# How to Parallelize a Transformer for Training

Part 5 of How To Scale Your Model (Part 4: Transformers | Part 6: Training LLaMA)

Here we discuss four main parallelism schemes used during LLM training: data parallelism, fully-sharded data parallelism (FSDP), tensor parallelism, and pipeline parallelism. For each, we calculate at what point we become bottlenecked by communication.


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


## What Do We Mean By Scaling?

The goal of "model scaling" is to be able to increase the number of chips used for training or inference while achieving a proportional, linear increase in throughput (we call this *strong scaling*). While performance on a single chip depends on the trade-off between memory bandwidth and FLOPs, performance at the cluster level depends on hiding inter-chip communication by overlapping it with useful FLOPs. This is non-trivial, because increasing the number of chips increases the communication load while reducing the amount of per-device computation we can use to hide it. As we saw in Section 3, sharded matrix multiplications often require expensive AllGathers or ReduceScatters that can block the TPUs from doing useful work. The goal of this section is to find out when these become *too expensive*.

↳ FLOPs

In this section, we'll discuss four common parallelism schemes: (pure) **data parallelism, fully-sharded data parallelism** (FSDP / ZeRO sharding), **tensor parallelism** (also known as model parallelism), and (briefly) **pipeline parallelism**. For each, we'll show what communication cost we incur and at what point that cost starts to bottleneck our compute cost.[1] For this section, you can focus solely on inter-chip communication costs, since as long as we have a large enough single-chip batch size, the transfer of data from HBM to MXU is already overlapped with computation.

We'll use the following notation to simplify calculations throughout this section.

| Notation | Meaning (model parameters) |
|---|---|
| D | $\mathbf{d}_{\text{model}}$ (the hidden dimension/residual stream dim) |
| F | $\mathbf{d}_{\text{ff}}$ (the feed-forward dimension) |
| B | Batch dimension (number of tokens in the batch; total, not per-device) |
| T | Sequence length |
| L | Number of layers in the model |

global batch dimension

| Notation | Meaning (hardware characteristic) |
|---|---|
| C | FLOPS/s per chip |
| W | Network bandwidth (bidirectional, often subscripted as e.g. $W_{\text{ici}}$ or $W_{\text{dcn}}$) |
| X | Number of chips along mesh axis X |
| Y | Number of chips along an alternate mesh axis, labeled Y |
| Z | Number of chips along a third mesh axis, labeled Z |

previous section: $T > 8D \Leftrightarrow$ attention is dominant

For simplicity's sake, **we'll approximate a Transformer as a stack of MLP blocks —** attention is a comparatively small fraction of the FLOPs for larger models as we saw in Section 4. We will also ignore the gating matmul, leaving us with the following simple structure for each layer:

# A Transformer layer:

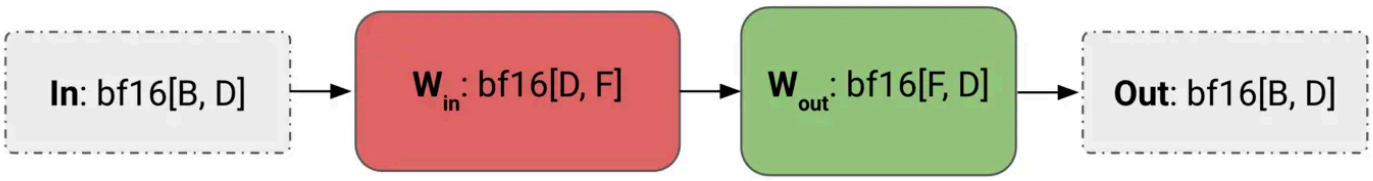



Figure: a simplified Transformer layer. We treat each FFW block as a stack of two matrices $W_{in}$: `bf16[D, F]` (up-projection) and $W_{out}$: `bf16[F, D]` (down-projection) with an input In: `bf16[B, D]`.

▼ Here's the full algorithm for our little Transformer with no parallelism.

**Forward pass:** need to compute Loss[B]

1. Tmp[B, F] = In[B, D] $*_D$ $W_{in}$[D, F] → 2BDF matmul
2. Out[B, D] = Tmp[B, F] $*_F$ $W_{out}$[F, D]
3. Loss[B] = ...

**Backward pass:** need to compute $dW_{out}$[F, D], $dW_{in}$[D, F]

1. dOut[B, D] = ...
2. $dW_{out}$[F, D] = Tmp[B, F] $*_B$ dOut[B, D] → 2BDF matmul $\frac{\partial L}{\partial W_{out}} = Tmp^T \frac{\partial L}{\partial out}$
3. dTmp[B, F] = dOut[B, D] $*_D$ $W_{out}$[F, D] → 2BDF $\frac{\partial L}{\partial Tmp} = \frac{\partial L}{\partial out} W_{out}^T$
4. $dW_{in}$[D, F] = In[B, D] $*_B$ dTmp[B, F]
5. dIn[B, D] = dTmp[B, F] $*_F$ $W_{in}$[D, F] (*needed for previous layers*)

2x of forward flops

We provide this for comparison to the algorithms with communication added.

Here are the 4 parallelism schemes we will discuss. Each scheme can be thought of as uniquely defined by a sharding for **In**, **$W_{in}$**, **$W_{out}$**, **and Out** in the above diagram.

**1. Data parallelism:** *activations sharded along batch, parameters and optimizer state are replicated on each device. Communication only occurs during the backwards pass.*

$$\text{In}[B_X, D] \cdot_D W_{\text{in}}[D, F] \cdot_F W_{\text{out}}[F, D] \to \text{Out}[B_X, D]$$

**2. Fully-sharded data parallelism (FSDP or ZeRO-3):** *activations sharded along batch (like pure data parallelism), parameters sharded along same mesh axis and AllGathered just-in-time before use in forward pass. Optimizer state also sharded along batch. Reduces duplicated memory.*

$$\text{In}[B_X, D] \cdot_D W_{\text{in}}[D_X, F] \cdot_F W_{\text{out}}[F, D_X] \to \text{Out}[B_X, D]$$

**3. Tensor parallelism (also called Megatron sharding or model parallelism):** *activations sharded along D ($d_{\text{model}}$), parameters sharded along F ($d_{ff}$). AllGather and ReduceScatter activations before and after each block. Compatible with FSDP.*

$$\text{In}[B, D_Y] \cdot_D W_{\text{in}}[D, F_Y] \cdot_F W_{\text{out}}[F_Y, D] \to \text{Out}[B, D_Y]$$

**4. Pipeline parallelism:** *weights sharded along the layer dimension, activations microbatched and rolled along the layer dimension. Communication between pipeline stages is minimal (just moving activations over a single hop). To abuse notation:* → using jax.lax.ppermute

$$\text{In}[L_Z, B, D][i] \cdot_D W_{\text{in}}[L_Z, D, F][i] \cdot_F W_{\text{out}}[L_Z, F, D][i] \to \text{Out}[L_Z, B, D][i]$$

sharded over layers

## Data Parallelism

**Syntax:** $\text{In}[B_X, D] \cdot_D W_{\text{in}}[D, F] \cdot_F W_{\text{out}}[F, D] \to \text{Out}[B_X, D]$

When your model fits on a single chip with even a tiny batch size (>240 tokens, so as to be compute-bound), **you should always use simple data parallelism.** Pure data parallelism splits our activations across any number of TPUs so long as the number of TPUs is smaller than our batch size. The forward pass involves no communication, but at the end of every step, **each TPU performs an AllReduce on its local gradients to synchronize them before updating the parameters.**

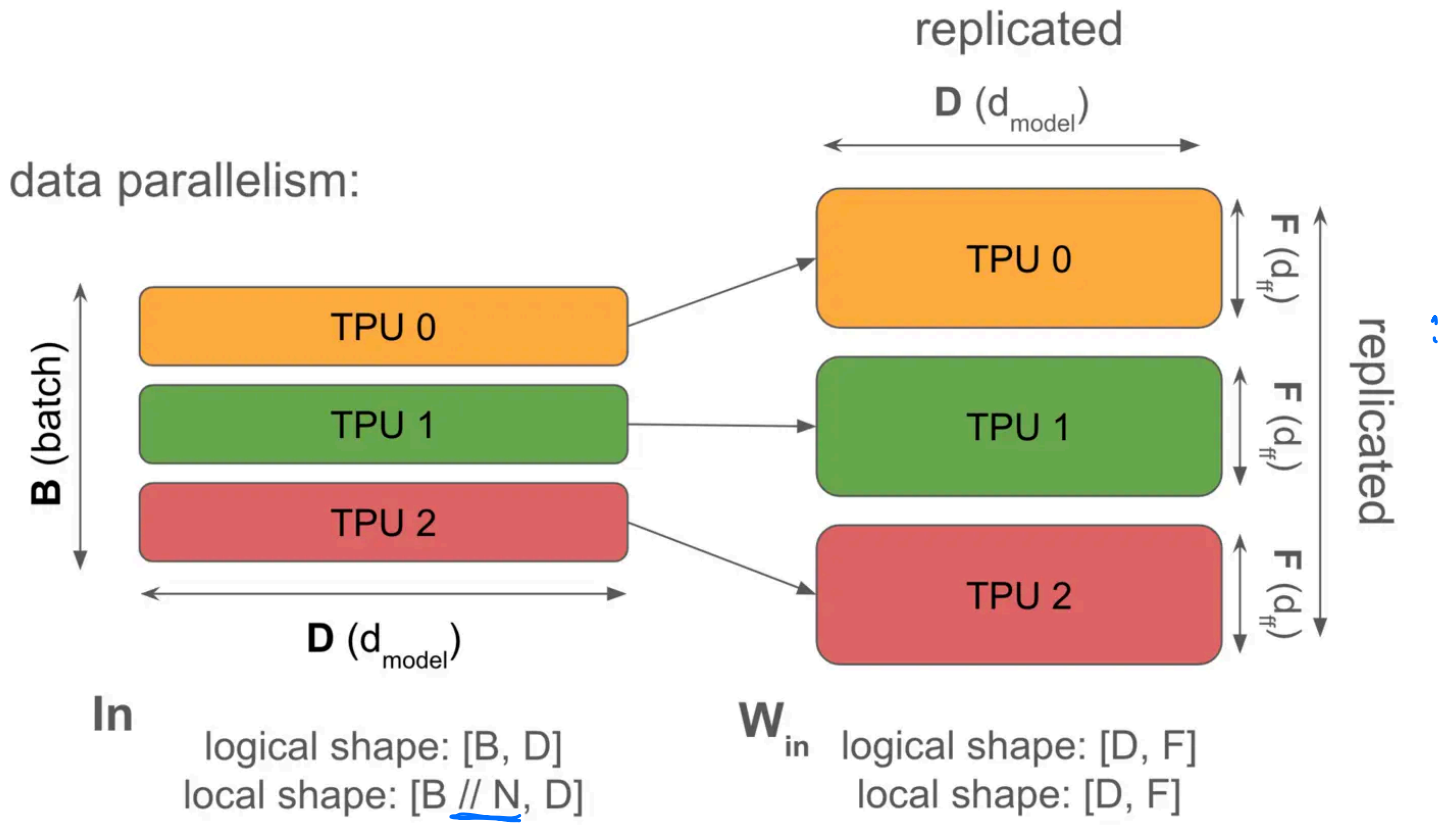



**Figure:** a diagram of pure data parallelism (forward pass). Our activations (left) are fully sharded along the batch dimension and our weights are fully replicated, so each TPU has an identical copy of the weights. This means the total memory of our weights is increased by a factor of N, but no communication is required on the forward-pass.

▼ Here's the full algorithm for the forward and backwards pass. We abuse notation to write dL/dOut as dOut, purely for compactness.

**Pure Data Parallelism Algorithm:**

**Forward pass:** need to compute Loss[$B_X$]

1. Tmp[$B_X$, F] = In[$B_X$, D] *$_D$ $W_{in}$[D, F]
2. Out[$B_X$, D] = Tmp[$B_X$, F] *$_F$ $W_{out}$[F, D]
3. Loss[$B_X$] = ...

**Backward pass:** need to compute $dW_{out}$[F, D], $dW_{in}$[D, F]

1. dOut[$B_X$, D] = ...
2. $dW_{out}$[F, D] {$U_X$} = Tmp[$B_X$, F] *$_B$ dOut[$B_X$, D]
3. $dW_{out}$[F, D] = **AllReduce**($dW_{out}$[F, D] {$U_X$}) (*not on critical path, can be done async*)
4. dTmp[$B_X$, F] = dOut[$B_X$, D] *$_D$ $W_{out}$[F, D]
5. $dW_{in}$[D, F] {$U_X$} = In[$B_X$, D] *$_B$ dTmp[$B_X$, F]
6. $dW_{in}$[D, F] = **AllReduce**($dW_{in}$[D, F] {$U_X$}) (*not on critical path, can be done async*)
7. dIn[$B_X$, D] = dTmp[$B_X$, F] *$_F$ $W_{in}$[D, F] (*needed for previous layers*)

what does this mean? I think it meant: we can still compute $\frac{\partial L}{\partial Tmp}$ while dWout's AllReduce is executing

We ignore the details of the loss function and abbreviate $\text{Tmp} = W_{\text{in}} \cdot \text{In}$. Note that, although our final loss is the average **AllReduce**(Loss[$B_X$]), we only need to compute the AllReduce on the backward pass when averaging weight gradients.

Note that the forward pass has no communication — **it's all in the backward pass**! The backward pass also has the great property that the AllReduces aren't in the "critical path", meaning that each AllReduce can be performed whenever it's convenient and doesn't block you from performing subsequent operations. The overall communication cost *can still bottleneck us* if it exceeds our total compute cost, but it is much more forgiving from an implementation standpoint. We'll see that model/tensor parallelism doesn't have this property.

**Why do this?** Pure data parallelism reduces activation memory pressure by splitting our activations over the batch dimension, allowing us to almost arbitrarily increase batch size as long as we have more chips to split the batch dimension over. Especially during training when our activations often dominate our memory usage, this is very helpful.

**Why not do this?** Pure data parallelism does nothing to reduce memory pressure from model parameters or optimizer states, which means pure data parallelism is rarely useful for interesting models at scale where our parameters + optimizer state don't fit in a single TPU. To give a sense of scale, if we train with parameters in bf16 and optimizer state in fp32 with Adam[2], the largest model we can fit has $\text{TPU memory}/10$ parameters, so e.g. on a TPUv5p chip with 96GB of HBM and pure data parallelism this is about 9B parameters.

**Takeaway**: the largest model we can train with Adam and pure data parallelism has $\text{num\_params} = \text{HBM per device}/10$. For TPU v5p this is roughly 9B parameters.[3]

*To make this useful for real models during training, we'll need to at least partly shard the model parameters or optimizer.*

**When do we become bottlenecked by communication?** As we can see above, we have two AllReduces per layer, each of size $2DF$ (for bf16 weights). When does data parallelism make us communication bound?

As in the table above, let $C$ = per-chip FLOPs, $W_{\text{ici}}$ = **bidirectional** network bandwidth, and $X$ = number of shards across which the batch is partitioned[4]. Let's calculate the time required to perform the relevant matmuls, $T_{\text{math}}$, and the required communication time $T_{\text{comms}}$. Since this parallelism scheme requires no communication in the forward pass, we only need to calculate these quantities for the backwards pass.

*Communication time:* From a previous section we know that the time required to perform an AllReduce in a 1D mesh depends only on the total bytes of the array being AllReduced and the ICI bandwidth $W_{\text{ici}}$; specifically the AllReduce time is $2 \cdot \text{total bytes}/W_{\text{ici}}$. Since we need to AllReduce for both $W_{\text{in}}$ and $W_{\text{out}}$, we have 2 AllReduces per layer. Each AllReduce is for a weight matrix, i.e. an array of $DF$ parameters, or $2DF$ bytes. Putting this all together, the total time for the AllReduce in a single layer is

$$T_{\text{comms}} = \frac{2 \cdot 2 \cdot 2 \cdot D \cdot F}{W_{\text{ici}}} \quad (1)$$

*Matmul time:* Each layer comprises two matmuls in the forward pass, or four matmuls in the backwards pass, each of which requires $2(B/X)DF$ FLOPs. Thus, for a single layer in the backward pass, we have

$$T_{\text{math}} = \frac{2 \cdot 2 \cdot 2 \cdot B \cdot D \cdot F}{X \cdot C} \quad (2)$$

Since we overlap, the total time per layer is the max of these two quantities:

$$T \approx \max\left(\frac{8 \cdot B \cdot D \cdot F}{X \cdot C}, \frac{8 \cdot D \cdot F}{W_{\text{ici}}}\right)$$

$$T \approx 8 \cdot D \cdot F \cdot \max\left(\frac{B}{X \cdot C}, \frac{1}{W_{\text{ici}}}\right)$$

We become compute-bound when $T_{\text{math}}/T_{\text{comms}} > 1$, or when

$$\frac{B}{X} > \frac{C}{W_{\text{ici}}} \quad (3)$$

The upshot is that, to remain compute-bound with data parallelism, we need the per-device batch size $B/X$ to exceed the ICI operational intensity, $C/W_{\text{ici}}$. This is ultimately a consequence of the fact that the computation time scales with the per-device batch size, while the communication time is independent of this quantity (since we are transferring model weights). Note the resemblance of the $B/X > C/W_{\text{ici}}$ condition to the single-device compute-bound rule $B > 240$; in that case as well, the rule came from the fact that computation time scaled with batch size while data-transfer size was (in the $B \ll F, D$ regime) independent of batch size.

Let's put in some real numbers to get a sense of scale. For TPUv5p, `C=4.6e14` and `W=2 * 9e10` for 1D data parallelism over ICI, so **our batch size per chip must be at least 2,550 to avoid being communication-bound**. Since we can do data parallelism over multiple axes, if we dedicate all three axes of a TPUv5p pod to pure data parallelism, we 3x our bandwidth $W_{\text{ici}}$ and can scale down to only BS=850 per TPU or 7.6M tokens per batch per pod (of 8960 chips)! **This tells us that it's fairly hard to become bottlenecked by pure data parallelism!**

**Note [context parallelism]:** Throughout this section, $B$ always refers to the total batch size **in tokens**. Clearly, however, our batch is made up of many different sequences, so how does this work? As far as the MLP is concerned, **tokens are tokens**! It doesn't matter if they belong to the same sequence or two different sequences. So we are more or less free to do data parallelism over both the batch and sequence dimension: we call this context parallelism or sequence parallelism, but you can think of it as simply being another kind of data parallelism. Attention is trickier than the MLP since we do some cross-sequence computation, but this can be handled by gathering KVs or Qs during attention and carefully overlapping FLOPs and comms (typically using something called "ring attention"). Throughout this section, we will just ignore our sequence dimension entirely and assume some amount of batch or sequence parallelism.

**Note on multiple mesh axes:** We should quickly note how multiple axes affects the available bandwidth. When we use multiple mesh axes for a given parallelism strategy, we get more bandwidth.

- **Definition:** $M_X$ ($M_Y$, $M_Z$, etc.) is the number of hardware mesh axes that a given parallelism strategy spans.
- **Effect (bandwidth-bound):** Using $M$ axes provides ($\approx M$ times) aggregate link bandwidth, so collective time scales $\propto 1/M_X$.

### Fully-Sharded Data Parallelism (FSDP)

**Syntax:** $\text{In}[B_X, D] \cdot_D W_{\text{in}}[D_X, F] \cdot_F W_{\text{out}}[F, D_X] \to \text{Out}[B_X, D]$

Fully-sharded data parallelism (often called FSDP or ZeRO-sharding [1]) splits the model optimizer states and weights across the data parallel shards and efficiently gathers and scatters them as needed. **Compared to pure data parallelism, FSDP drastically reduces per-device memory usage and saves on backward pass FLOPs, with very minimal overhead.**

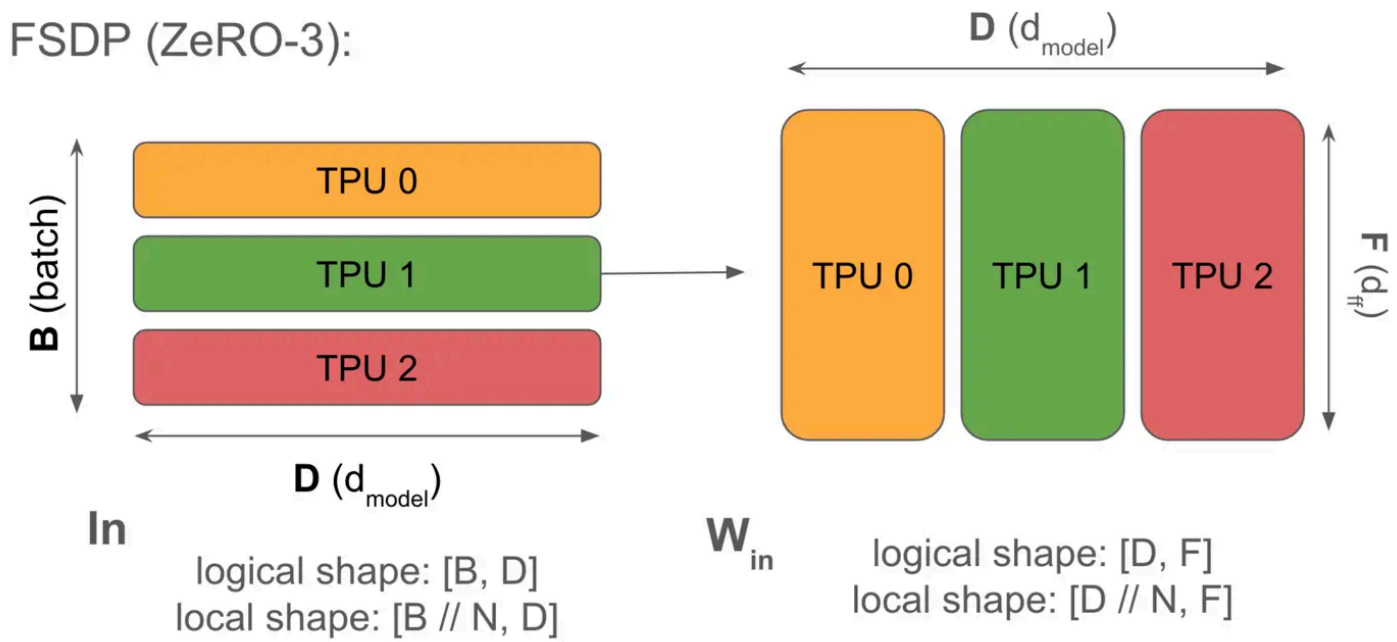



**Figure:** FSDP shards the contracting dimension of Win and the output dimension of Wout along the data dimension. This reduces memory but (from Section 3) requires us to gather the weights for W before we perform the matmul. Note that the activations (left) are not sharded along the contracting dimension, which is what forces us to gather. **Note that our weight optimizer state is likewise sharded along the contracting dimension.**

You'll remember (from Section 3) that an AllReduce can be decomposed into an AllGather and a ReduceScatter. This means that, instead of doing the full gradient AllReduce for standard data parallelism, we can shard the weights and optimizer states across chips, AllGather them at each layer during the forward pass and ReduceScatter across the weights during the backward pass at no extra cost.

▼ Here's the full algorithm for FSDP.

**Fully-Sharded Data Parallelism (FSDP):**

**Forward pass:** need to compute Loss[$B_X$]

1. $W_{in}[D, F]$ = **AllGather**($W_{in}[D_X, F]$) (*not on critical path, can do it during previous layer*)
2. $Tmp[B_X, F] = In[B_X, D] *_D W_{in}[D, F]$ (*can throw away $W_{in}[D, F]$ now*)
3. $W_{out}[F, D]$ = **AllGather**($W_{out}[F, D_X]$) (*not on critical path, can do it during previous layer*)
4. $Out[B_X, D] = Tmp[B_X, F] *_F W_{out}[F, D]$
5. Loss[$B_X$] = ...

**Backward pass:** need to compute $dW_{out}[F, D_X]$, $dW_{in}[D_X, F]$

1. $dOut[B_X, D]$ = ...
2. $dW_{out}[F, D]$ {$U_X$} = $Tmp[B_X, F]$ *$_B$ $dOut[B_X, D]$
3. $dW_{out}[F, D_X]$ = **ReduceScatter**($dW_{out}[F, D]$ {$U_X$}) (*not on critical path, can be done async*)
4. $W_{out}[F, D]$ = **AllGather**($W_{out}[F, D_X]$) (*can be done ahead of time*)
5. $dTmp[B_X, F]$ = $dOut[B_X, D]$ *$_D$ $W_{out}[F, D]$ *(can throw away $W_{out}$[F, D] here)*
6. $dW_{in}[D,F]$ {$U_X$} = $dTmp[B_X, F]$ *$_B$ $In[B_X, D]$
7. $dW_{in}[D_X, F]$ = **ReduceScatter**($dW_{in}[D, F]$ {$U_X$}) *(not on critical path, can be done async)*
8. $W_{in}[D, F]$ = **AllGather**($W_{in}[D_X, F]$) (*can be done ahead of time*)
9. $dIn[B_X, D]$ = $dTmp[B_X, F]$ *$_F$ $W_{in}[D, F]$ (*needed for previous layers) (can throw away $W_{in}$[D, F] here*)

This is also called "ZeRO Sharding", from "ZeRo Overhead sharding" since we don't perform any unnecessary compute or store any unnecessary state. ZeRO-{1,2,3} are used to refer to sharding the optimizer states, gradients, and weights in this way, respectively. Since all have the same communication cost[5], we can basically always do ZeRO-3 sharding, which shards the parameters, gradients, and optimizer states across a set of devices.

**Why would we do this?** Standard data parallelism involves a lot of duplicated work. Each TPU AllReduces the full gradient, then updates the full optimizer state (identical work on all TPUs), then updates the parameters (again, fully duplicated). For ZeRO sharding (sharding the gradients/optimizer state), instead of an AllReduce, you can ReduceScatter the gradients, update only your shard of the optimizer state, update a shard of the parameters, then AllGather the parameters as needed for your forward pass.

**When do we become bottlenecked by communication?** Our relative FLOPs and comms costs are exactly the same as pure data parallelism, since each AllReduce in the backward pass has become an AllGather + ReduceScatter. Recall that an AllReduce is implemented as an AllGather and a ReduceScatter, each with half the cost. Here we model the forward pass since it has the same FLOPs-to-comms ratio as the backward pass:

$$
\begin{aligned}
T_{\text{math}} &= \frac{2 \cdot 2 \cdot B \cdot D \cdot F}{X \cdot C} \\
T_{\text{comms}} &= \frac{2 \cdot 2 \cdot D \cdot F}{W_{\text{ici}}} \\
T &\approx \max\left(\frac{4 \cdot B \cdot D \cdot F}{X \cdot C}, \frac{4 \cdot D \cdot F}{W_{\text{ici}}}\right) \\
T &\approx 4 \cdot D \cdot F \cdot \max\left(\frac{B}{X \cdot C}, \frac{1}{W_{\text{ici}}}\right)
\end{aligned}
$$

Therefore, as with pure data-parallelism, we are compute bound when $B/X > C/W_{\text{ici}}$, i.e. when the per-device batch size $B/X$ exceeds the "ICI operational intensity" $C/W_{\text{ici}}$ (`4.59e14 / 1.8e11 = 2550` for v5p). This is great for us, because it means if our per-device batch size is big enough to be compute-bound for pure data-parallelism, we can — without worrying about leaving the compute-bound regime — simply upgrade to FSDP, saving ourselves a massive amount of parameter and optimizer state memory! Though we did have to add communication to the forward pass, this cost is immaterial since it just overlaps with forward-pass FLOPs.

**Takeaway:** Both FSDP and pure Data Parallelism become bandwidth bound on TPUv5 when the batch size per device is less than $2550/M_X$, where $M_X$ is the number of mesh axes.

For example, DeepSeek-V2 (one of the only recent strong model to release information about its training batch size) used a batch size of ~40M tokens. **This would allow us to scale to roughly 47,000 chips, or around 5 TPUv5 pods, before we hit a bandwidth limit.**

For LLaMA-3 70B, which was trained for approximately `6.3e24 (15e12 * 70e9 * 6)` FLOPs, we could split a batch of 16M tokens over roughly `16e6 / (2550 / 3) = 18,823` chips (roughly 2 pods of 8960 chips), each with `4.59e14` FLOPs running at 50% peak FLOPs utilization (often called MFU), and **train it in approximately 17 days**. Not bad! But let's explore how we can do better.

**Note on critical batch size**: somewhat unintuitively, we become more communication bottlenecked as our total batch size decreases (with fixed chip number). Data parallelism and FSDP let us scale to arbitrarily many chips so long as we can keep increasing our batch size! However, in practice, as our batch size increases, we tend to see diminishing returns in training since our gradients become almost noise-free. We also sometimes see training instability. Thus, the game of finding an optimal sharding scheme in the "unlimited compute regime" often starts from a fixed batch size, determined by scaling laws, and a known (large) number of chips, and then aims to find a partitioning that allows us to fit that small batch size on so many chips.

## Tensor Parallelism

**Syntax:** $\text{In}[B, D_Y] \cdot_D W_{\text{in}}[D, F_Y] \cdot_F W_{\text{out}}[F_Y, D] \to \text{Out}[B, D_Y]$ (we use $Y$ to eventually combine with FSDP)

In a fully-sharded data-parallel AllReduce we move the weights across chips. We can also shard the feedforward dimension of the model and move the activations during the layer — this is called "1D model parallelism" or Megatron sharding [2]. This can unlock a smaller efficient batch size per pod. The figure below shows an example of a single matrix sharded in this way:

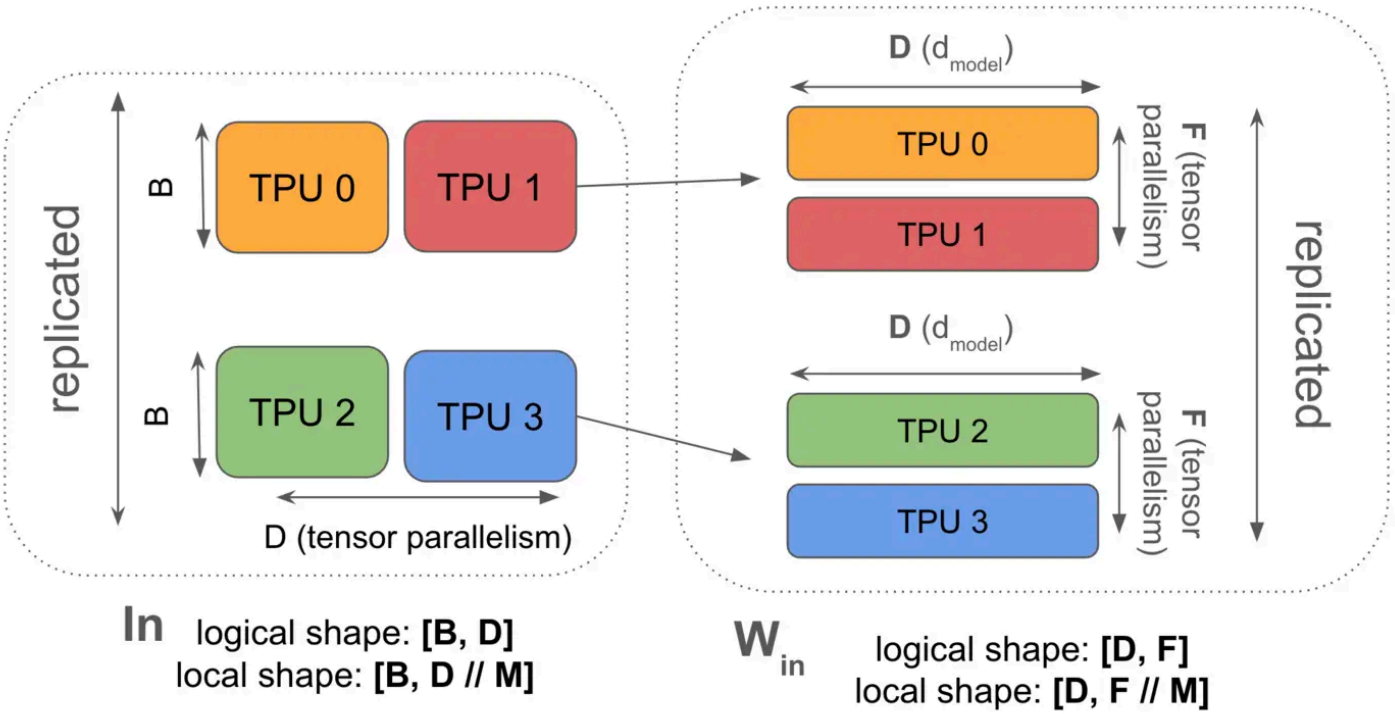



**Figure:** an example of basic tensor parallelism. Since we're only sharding our activations over Y (unlike in FSDP where we shard over X), we replicate our activations over X. Using our standard syntax, this is A[B, $D_Y$] * B[D, $F_Y$] -> C[B, $F_Y$]. Because we're only sharding over one of the contracting dimensions, we typically AllGather the activations A before the matmul.

As noted, **In[B, $D_Y$] *$_D$ $W_{in}$[D, $F_Y$] *$_F$ $W_{out}$[$F_Y$, D] -> Out[B, $D_Y$] means we have to gather our activations before the first matmul. This is cheaper than ZeRO sharding when the activations are smaller than the weights.** This is typically true only with some amount of ZeRO sharding added (which reduces the size of the gather). This is one of the reasons we tend to mix ZeRO sharding and tensor parallelism.

▼ Here's the algorithm for tensor parallelism!

**Tensor Parallelism:**

**Forward pass:** need to compute Loss[B]

1. In[B, D] = **AllGather**(In[B, $D_Y$]) *(on critical path)*
2. Tmp[B, $F_Y$] = In[B, D] *$_D$ $W_{in}$[D, $F_Y$] *(not sharded along contracting, so no comms)*
3. Out[B, D] {$U_Y$} = Tmp[B, $F_Y$] *$_F$ $W_{out}$[$F_Y$, D]
4. Out[B, $D_Y$] = **ReduceScatter**(Out[B, D] {$U_Y$}) *(on critical path)*
5. Loss[B] = ...

**Backward pass:** need to compute $dW_{out}$[$F_Y$, D], $dW_{in}$[D, $F_Y$]

1. dOut[B, $D_Y$] = ...
2. dOut[B, D] = **AllGather**(dOut[B, $D_Y$]) *(on critical path)*
3. $dW_{out}$[$F_Y$, D] = Tmp[B, $F_Y$] *$_B$ dOut[B, D]
4. dTmp[B, $F_Y$] = dOut[B, D] *$_D$ $W_{out}$[$F_Y$, D] *(can throw away dOut[B, D] here)*
5. In[B, D] = **AllGather**(In[B, $D_Y$]) *(this can be skipped by sharing with (1) from the forward pass)*
6. $dW_{in}$[D, $F_Y$] = dTmp[B, $F_Y$] *$_B$ In[B, D]
7. dIn[B, D] {$U_Y$} = dTmp[B, $F_Y$] *$_F$ $W_{in}$[D, $F_Y$] *(needed for previous layers)*
8. dIn[B, $D_Y$] = **ReduceScatter**(dIn[B, D] {$U_Y$}) *(on critical path)*

One nice thing about tensor parallelism is that it interacts nicely with the two matrices in our Transformer forward pass. Naively, we would do an AllReduce after each of the two matrices. But here we first do **In[B, $D_Y$] * $W_{in}$[D, $F_Y$] -> Tmp[B, $F_Y$]** and then **Tmp[B, $F_Y$] * $W_{out}$[$F_Y$, D] -> Out[B, $D_Y$]**. This means we AllGather **In** at the beginning, and ReduceScatter **Out** at the end, rather than doing an AllReduce.

**How costly is this?** Let's only model the forward pass - the backwards pass is just the transpose of each operation here. In 1D tensor parallelism we AllGather the activations before the first matmul, and ReduceScatter them after the second, sending two bytes at a time (bf16). Let's figure out when we're bottlenecked by communication.

$$T_{\text{math}} = \frac{4 \cdot B \cdot D \cdot F}{Y \cdot C} \tag{4}$$

$$T_{\text{comms}} = \frac{2 \cdot 2 \cdot (B \cdot D)}{W_{\text{ici}}} \tag{5}$$

$$\text{T} \approx \max\left(\frac{4 \cdot B \cdot D \cdot F}{Y \cdot C}, \frac{2 \cdot 2 \cdot (B \cdot D)}{W_{\text{ici}}}\right) \tag{6}$$

Noting that we want compute cost to be greater than comms cost, we get:

$$\frac{4 \cdot B \cdot D \cdot F}{Y \cdot C} > \frac{2 \cdot 2 \cdot (B \cdot D)}{W_{\text{ici}}} \tag{7}$$

$$\frac{F}{Y \cdot C} > \frac{1}{W_{\text{ici}}} \tag{8}$$

$$F > Y \cdot \frac{C}{W_{\text{ici}}} \tag{9}$$

Thus for instance, for TPUv5p, $C/W_{ici} = 2550$ in bf16, so we can only do tensor parallelism up to $Y < F/2550$. When we have multiple ICI axes, our $T_{\text{comms}}$ is reduced by a factor of $M_Y$, so we get $Y < M_Y \cdot F/2550$.

**Takeaway**: Tensor Parallelism becomes communication bound when $Y > M_Y \cdot F/2550$. For most models this is between 8 and 16-way tensor parallelism.

**Note that this doesn't depend on the precision of the computation**, since e.g. for int8, on TPUv5p, $C_{\text{int8}}/W_{ici}$ is $5100$ instead of $2550$ but the comms volume is also halved, so the two factors of two cancel.

**Let's think about some examples:**

- On TPUv5p with LLaMA 3-70B with $D = 8192, F \approx 30,000$, we can comfortably do 8-way tensor parallelism, but will be communication bound on 16 way tensor parallelism. The required F for 8-way model sharding is 20k.
- For Gemma 7B, $F \approx 50k$, so we become communication bound with 19-way tensor parallelism. That means we could likely do 16-way and still see good performance.

## Combining FSDP and Tensor Parallelism

**Syntax:** $\text{In}[B_X, D_Y] \cdot_D W_{\text{in}}[D_X, F_Y] \cdot_F W_{\text{out}}[F_Y, D_X] \to \text{Out}[B_X, D_Y]$

The nice thing about FSDP and tensor parallelism is that they can be combined. By sharding $\mathbf{W_{in}}$ and $\mathbf{W_{out}}$ along both axes we both save memory and compute. Because we shard B along X, we reduce the size of the model-parallel AllGathers, and because we shard F along Y, we reduce the communication overhead of FSDP. This means a combination of the two can get us to an even lower effective batch size than we saw above.

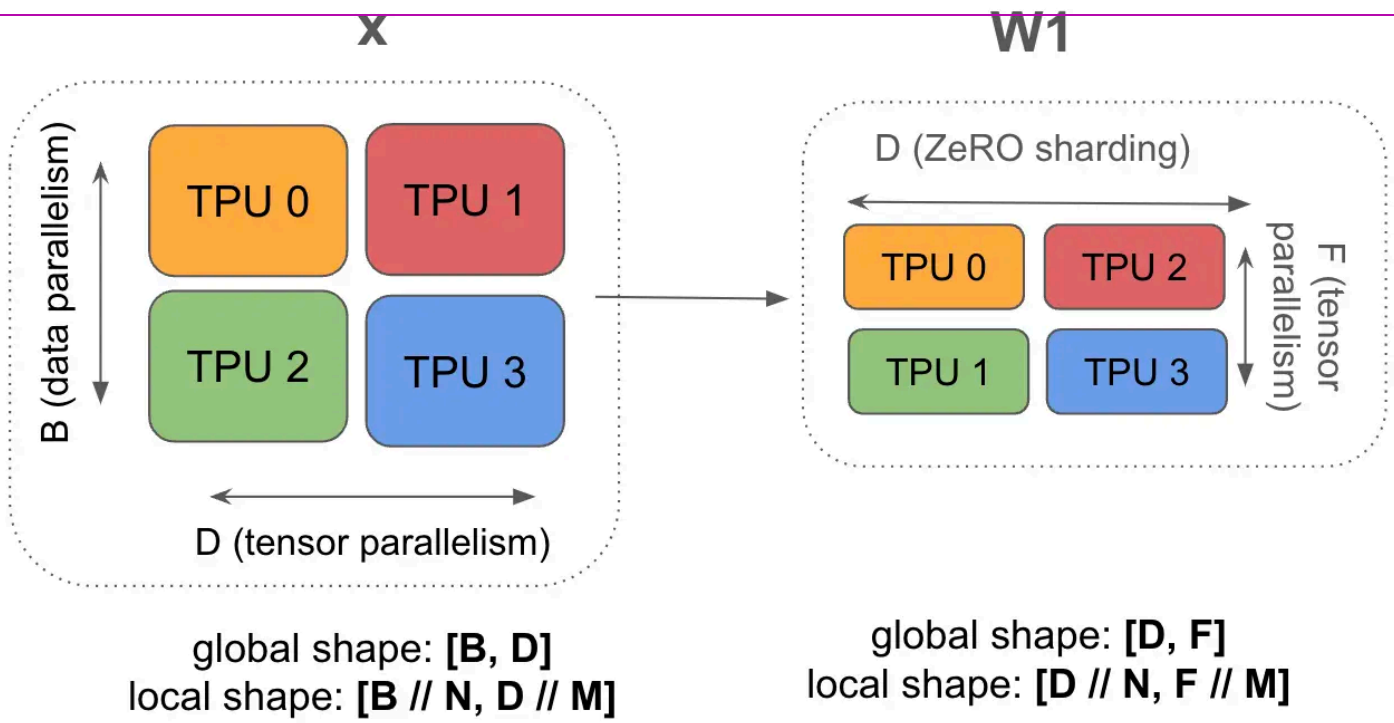



**Figure:** a diagram combining FSDP and tensor parallelism. Unlike the other cases, there is no duplication of model parameters.

▼ Here's the full algorithm for mixed FSDP + tensor parallelism. While we have a lot of communication, all our AllGathers and ReduceScatters are smaller because we have batch-sharded our activations and tensor sharded our weights much more!

**Forward pass:** need to compute Loss[B]

1. $In[B_X, D] = \textbf{AllGather}_Y(In[B_X, D_Y])$ *(on critical path)*
2. $W_{in}[D, F_Y] = \textbf{AllGather}_X(W_{in}[D_X, F_Y])$ *(can be done ahead of time)*
3. $Tmp[B_X, F_Y] = In[B_X, D] *_D W_{in}[D, F_Y]$
4. $W_{out}[F_Y, D] = \textbf{AllGather}_X(W_{out}[F_Y, D_X])$ *(can be done ahead of time)*
5. $Out[B_X, D]\ \{U_Y\} = Tmp[B_X, F_Y] *_F W_{out}[F_Y, D]$
6. $Out[B_X, D_Y] = \textbf{ReduceScatter}_Y(Out[B_X, D]\ \{U_Y\})$ *(on critical path)*
7. $Loss[B_X] = \ldots$

**Backward pass:** need to compute $dW_{out}[F_Y, D_X]$, $dW_{in}[D_X, F_Y]$

1. $dOut[B_X, D_Y] = \ldots$
2. $dOut[B_X, D] = \textbf{AllGather}_Y(dOut[B_X, D_Y])$ *(on critical path)*
3. $dW_{out}[F_Y, D]\ \{U.X\} = Tmp[B_X, F_Y] *_B dOut[B_X, D]$
4. $dW_{out}[F_Y, D_X] = \textbf{ReduceScatter}_X(dW_{out}[F_Y, D]\ \{U.X\})$
5. $W_{out}[F_Y, D] = \textbf{AllGather}_X(W_{out}[F_Y, D_X])$ *(can be done ahead of time)*
6. $dTmp[B_X, F_Y] = dOut[B_X, D] *_D W_{out}[F_Y, D]$ *(can throw away dOut[B, D] here)*
7. $In[B_X, D] = \textbf{AllGather}_Y(In[B_X, D_Y])$ *(not on critical path + this can be shared with (2) from the previous layer)*
8. $dW_{in}[D, F_Y]\ \{U.X\} = dTmp[B_X, F_Y] *_B In[B_X, D]$
9. $dW_{in}[D_X, F_Y] = \textbf{ReduceScatter}_X(dW_{in}[D, F_Y]\ \{U.X\})$
10. $W_{in}[D, F_Y] = \textbf{AllGather}_X(W_{in}[D_X, F_Y])$ *(can be done ahead of time)*
11. $dIn[B_X, D]\ \{U_Y\} = dTmp[B_X, F_Y] *_F W_{in}[D, F_Y]$ *(needed for previous layers)*
12. $dIn[B_X, D_Y] = \textbf{ReduceScatter}_Y(dIn[B_X, D]\ \{U_Y\})$ *(on critical path)*

**What's the right combination of FSDP and TP?** A simple but key maxim is that FSDP moves weights and tensor parallelism moves activations. That means as our batch size shrinks (especially as we do more data parallelism), tensor parallelism becomes cheaper because our activations per-shard are smaller.

- Tensor parallelism performs $\textbf{AllGather}_Y([B_X, D_Y])$ which shrinks as $X$ grows.

- FSDP performs $\mathbf{AllGather}_X([D_X, F_Y])$ which shrinks as $Y$ grows.

Thus by combining both we can push our minimum batch size per replica down even more. We can calculate the optimal amount of FSDP and TP in the same way as above:

Let $X$ be the number of chips dedicated to FSDP and $Y$ be the number of chips dedicated to tensor parallelism. Let $N$ be the total number of chips in our slice with $N = XY$. Let $M_X$ and $M_Y$ be the number of mesh axes over which we do FSDP and TP respectively (these should roughly sum to 3). We'll purely model the forward pass since it has the most communication per FLOP. Then adding up the comms in the algorithm above, we have

$$T_{\text{FSDP comms}}(B, X, Y) = \frac{2 \cdot 2 \cdot D \cdot F}{Y \cdot W_{\text{ici}} \cdot M_X}$$

$$T_{\text{TP comms}}(B, X, Y) = \frac{2 \cdot 2 \cdot B \cdot D}{X \cdot W_{\text{ici}} \cdot M_Y}$$

And likewise our total FLOPs time is

$$T_{\text{math}} = \frac{2 \cdot 2 \cdot B \cdot D \cdot F}{N \cdot C}.$$

To simplify the analysis, we make two assumptions: first, we allow $X$ and $Y$ to take on non-integer values (as long as they are positive and satisfy $XY = N$ ); second, we assume that we can fully overlap comms on the $X$ and $Y$ axis with each other. Under the second assumption, the total comms time is

$$T_{\text{comms}} = \max\left(T_{\text{FSDP comms}}, T_{\text{TP comms}}\right)$$

Before we ask under what conditions we'll be compute-bound, let's find the optimal values for $X$ and $Y$ to minimize our total communication. Since our FLOPs is independent of $X$ and $Y$, the optimal settings are those that simply minimize comms. To do this, let's write $T_{\text{comms}}$ above in terms of $X$ and $N$ (which is held fixed, as it's the number of chips in our system) rather than $X$ and $Y$:

$$T_{\text{comms}}(X) = \frac{4D}{W_{\text{ici}}} \max\left(\frac{F \cdot X}{N \cdot M_X}, \frac{B}{X \cdot M_Y}\right)$$

Because $T_{\text{FSDP comms}}$ is monotonically increasing in $X$, and $T_{\text{TP comms}}$ is monotonically decreasing in $X$, the maximum must be minimized when $T_{\text{FSDP comms}} = T_{\text{TP comms}}$, which occurs when

$$\frac{FX_{opt}}{M_X} = \frac{BN}{X_{opt}M_Y} \rightarrow$$

$$X_{opt} = \sqrt{\frac{B}{F}\frac{M_X}{M_Y}N}$$

This is super useful! This tells us, for a given $B$, $F$, and $N$, what amount of FSDP is optimal. Let's get a sense of scale. Plugging in realistic values, namely $N = 64$ (corresponding to a 4x4x4 array of chips), $B = 48,000$, $F = 32768$, gives roughly $X \approx 13.9$. So we would choose $X$ to be 16 and $Y$ to be 4, close to our calculated optimum.

**Takeaway:** in general, during training, the optimal amount of FSDP is $X_{opt} = \sqrt{\frac{B}{F}\frac{M_X}{M_Y}N}$.

Now let's return to the question we've been asking of all our parallelism strategies: **under what conditions will we be compute-bound?** Since we can overlap FLOPs and comms, we are compute-bound when

$$\max\left(T_{\text{FSDP comms}}, T_{\text{TP comms}}\right) < T_{\text{math}}$$

By letting $\alpha \equiv C/W_{\text{ici}}$, the ICI arithmetic intensity, we can simplify:

$$\max\left(\frac{F}{Y \cdot M_X}, \frac{B}{X \cdot M_Y}\right) < \frac{B \cdot F}{N \cdot \alpha}$$

Since we calculated $X_{opt}$ to make the LHS maximum equal, we can just plug it into either side (noting that $Y_{opt} = N/X_{opt}$), i.e.

$$\frac{F}{N \cdot W_{\text{ici}} \cdot M_X}\sqrt{\frac{B}{F}\frac{M_X}{M_Y}N} < \frac{B \cdot F}{N \cdot C}$$

Further simplifying, we find that

$$\sqrt{\frac{B \cdot F}{M_X \cdot M_Y \cdot N}} < \frac{B \cdot F}{N \cdot \alpha},$$

where the left-hand-side is proportional to the communication time and the right-hand-side is proportional to the computation time. Note that while the computation time scales linearly with the batch size (as it does regardless of parallelism), the communication time scales as the square root of the batch size. The ratio of the computation to communication time thus also scales as the square of the batch size:

$$\frac{T_{\text{math}}}{T_{\text{comms}}} = \frac{\sqrt{BF}\sqrt{M_X M_Y}}{\alpha\sqrt{N}}.$$

To ensure that this ratio is greater than one so we are compute bound, we require

$$\frac{B}{N} > \frac{\alpha^2}{M_X M_Y F}$$

To get approximate numbers, again plug in $F = 32,768$, $\alpha = 2550$, and $M_X M_Y = 2$ (as it must be for a 3D mesh). This gives roughly $B/N > 99$. This roughly wins us a factor of eight compared to the purely data parallel (or FSDP) case, where assuming a 3D mesh we calculate that $B/N$ must exceed about $850$ to be compute bound.

**Takeaway:** combining tensor parallelism with FSDP allows us to drop to a $B/N$ of $2550^2/2F$. This lets us handle a batch of as little as 100 per chip,

which is roughly a factor of eight smaller than we could achieve with just FSDP.

Below we plot the ratio of FLOPs to comms time for mixed FSDP + TP, comparing it both to only tensor parallelism (TP) and only data parallelism (FSDP), on a representative 4x4x4 chip array. While pure FSDP parallelism dominates for very large batch sizes, in the regime where batch size over number of chips is between roughly 100 and 850, a mixed FSDP + TP strategy is required in order to be compute-bound.

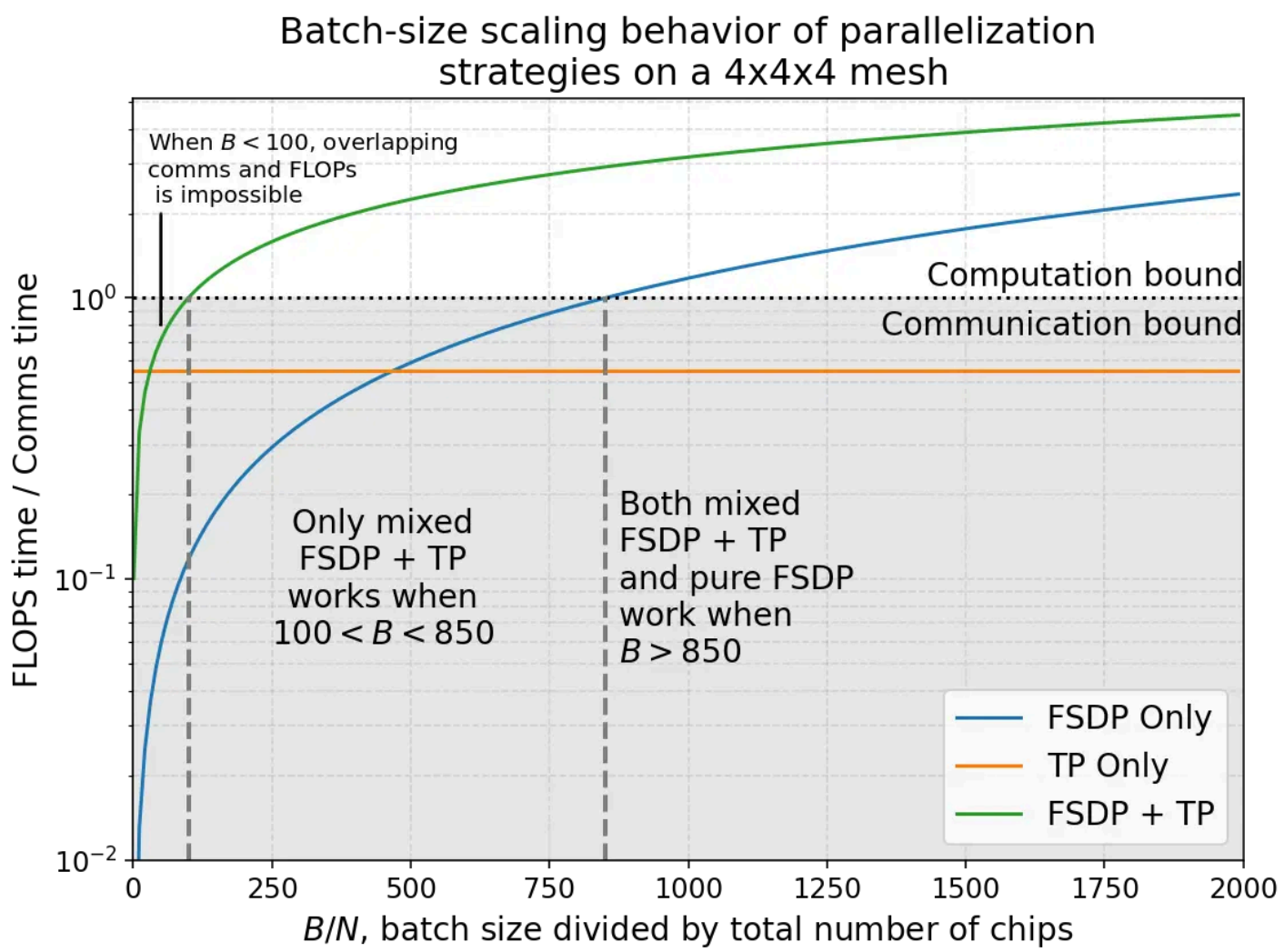



Figure: ratio of FLOPs to comms time for optimal mixed FSDP/TP on a TPUv5p 4x4x4 slice with F=30k. As expected, tensor parallelism has a fixed ratio with batch size; ideal mixed FSDP + TP scales with $\sqrt{B}$, and FSDP scales with $B$. However, in intermediate batch size regimes, only FSDP + TP achieves a ratio greater than unity.

Here's another example of TPU v5p 16x16x16 showing the FLOPs and comms time as a function of batch size for different sharding schemes.

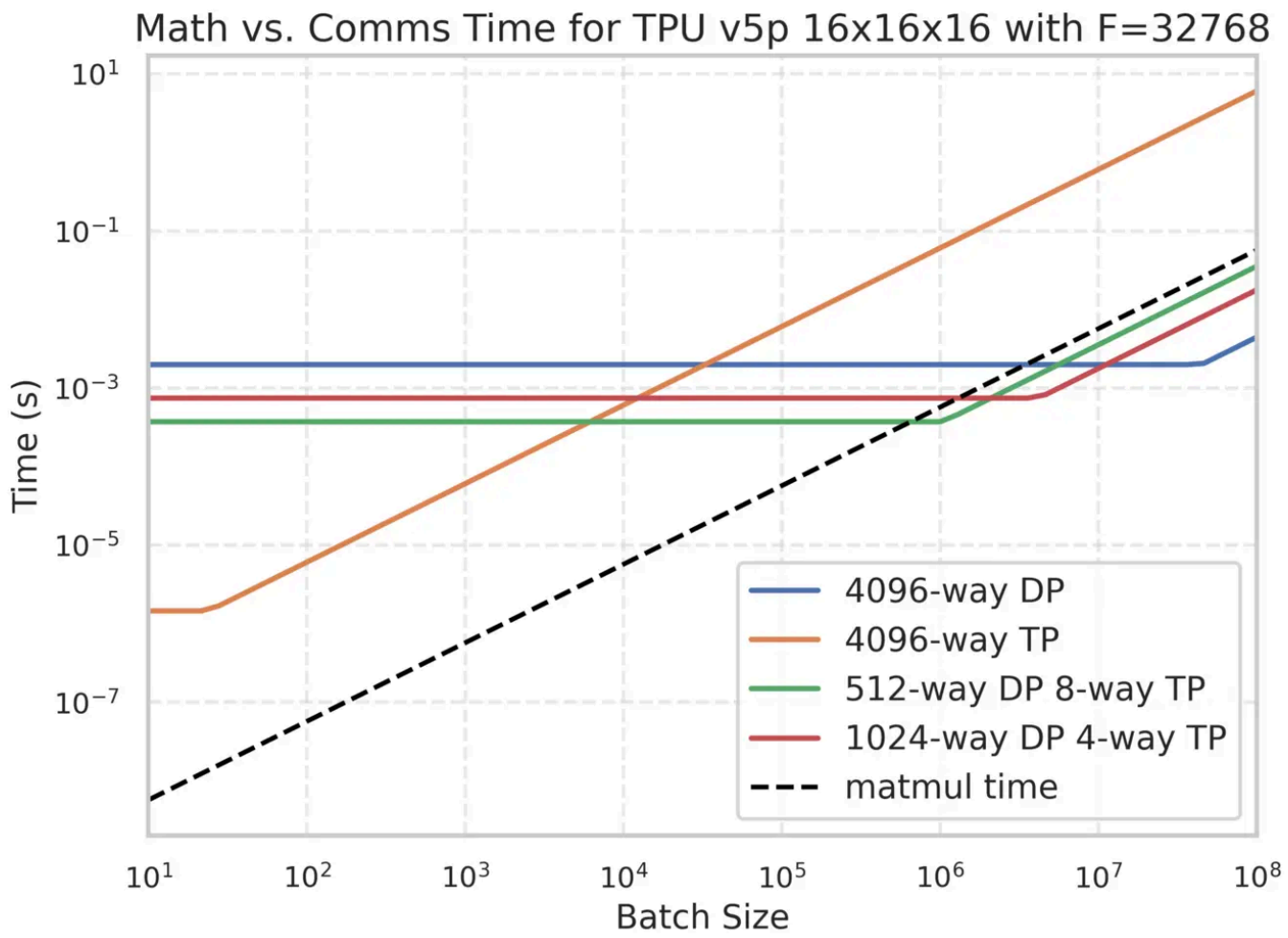
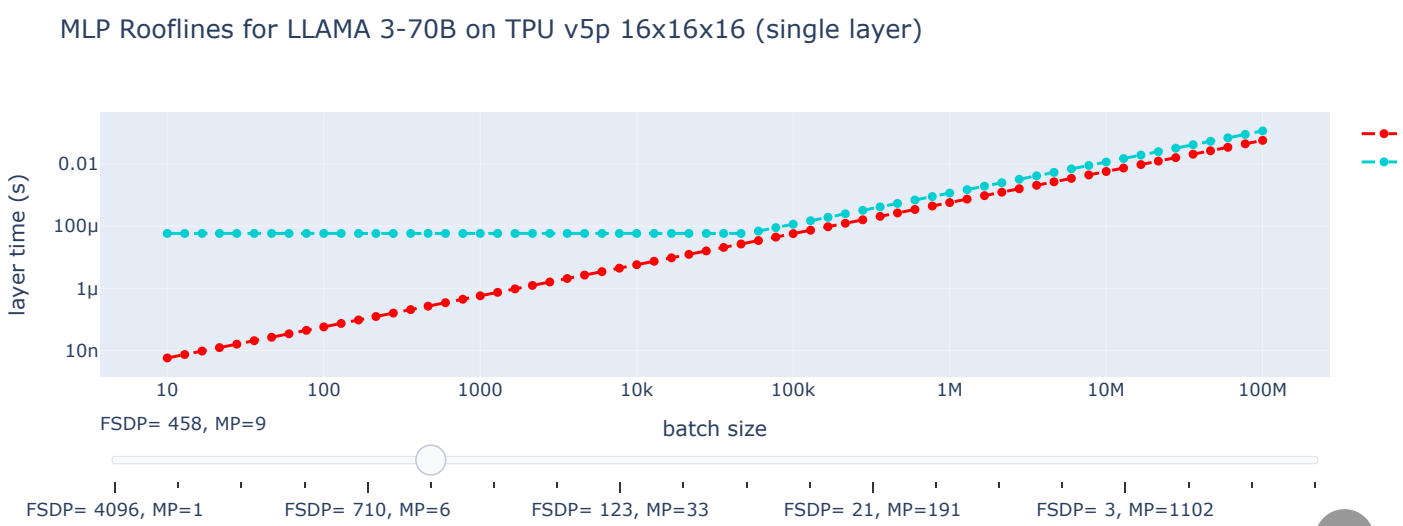



**Figure:** time taken for communication with different parallelism schemes. The black dashed line is the time taken by the matrix multiplication FLOPs, so any curve above this line is comms-bound. We note that all strategies become comms-bound below batch size 6e5, which is in line with our expected 4096 * 2550^2 / (2 * 8192 * 4) = 4e5.

The black curve is the amount of time spent on model FLOPs, meaning any batch size where this is lower than all comms costs is strictly comms bound. You'll notice the black curve intersects the green curve at about `4e5`, as predicted.

Here's an interactive animation to play with this, showing the total compute time and communication time for different batch sizes:



You'll notice this generally agrees with the above (minimum around FSDP=256, TP=16), plus or minus some wiggle factor for some slight differences in the number of axes for each.

## Pipelining

You'll probably notice we've avoided talking about pipelining at all in the previous sections. Pipelining is a dominant strategy for GPU parallelism that is somewhat less essential on TPUs. Briefly, pipelined training involves splitting the layers of a model across multiple devices and passing the activations between pipeline stages during the forward and backward pass. The algorithm is something like:

1. Initialize your data on TPU 0 with your weights sharded across the layer dimension ($W_{\text{in}}[L_Z, D_X, F_Y]$ for pipelining with FSDP and tensor parallelism).
2. Perform the first layer on TPU 0, then copy the resulting activations to TPU 1, and repeat until you get to the last TPU.
3. Compute the loss function and its derivative $\partial L / \partial x_L$.
4. For the last pipeline stage, compute the derivatives $\partial L / \partial W_L$ and $\partial L / \partial x_{L-1}$, then copy $\partial L / \partial x_{L-1}$ to the previous pipeline stage and repeat until you reach TPU 0.

▼ Here is some (working) Python pseudo-code

This pseudocode should run on a Cloud TPU VM. While it's not very efficient or realistic, it gives you a sense how data is being propagated across devices.

```
batch_size = 32
d_model = 128
d_ff = 4 * d_model

num_layers = len(jax.devices())

key = jax.random.PRNGKey(0)

# Pretend each layer is just a single matmul.
x = jax.random.normal(key, (batch_size, d_model))
weights = jax.random.normal(key, (num_layers, d_model, d_model))

def layer_fn(x, weight):
  return x @ weight

# Assume we have num_layers == num_pipeline_stages
intermediates = [x]
for i in range(num_layers):
  x = layer_fn(x, weights[i])
  intermediates.append(x)

  if i != num_layers - 1:
    x = jax.device_put(x, jax.devices()[i+1])

def loss_fn(batch):
  return jnp.mean(batch ** 2)  # make up some fake loss function

loss, dx = jax.value_and_grad(loss_fn)(x)

for i in range(num_layers - 1, -1, -1):
  _, f_vjp = jax.vjp(layer_fn, intermediates[i], weights[i])
  dx, dw = f_vjp(dx)  # compute the jvp dx @ J(L)(x[i], W[i])
  weights[i] = weights[i] - 0.01 * dw  # update our weights

  if i != 0:
    dx = jax.device_put(dx, jax.devices()[i-1])
```

**Why is this a good idea?** Pipelining is great for many reasons: it has a low communication cost between pipeline stages, meaning you can train very large models even with low bandwidth interconnects. This is often very useful on GPUs since they are not densely connected by ICI in the way TPUs are.

**Why is this difficult/annoying?** You might have noticed in the pseudocode above that TPU 0 is almost always idle! It's only doing work on the very first and last step of the pipeline. The period of idleness is called a pipeline bubble and is very annoying to deal with. Typically we try to mitigate this first with microbatching, which sends multiple small batches through the pipeline, keeping TPU 0 utilized for at least a larger fraction of the total step time.

A second approach is to carefully overlap the forward matmul $W_i @ x_i$, the backward $dx$ matmul $W_i @ \partial L/\partial x_{i+1}$, and the $dW$ matmul $\partial L/\partial x_{i+1} @ x_i$. Since each of these requires some FLOPs, we can overlap them to fully hide the bubble. Here's a plot from the recent DeepSeek v3 paper [3] showing their "bubble-free" pipeline schedule:


**Figure:** the DeepSeek v3 pipeline schedule (from their recent paper). Orange is the forward matmul, green is the dL/dx matmul, and blue is the dL/dW matmul. By prioritizing the backwards dL/dx multiplications, we can avoid "stranding" FLOPs.

Because it is less critical for TPUs (which have larger interconnected pods), we won't delve into this as deeply, but it's a good exercise to understand the key pipelining bottlenecks.

### Scaling Across Pods

The largest possible TPU slice is a TPU v5p SuperPod with 8960 chips (and 2240 hosts). When we want to scale beyond this size, we need to cross the Data-Center Networking (DCN) boundary. Each TPU host comes equipped with one or several NICs (Network Interface Cards) that connect the host to other TPU v5p pods over Ethernet. As noted in the TPU Section, each host has about 200Gbps (25GB/s) of full-duplex DCN bandwidth, which is about 6.25GB/s full-duplex (egress) bandwidth per TPU.

Typically, when scaling beyond a single pod, we do some form of model parallelism or FSDP within the ICI domain, and then pure data parallelism across multiple pods. Let $N$ be the number of TPUs we want to scale to and $M$ be the number of TPUs per ICI-connected slice. To do an AllReduce over DCN, we can do a ring-reduction over the set of pods, giving us (in the backward pass):

$$T_{\text{math}} = \frac{2 \cdot 2 \cdot 2 \cdot BDF}{N \cdot C}$$

$$T_{\text{comms}} = \frac{2 \cdot 2 \cdot 2 \cdot DF}{M \cdot W_{\text{dcn}}}$$

The comms bandwidth scales with $M$, since unlike ICI the total bandwidth grows as we grow our ICI domain and acquire more NICs. Simplifying, we find that $T_{\text{math}} > T_{\text{comms}}$ when

$$\frac{B}{\text{slice}} > \frac{C}{W_{\text{dcn}}}$$

For TPU v5p, the $\frac{C}{W_{\text{dcn}}}$ is about `4.46e14 / 6.25e9 = 71,360`. This tells us that to efficiently scale over DCN, there is a minimum batch size per ICI domain needed to egress each node.

**How much of a problem is this?** To take a specific example, say we want to train LLaMA-3 70B on TPU v5p with a BS of 2M tokens. LLaMA-3 70B has $F \approx 30,000$. From the above sections, we know the following:

- We can do Tensor Parallelism up to $Y = M_Y \cdot F/2550 \approxeq 11 \cdot M_Y$.
- We can do FSDP so long as $B/N > 2550/M_X$. That means if we want to train with BS=2M and 3 axes of data parallelism, we'd at most be able to use $\approx 2400$ chips, roughly a quarter of a TPU v5p pod.
- When we combine FSDP + Tensor Parallelism, become comms-bound when we have $B/N < 2550^2/(2 \cdot 30000) = 108$, so this lets us scale to roughly 18k chips! However, the maximum size of a TPU v5p pod is 8k chips, so beyond that we have to use DCN.

The TLDR is that we have a nice recipe for training with BS=1M, using roughly X (FSDP) = 1024 and Y (TP) = 8, but with BS=2M we need to use DCN. As noted above, we have a DCN arithmetic intensity of $71,360$, so we just need to make sure our batch size per ICI domain is greater than this. This is trivial for us, since with 2 pods we'd have a per-pod BS of 1M, and a per TPU batch size of 111, which is great (maybe cutting it a bit close, but theoretically sound).

**Takeaway:** Scaling across multiple TPU pods is fairly straightforward using pure data parallelism so long as our per-pod batch size is at least 71k tokens.

## Takeaways from LLM Training on TPUs

- Increasing parallelism or reducing batch size both tend to make us more communication-bound because they reduce the amount of compute performed

per chip.

- Up to a reasonable context length (~32k) we can get away with modeling a Transformer as a stack of MLP blocks and define each of several parallelism schemes by how they shard the two/three main matmuls per layer.
- During training there are 4 main parallelism schemes we consider, each of which has its own bandwidth and compute requirements (data parallelism, FSDP, tensor parallelism).

| Strategy | Description |
|---|---|
| **Data Parallelism** | Activations are batch sharded, everything else is fully-replicated, we all-reduce gradients during the backward pass. |
| **FSDP** | Activations, weights, and optimizer are batch sharded, weights are gathered just before use, gradients are reduce-scattered. |
| **Tensor Parallelism (aka Megatron, Model)** | Activations are sharded along $d_{\text{model}}$, weights are sharded along $d_{ff}$, activations are gathered before $W_{in}$, the result reduce-scattered after $W_{out}$. |
| **Mixed FSDP + Tensor Parallelism** | Both of the above, where FSDP gathers the model sharded weights. |

And here are the "formulas" for each method:

| Strategy | Formula |
|---|---|
| DP | $\text{In}[B_X, D] \cdot_D W_{\text{in}}[D, F] \cdot_F W_{\text{out}}[F, D] \to \text{Out}[B_X, D]$ |
| FSDP | $\text{In}[B_X, D] \cdot_D W_{\text{in}}[D_X, F] \cdot_F W_{\text{out}}[F, D_X] \to \text{Out}[B_X, D]$ |
| TP | $\text{In}[B, D_Y] \cdot_D W_{\text{in}}[D, F_Y] \cdot_F W_{\text{out}}[F_Y, D] \to \text{Out}[B, D_Y]$ |
| TP + FSDP | $\text{In}[B_X, D_Y] \cdot_D W_{\text{in}}[D_X, F_Y] \cdot_F W_{\text{out}}[F_Y, D_X] \to \text{Out}[B_X, D_Y]$ |

- Each of these strategies has a limit at which it becomes network/communication bound, based on their per-device compute and comms. Here's compute and comms per-layer, assuming $X$ is FSDP and $Y$ is tensor parallelism.

| Strategy | Compute per layer (ignoring gating einsum) | Comms per layer (bytes, forward + backward pass) |
|---|---|---|
| DP | $4BDF/X + 8BDF/X$ | $0 + 8DF$ |
| FSDP | $4BDF/X + 8BDF/X$ | $4DF + 8DF$ |
| TP | $4BDF/Y + 8BDF/Y$ | $4BD + 4BD$ |
| FSDP + TP | $4BDF/(XY) + 8BDF/(XY)$ | $(4BD/X + 4DF/Y) + (8BD/X + 8DF/Y)$ |

- Pure data parallelism is rarely useful because the model and its optimizer state use bytes = 10x parameter count. This means we can rarely fit more than a few billion parameters in memory.
- Data parallelism and FSDP become comms bound when the $\text{batch size per shard} < C/W$, the arithmetic intensity of the network. For ICI this is 2,550 and for DCN this is about 71,000. This can be increased with more parallel axes.
- Tensor parallelism becomes comms bound when $|Y| > F/2550$. **This is around 8-16 way for most models.** This is independent of the batch size.
- Mixed FSDP + tensor parallelism allows us to drop the batch size to as low as $2550^2/2F \approx 100$. This is remarkably low.
- Data parallelism across pods requires a minimum batch size per pod of roughly 71,000 before becoming DCN-bound.
- Basically, if your batch sizes are big or your model is small, things are simple. You can either do data parallelism or FSDP + data parallelism across DCN. The middle section is where things get interesting.

## Some Problems to Work

Let's use LLaMA-2 13B as a basic model for this section. Here are the model details:

| hyperparam | value |
|---|---|
| L | 40 |
| D | 5,120 |
| F | 13824 |
| N | 40 |
| K | 40 |

| hyperparam | value |
|---|---|
| H | 128 |
| V | 32,000 |

LLaMA-2 has separate embedding and output matrices and a gated MLP block.

**Question 1:** How many parameters does LLaMA-2 13B have (I know that's silly but do the math)? *Note that, as in Transformer Math, LLaMA-3 has 3 big FFW matrices, two up-projection and one down-projection. We ignored the two "gating" einsum matrices in this section, but they behave the same as $W_{in}$ in this section.*

▶ Click here for the answer.

**Question 2:** Let's assume we're training with BS=16M tokens and using Adam. Ignoring parallelism for a moment, how much total memory is used by the model's parameters, optimizer state, and activations? *Assume we store the parameters in bf16 and the optimizer state in fp32 and checkpoint activations three times per layer (after the three big matmuls).*

▶ Click here for the answer.

**Question 3:** Assume we want to train with 32k sequence length and a total batch size of 3M tokens on a TPUv5p 16x16x16 slice. Assume we want to use bfloat16 weights and a float32 optimizer, as above.

1. Can we use pure data parallelism? Why or why not?
2. Can we use pure FSDP? Why or why not? With pure FSDP, how much memory will be used per device (assume we do gradient checkpointing only after the 3 big FFW matrices).
3. Can we use mixed FSDP + tensor parallelism? Why or why not? If so, what should $X$ and $Y$ be? How much memory will be stored per device? Using only roofline FLOPs estimates and ignoring attention, how long will each training step take at 40% MFU?

▶ Click here for the answer.

**That's it for Part 5! For Part 6, which applies this content to real LLaMA models, click here!**

# Appendix

### Appendix A: Deriving the backward pass comms

Above, we simplified the Transformer layer forward pass as Out[B, D] = In[B, D] $*_D$ $W_{in}$[D, F] $*_F$ $W_{out}$[F, D]. How do we derive the comms necessary for the backwards pass?

This follows fairly naturally from the rule in the previous section for a single matmul **Y = X * A**:

$$\frac{dL}{dA} = \frac{dL}{dY}\frac{dY}{dA} = X^T\left(\frac{dL}{dY}\right)$$

$$\frac{dL}{dX} = \frac{dL}{dY}\frac{dY}{dX} = \left(\frac{dL}{dY}\right)A^T$$

Using this, we get the following formulas (letting Tmp[B, F] stand for In[B, D] * $W_{in}$[D, F]):

1. $dW_{out}$[F, D] = Tmp[B, F] $*_B$ dOut[B, D]
2. dTmp[B, F] = dOut[B, D] $*_D$ $W_{out}$[F, D]
3. $dW_{in}$[D, F] = In[B, D] $*_B$ dTmp[B, F]
4. dIn[B, D] = dTmp[B, F] $*_F$ $W_{in}$[D, F]

Note that these formulas are mathematical statements, with no mention of sharding. The job of the backwards pass is to compute these four quantities. So to figure out the comms necessary, we just take the shardings of all the quantities which are to be matmulled in the four equations above (Tmp, dOut, $W_{out}$, $W_{in}$), which are specified by our parallelization scheme, and use the rules of sharded matmuls to figure out what comms we have to do. Note that dOut is sharded in the same way as Out.

## Footnotes

1. We'll focus on communication bounds — since while memory capacity constraints are important, they typically do not bound us when using rematerialization (activation checkpointing) and a very large number of chips during pre-training. We also do not discuss expert parallelism here for MoEs — which expands the design space substantially, only the base case of a dense Transformer. [↩]

2. Adam stores parameters, first order and second order accumulators. Since the params are in bfloat16 and optimizer state is in float32, this gives us `2 + 8 = 10` bytes per parameters. [↩]

3. Note that this doesn't include gradient checkpoints, so this wouldn't actually be useful. This is an absolute lower bound with a batch of 1 token. [↩]

4. We assume this partitioning is done over an ICI mesh, so the relevant network bandwidth is $W_{\text{ici}}$ [↩]

5. Technically, FSDP adds communication in the forward pass that pure DP doesn't have, but this is in the same proportion as the backward pass so it should have no effect on the comms roofline. The key here is that ZeRO-3 turns a backward-pass AllReduce into an AllGather and a ReduceScatter, which have the same total comms volume. [↩]

## References


1. **ZeRO: Memory optimizations toward training Trillion parameter models**
   Rajbhandari, S., Rasley, J., Ruwase, O. and He, Y., 2019. arXiv [cs.LG].

2. **Megatron-LM: Training multi-billion parameter language models using model parallelism**
   Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J. and Catanzaro, B., 2019. arXiv [cs.CL].

3. **DeepSeek-V3 Technical Report**
   {DeepSeek-AI},, Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., Dai, D., Guo, D., Yang, D., Chen, D., Ji, D., Li, E., Lin, F., Dai, F., Luo, F., Hao, G., Chen, G., Li, G., Zhang, H., Bao, H., Xu, H., Wang, H., Zhang, H., Ding, H., Xin, H., Gao, H., Li, H., Qu, H., Cai, J.L., Liang, J., Guo, J., Ni, J., Li, J., Wang, J., Chen, J., Chen, J., Yuan, J., Qiu, J., Li, J., Song, J., Dong, K., Hu, K., Gao, K., Guan, K., Huang, K., Yu, K., Wang, L., Zhang, L., Xu, L., Xia, L., Zhao, L., Wang, L., Zhang, L., Li, M., Wang, M., Zhang, M., Zhang, M., Tang, M., Li, M., Tian, N., Huang, P., Wang, P., Zhang, P., Wang, Q., Zhu, Q., Chen, Q., Du, Q., Chen, R.J., Jin, R.L., Ge, R., Zhang, R., Pan, R., Wang, R., Xu, R., Zhang, R., Chen, R., Li, S.S., Lu, S., Zhou, S., Chen, S., Wu, S., Ye, S., Ye, S., Ma, S., Wang, S., Zhou, S., Yu, S., Zhou, S., Pan, S., Wang, T., Yun, T., Pei, T., Sun, T., Xiao, W.L., Zeng, W., Zhao, W., An, W., Liu, W., Liang, W., Gao, W., Yu, W., Zhang, W., Li, X.Q., Jin, X., Wang, X., Bi, X., Liu, X., Wang, X., Shen, X., Chen, X., Zhang, X., Chen, X., Nie, X., Sun, X., Wang, X., Cheng, X., Liu, X., Xie, X., Liu, X., Yu, X., Song, X., Shan, X., Zhou, X., Yang, X., Li, X., Su, X., Lin, X., Li, Y.K., Wang, Y.Q., Wei, Y.X., Zhu, Y.X., Zhang, Y., Xu, Y., Xu, Y., Huang, Y., Li, Y., Zhao, Y., Sun, Y., Li, Y., Wang, Y., Yu, Y., Zheng, Y., Zhang, Y., Shi, Y., Xiong, Y., He, Y., Tang, Y., Piao, Y., Wang, Y., Tan, Y., Ma, Y., Liu, Y., Guo, Y., Wu, Y., Ou, Y., Zhu, Y., Wang, Y., Gong, Y., Zou, Y., He, Y., Zha, Y., Xiong, Y., Ma, Y., Yan, Y., Luo, Y., You, Y., Liu, Y., Zhou, Y., Wu, Z.F., Ren, Z.Z., Ren, Z., Sha, Z., Fu, Z., Xu, Z., Huang, Z., Zhang, Z., Xie, Z., Zhang, Z., Hao, Z., Gou, Z., Ma, Z., Yan, Z., Shao, Z., Xu, Z., Wu, Z., Zhang, Z., Li, Z., Gu, Z., Zhu, Z., Liu, Z., Li, Z., Xie, Z., Song, Z., Gao, Z. and Pan, Z., 2024. arXiv [cs.CL].


## 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}
}
```