total number of activated parameters per token by k, compared with the dense version.

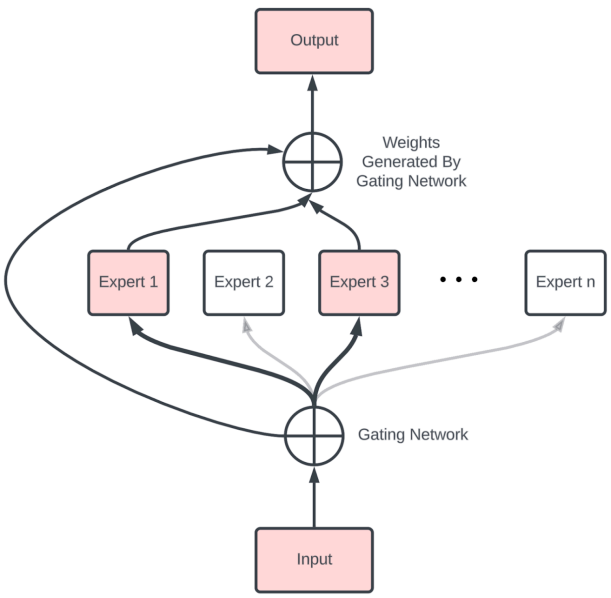


Figure: an example MoE layer with n experts. The gating expert routes each token to k of them, and the output of those k MLPs get summed. Our parameter count is n times the size of each expert, but only k are used for each token. Source.

Compared to a dense model, an MoE introduces new comms, primarily two AllToAlls (one before and one after the MoE block) that route tokens to the correct expert and brings them back to their home device.[1] However as we saw in the previous section, the cost of each AllToAll is only 1/4 that of a comparable AllGather along a single axis (for a bidirectional ring).

Gradient checkpointing

Backpropagation as an algorithm trades memory for compute. Instead of a backward pass requiring $O(n_{\text{layers}}^2)$ FLOPs, **it requires $O(n_{\text{layers}})$ memory**, saving all intermediate activations generated during the forward pass. While this is better than quadratic compute, it's incredibly expensive memory-wise: a

model with $B * T = 4M$ (4M total tokens per batch), L=64, and D=8192 that avoids all unnecessary backward pass compute would have to save roughly $2 * 20 * B * T * D * L = 84TB$ of activations in bfloat16. 20 comes from (roughly) counting every intermediate node in the Transformer diagram above, since e.g.

$$f(x) = \exp(g(x))$$

$$\frac{df}{dx} = \exp(g(x)) \cdot \frac{dg}{dx}$$

so to avoid recomputing we need to save $g(x)$ and $\exp(g(x))$ from the forward pass. To avoid saving this much memory, we can choose to only save some fraction of the intermediate activations. Here are a few strategies we use.

- **Block remat**: only save the input to each layer. This is the most aggressive method we use and only saves 1 checkpoint per layer, meaning we'd only save 4.2TB in the example above. This forces us to repeat essentially all forward pass FLOPs in the backward pass, meaning we increase our FLOPs from $6ND$ to roughly $8ND$.
- **Big matmuls only:** another simple policy is to only save the outputs of large matmuls. This lets us avoid recomputing any large matmuls during the backward pass, but still makes us recompute other activation functions and parts of attention. This reduces 20 per layer to closer to 7 per layer.

This by no means comprehensive. When using JAX, these are typically controlled by `jax.remat` / `jax.checkpoint` (you can read more here).

Key-Value (KV) caching

As we'll see in Section 7, LLM inference has two key parts, prefill and generation.

- **Prefill** processes a long prompt and saves its attention activations in a Key-Value Cache (KV Cache) for use in generation, specifically the key-value projections in the attention block.
- **Generation** batches several of these KV caches together and samples tokens from each of them.

Each KV cache is then effectively an array of size $[2, S, L, K, H]$ where the 2 accounts for the keys and values. This is quite large! The total size of the Key-Value cache in int8 is $2SLKH$. For a moderately-sized model with 8k context length, 64 layers, and $KH = NH = D = 8192$, this is $2 \cdot 8192 \cdot 64 \cdot 8192 = 8\text{GiB}$. You can see why we would want to use GMQA with $K \ll N$.

What Should You Take Away from this Section?

- The overall parameters and FLOPs of a Transformer are fairly easy to calculate, and are summarized here, assuming MHA (with batch size B, vocab size V, a sequence of length T, D=d_{model}, and F=d_{ff}):

Component	Params per layer	Training FLOPs per layer
MLP	3DF	18BTDF
Attention	4DNH	24BTDNH + 12BT^2NH
Other	D	BTD
Vocab	DV (total, not per-layer)	12BTDV

- The parameter count of the MLP block dominates the total parameter count and the MLP block also dominates the FLOPs budget as long as the sequence length $T < 8D$.
- The total FLOPs budget during training is well approximated by $6 \cdot \text{num\_params} \cdot \text{num\_tokens}$ for reasonable context lengths.
- During inference, our KV caches are roughly $2 \cdot S \cdot L \cdot K \cdot H$ per cache (where K is the number of KV heads), although architectural modifications can often reduce this.

A Few Problems to Work

Question 1: How many parameters does a model with $D = 4096$, $F = 4 \cdot D$, $V = 32,000$, and $L = 64$ have? What fraction of these are attention parameters? How large are our KV caches per token? *You can assume $N \cdot H = D$ and multi-head attention with int8 KVs.*

▶ Click here for the answer.

Question 2: How many total FLOPs are required to perform A[B_X, D_Y] *$_D$ W[D_Y, F] on `{'X': 4, 'Y': 8, 'Z': 4}`. How many FLOPs are performed by each TPU?

▶ Click here for the answer.

Question 3: How many FLOPs are involved in performing $A[I, J, K, L] * B[I, J, M, N, O] \to C[K, L, M, N, O]$?

▶ Click here for the answer.

Question 4: What is the arithmetic intensity of self-attention (ignoring the Q/K/V/O projections)? *Give the answer as a function of the Q and KV lengths T and S.* At what context length is attention FLOPs-bound? Given the HBM bandwidth of our TPUs, plot the effective relative cost of attention to the FFW block as the context length grows.

▶ Click here for the answer.

Question 5: At what sequence length are self-attention FLOPs equal to the QKVO projection FLOPs?

▶ Click here for the answer.

Question 6: Say we only save the output of each of the 7 main matmuls in a Transformer layer during our forward pass (Q, K, V, O + the three FFW matrices). How many extra FLOPs do we need to "rematerialize" during the backwards pass?

▶ Click here for the answer.

Question 7: DeepSeek v3 says it was trained for 2.79M H800 hours on 14.8T tokens (source). Given that it has 37B activated parameters, roughly what hardware utilization did they achieve? *Hint: note that they used FP8 FLOPs without structured sparsity.*

▶ Click here for the answer.

Question 8: Mixture of Experts (MoE) models have E copies of a standard dense MLP block, and each token activates k of these experts. What batch size in tokens is required to be compute-bound for an MoE with weights in int8 on TPU v5e? For DeepSeek, which has 256 (routed) experts and $k = 8$, what is this number?

▶ Click here for the answer.

That's it for Part 4! For Part 5 (about scaling Transformer training), click here!

Appendix

Appendix A: How does Flash Attention work?

The traditional objection to scaling Transformers to very long context is that the attention FLOPs and memory usage scale quadratically with context length. While it's true that the attention QK product has shape $[B, T, S, N]$ where B is the batch size, S and T are the Q and K sequence dims, and N is the number of heads, this claim comes with some serious caveats:

1. As we noted earlier, even though this is quadratic, the attention FLOPs only dominated when $S > 8 \cdot D$, and especially during training the memory of a single attention matrix is small compared to all of the weights and activation checkpoints living in memory, especially when sharded.
2. We don't need to materialize the full attention matrix in order to compute attention! We can compute local sums and maxes and avoid ever materializing more than a small chunk of the array. While the total FLOPs is still quadratic, we drastically reduce memory pressure.

This second observation was first made by Rabe et al. 2021 and later in the Flash Attention paper (Dao et al. 2022). The basic idea is to compute the attention in chunks of K/V, where we compute the local softmax and some auxiliary statistics, then pass them onto the next chunk which combines them with its local chunk. Specifically, we compute

1. **M:** The running max of $q \cdot k$ over the sequence dimension
2. **O:** The running full attention softmax over the sequence dimension
3. **L:** The running denominator $\sum_i (q \cdot k_i - \text{running max})$

With these, we can compute the new max, the new running sum, and the new output with only a constant amount of memory. To give a sketchy description of how this works, attention is roughly this operation:

$$\text{Attn}(Q,K,V) = \sum_i \frac{\exp(Q\cdot K_i - \max_j Q\cdot K_j)V_i}{\sum_l \exp(Q\cdot K_l - \max_j Q\cdot K_j)}$$

The max is subtracted for numerical stability and can be added without affecting the outcome since $\sum_i \exp(a_i + b) = \exp(b)\sum \exp(a)$. Looking just at the denominator above, if we imagine having two contiguous chunks of key vectors, K^1 and K^2 and we compute the local softmax sums L^1 and L^2 for each

$$L^1 = \sum_i \exp(Q\cdot K_i^1 - \max_j Q\cdot K_j^1)$$

$$L^2 = \sum_i \exp(Q\cdot K_i^2 - \max_j Q\cdot K_j^2)$$

Then we can combine these into the full softmax sum for these two chunks together by using

$$L^{\text{combined}} = \exp(M^1 - \max(M^1, M^2))\cdot L^1 + \exp(M^2 - \max(M^1, M^2))\cdot L^2$$

where

$$M^1 = \max_j Q\cdot K_j^1 \text{ and } M^2 = \max_j Q\cdot K_j^2$$

This can be done for the full softmax as well, giving us a way of accumulating arbitrarily large softmax sums. Here's the full algorithm from the Flash Attention paper.

Algorithm 1 FlashAttention

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N\times d}$ in HBM, on-chip SRAM of size M.

1: Set block sizes $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min\left(\lceil \frac{M}{4d} \rceil, d\right)$.
2: Initialize $\mathbf{O} = (0)_{N\times d} \in \mathbb{R}^{N\times d}$, $\ell = (0)_N \in \mathbb{R}^N$, $m = (-\infty)_N \in \mathbb{R}^N$ in HBM.
3: Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \ldots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \ldots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \ldots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
4: Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \ldots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \ldots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \ldots, m_{T_r}$ of size B_r each.
5: **for** $1 \le j \le T_c$ **do**
6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
7: **for** $1 \le i \le T_r$ **do**
8: Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
9: On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
10: On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
11: On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}}\ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}}\tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
12: Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1}(\text{diag}(\ell_i)e^{m_i - m_i^{\text{new}}}\mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}}\tilde{\mathbf{P}}_{ij}\mathbf{V}_j)$ to HBM.
13: Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.
14: **end for**
15: **end for**
16: Return $\mathbf{O}$.

From a hardware standpoint, this lets us fit our chunk of Q into VMEM (what the algorithm above calls on-chip SRAM) so we only have to load the KV chunks on each iteration, increasing the arithmetic intensity. We can also keep the running statistics in VMEM.

One last subtle point worth emphasizing is an attention softmax property that's used to make the Flash VJP (reverse mode derivative) calculation practical for training. If we define an intermediate softmax array as:

$$S_{ij} = \frac{e^{\tau q_i \cdot k_j}}{\sum_l e^{\tau q_i \cdot k_l}}$$

In attention, we obtain *dS* from reverse-mode *dO* and *V* arrays:

$$dS_{ij} = dO_{id} \cdot_d V_{jd} = \sum_d dO_{id} V_{jd}$$

During the backpropagation of this gradient to Q and K

$$d(q_i \cdot k_j) = (dS_{ij} - S_{ij} \cdot_j dS_{ij}) S_{ij}$$

We exploit an identity that allows us to exchange a contraction along the large key **length** dimension with a local contraction along the feature **depth** dimension.

$$\begin{aligned} S_{ij} \cdot_j dS_{ij} &= \sum_j \frac{e^{\tau q_i \cdot k_j}}{\sum_k e^{\tau q_i \cdot k_k}} \sum_d dO_{id} V_{jd} \\ &= \sum_d dO_{id} \sum_j \frac{e^{\tau q_i \cdot k_j}}{\sum_k e^{\tau q_i \cdot k_k}} V_{jd} \\ &= \sum_d dO_{id} O_{id} \\ &= dO_{id} \cdot_d O_{id} \end{aligned}$$

This replacement is crucial for being able to implement a sequence-block *local* calculation for the VJP, and enables further clever sharding schemes like ring attention.

Footnotes

1. Technically, this only happens if we are data or sequence sharded along the same axis as our experts. [↩]

References

1. **GLU Variants Improve Transformer**
   Shazeer, N., 2020. arXiv [cs.LG].
2. **Fast Transformer decoding: One write-head is all you need**
   Shazeer, N., 2019. arXiv [cs.NE].
3. **GQA: Training generalized multi-query transformer models from multi-head checkpoints**
   Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebrón, F. and Sanghai, S., 2023. arXiv [cs.CL].
4. **Outrageously large neural networks: The Sparsely-Gated Mixture-of-experts layer**
   Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G. and Dean, J., 2017. arXiv [cs.LG].

Miscellaneous

*Work done at Google DeepMind, now at MatX.

Citation

For attribution in academic contexts, please cite this work as:

```
Austin et al., "How to Scale Your Model", Google DeepMind, online, 2025.
```

or as a BibTeX entry:

```
@article{scaling-book,
  title = {How to Scale Your Model},
  author = {Austin, Jacob and Douglas, Sholto and Frostig, Roy and Levskaya, Anselm and Chen, Charlie and Vikram, Sharad
  and Lebron, Federico and Choy, Peter and Ramasesh, Vinay and Webson, Albert and Pope, Reiner},
  publisher = {Google DeepMind},
  howpublished = {Online},
  note = {Retrieved from https://jax-ml.github.io/scaling-book/},
  year = {2025}
}
```

12 reactions

14 comments · 22 replies — *powered by giscus*

Oldest Newest