Sharded Matrices and How to Multiply Them

Part 3 of How To Scale Your Model (Part 2: TPUs | Part 4: Transformer Math)

When we train large ML models, we have to split (or "shard") their parameters or inputs across many accelerators. Since LLMs are mostly made up of matrix multiplications, understanding this boils down to understanding how to multiply matrices when they're split across devices. We develop a simple theory of sharded matrix multiplication based on the cost of TPU communication primitives.

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

Partitioning Notation and Collective Operations

When we train an LLM on ten thousand TPUs or GPUs, we're still doing abstractly the same computation as when we're training on one. The difference is that **our arrays don't fit in the HBM of a single TPU/GPU**, so we have to split them.[1] We call this "*sharding*" or "*partitioning*" our arrays. The art of scaling is figuring out how to shard our models so computation remains efficient.

Here's an example 2D array **A** sharded across 4 TPUs:

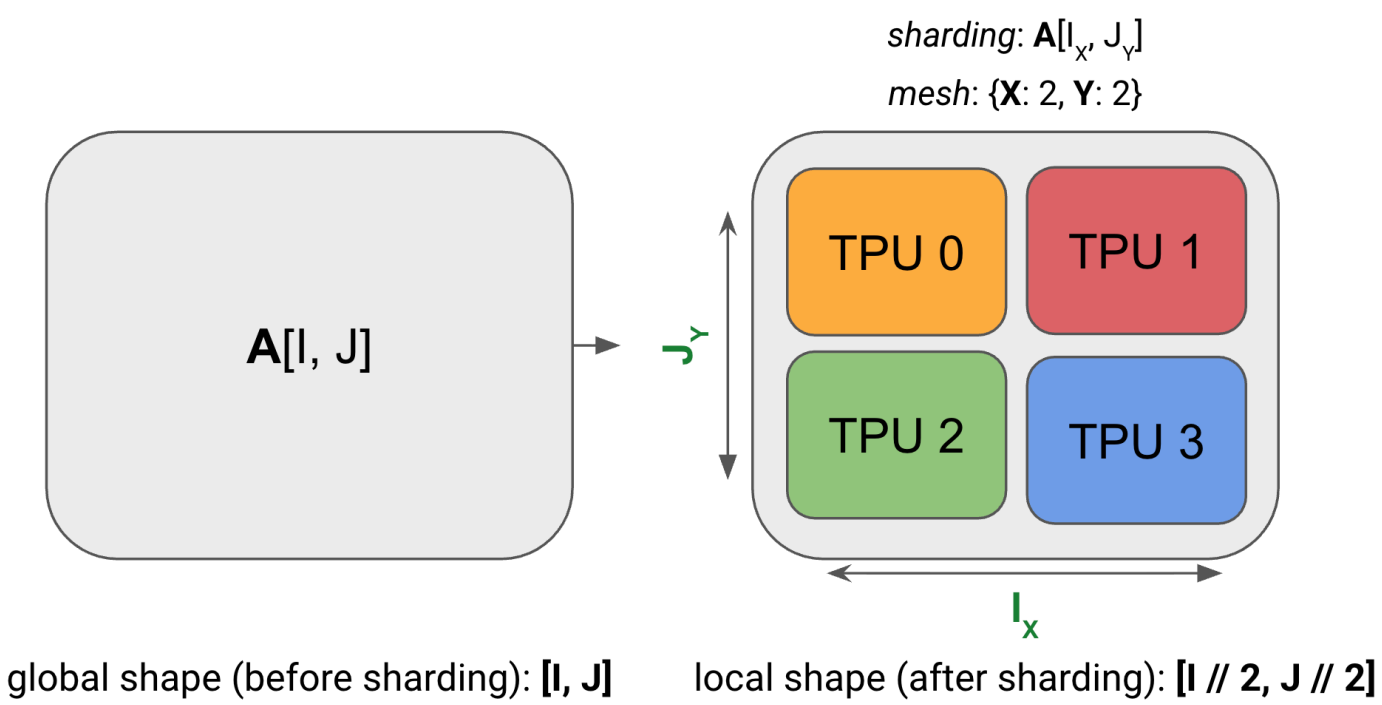


Figure: an example array of shape A[I, J] gets sharded across 4 devices. Both dimensions are evenly sharded across 2 devices with a sharding $A[I_X, J_Y]$. Each TPU holds 1/4 of the total memory.

Note how the sharded array still has the same *global* or *logical shape* as unsharded array, say `(4, 128)`, but it also has a *device local shape*, like `(2, 64)`, which gives us the actual size in bytes that each TPU is holding (in the figure above, each TPU holds ¼ of the total array). Now we'll generalize this to arbitrary arrays.

A unified notation for sharding

We use a variant of *named-axis notation* to describe *how* the tensor is sharded in blocks across the devices: we assume the existence of a 2D or 3D grid of devices called the **device mesh** where each axis has been given **mesh axis names e.g. X, Y, and Z.** We can then specify how the matrix data is laid out across the device mesh by describing how each named dimension of the array is partitioned across the physical mesh axes. We call this assignment a **sharding**.

Example (the diagram above): For the above diagram, we have:

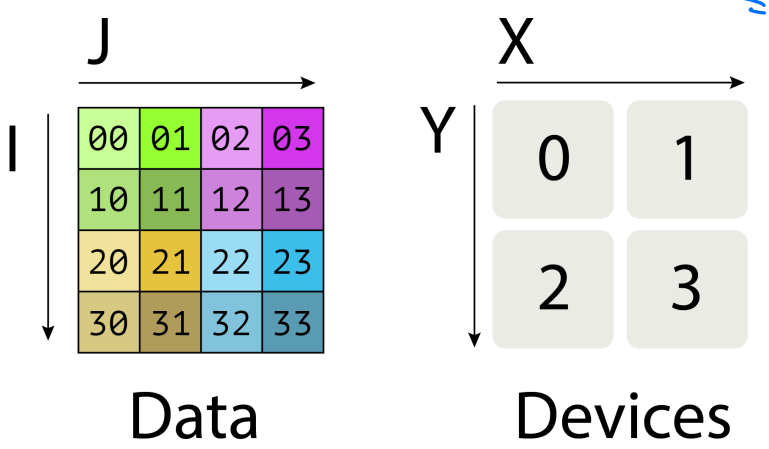
- **Mesh:** the device mesh above `Mesh(devices=((0, 1), (2, 3)), axis_names=('X', 'Y'))`, which tells us we have 4 TPUs in a 2x2 grid, with axis names X and Y.
- **Sharding:** $A[I_X, J_Y]$, which tells us to shard the first axis, I, along the mesh axis X, and the second axis, J, along the mesh axis Y. This sharding tells us that each shard holds $1/(|X| \cdot |Y|)$ of the array.

Taken together, we know that the local shape of the array (the size of the shard that an individual device holds) is $(|I|/2, |J|/2)$, where $|I|$ is the size of A's first dimension and $|J|$ is the size of A's second dimension.

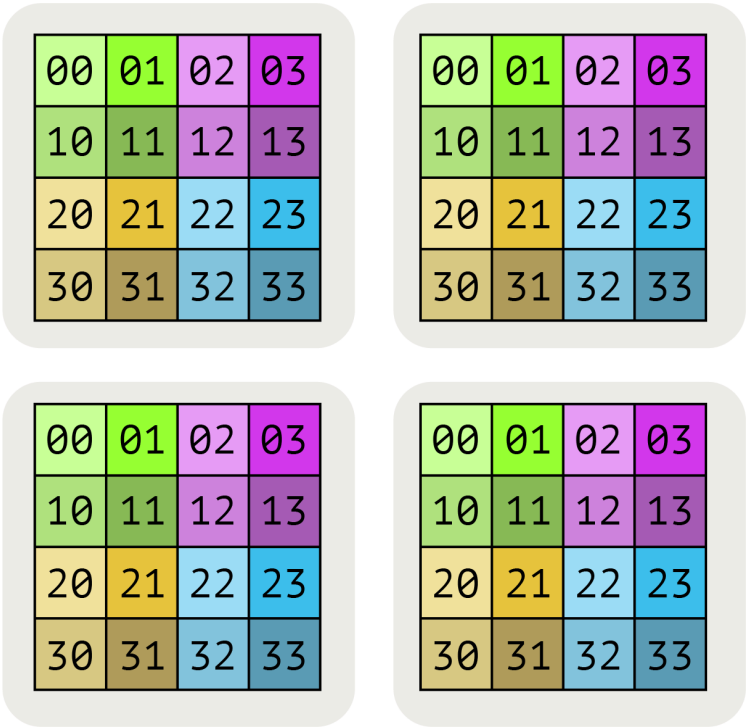
Pop Quiz [2D sharding across 1 axis]: Consider an array `fp32[1024, 4096]` with sharding $A[I_{XY}, J]$ and mesh `{'X': 8, 'Y': 2}`. How much data is held by each device? How much time would it take to load this array from HBM on H100s (assuming `3.4e12` memory bandwidth per chip)?

▶ Click here for the answer.

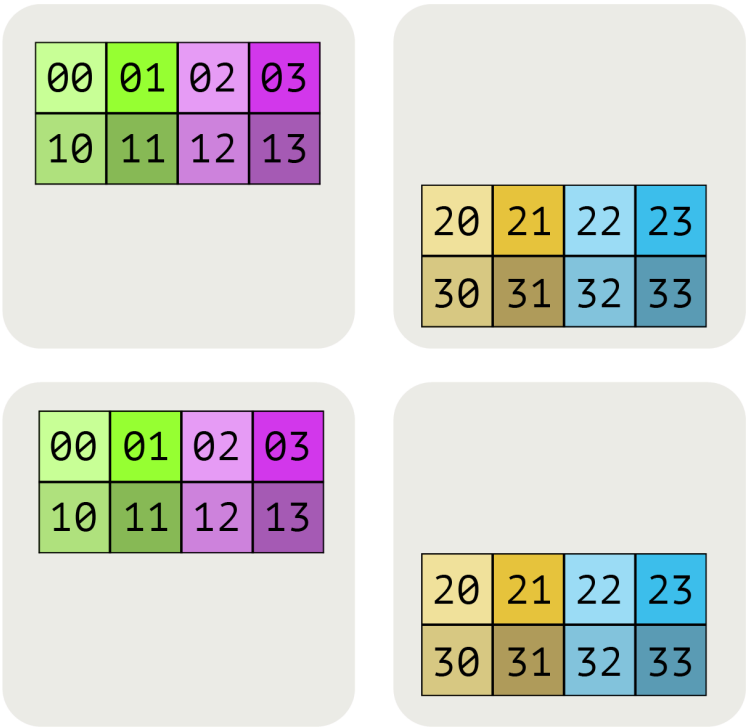
Visualizing these shardings: Let's try to visualize these shardings by looking at a 2D array of data split over 4 devices:



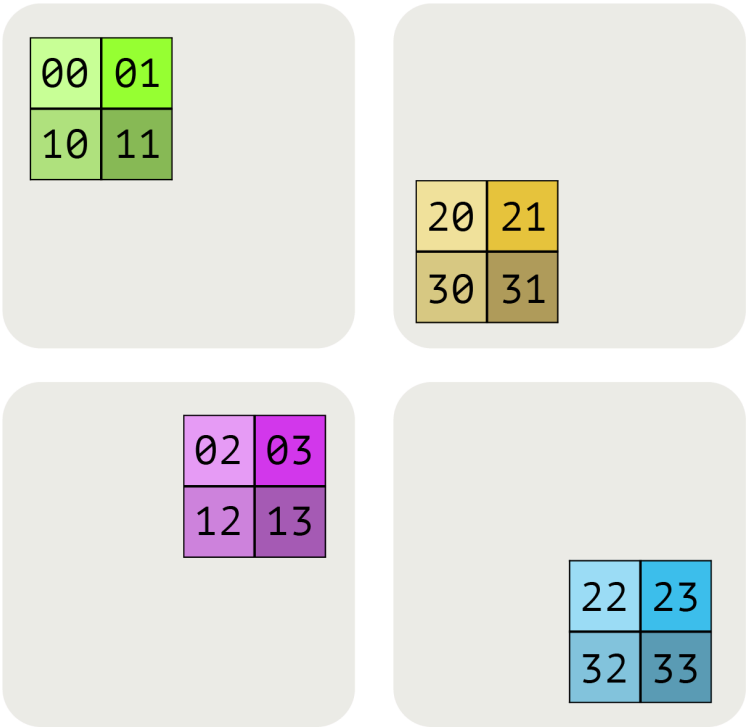
We write the *fully-replicated* form of the matrix simply as $A[I, J]$ with no sharding assignment. This means that *each* device contains a full copy of the entire matrix.



We can indicate that one of these dimensions has been partitioned across a mesh axis with a subscript mesh axis. For instance $A[I_X, J]$ would mean that the **I** logical axis has been partitioned across the **X** mesh dimension, but that the **J** dimension is *not* partitioned, and the blocks remain *partially-replicated* across the **Y** mesh axis.

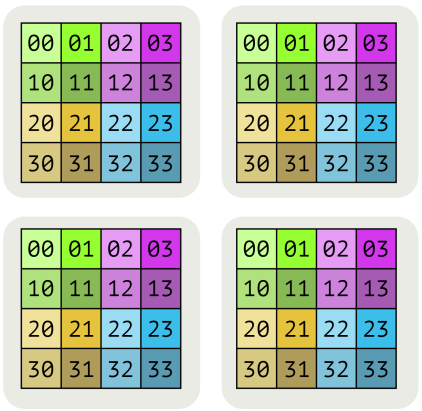


$A[I_X, J_Y]$ means that the **I** logical axis has been partitioned across the **X** mesh axis, and that the **J** dimension has been partitioned across the **Y** mesh axis.

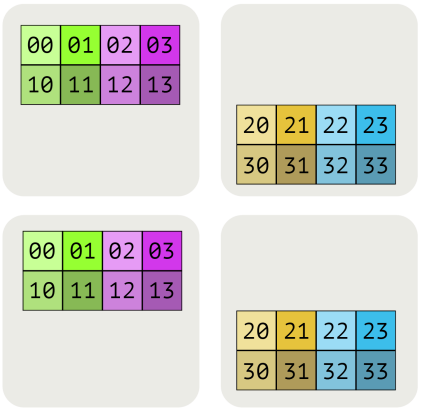


We illustrate the other possibilities in the figure below:

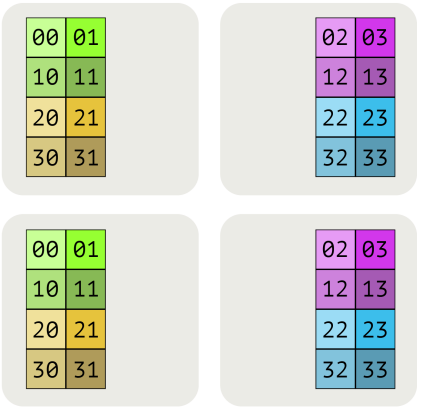
I, J



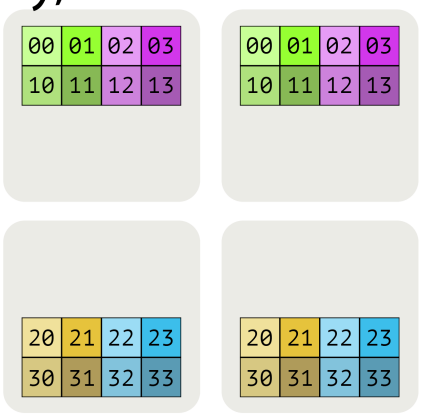
Ix, J



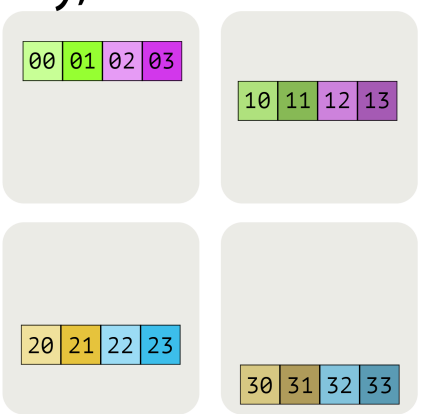
I, Jx



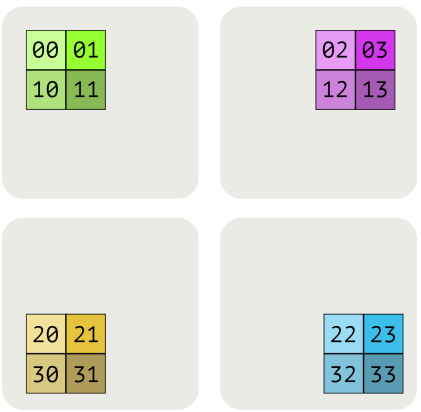
Iy, J



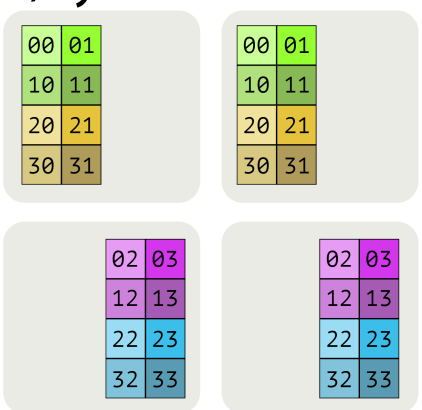
Ixy, J



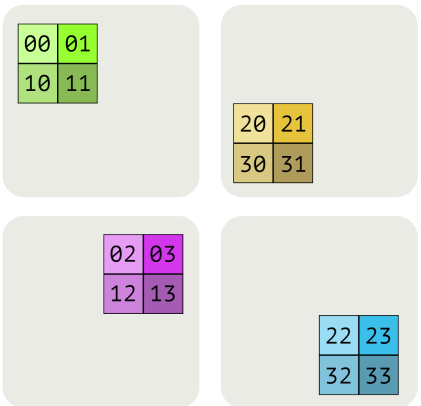
Iy, Jx



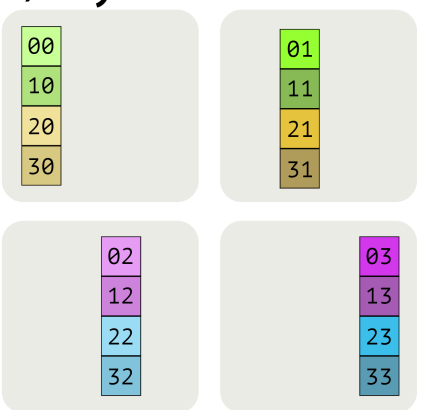
I, Jy



Ix, Jy

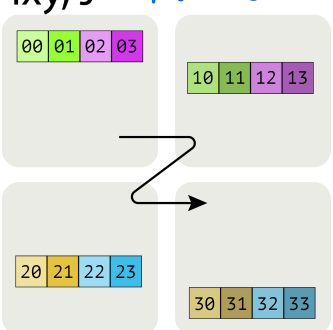


I, Jxy

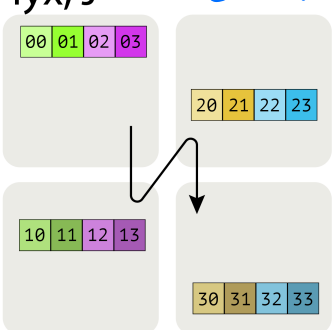


Here $A[I_{XY}, J]$ means that we treat the **X** and **Y** mesh axes as a larger flattened dimension and partition the **I** named axis across all the devices. The order of the multiple mesh-axis subscripts matters, as it specifies the traversal order of the partitioning across the grid.

Ixy, J row order



Iyx, J column order



Lastly, note that we *cannot* have multiple named axes sharded along the *same* mesh dimension. e.g. $A[I_X, J_X]$ is a nonsensical, forbidden sharding. Once a mesh dimension has been used to shard one dimension of an array, it is in a sense "spent".

Pop Quiz: Let **A** be an array with shape `int8[128, 2048]`, sharding $A[I_{XY}, J]$, and mesh `Mesh({'X': 2, 'Y': 8, 'Z': 2})` (so 32 devices total). How much memory does **A** use per device? How much total memory does **A** use across all devices?

▶ Click here for the answer.

1 byte × 128×2048 = $2^7 \times 2^{11} = 2^{18} = 0.25$ MiB

I is sharded across (X, Y) but (Z) is replicated

How do we describe this in code?

So far we've avoided talking about code, but now is a good chance for a sneak peek. JAX uses a named sharding syntax that very closely matches the abstract syntax we describe above. We'll talk more about this in Section 10, but here's a quick preview. You can play with this in a Google Colab here and profile the result to see how JAX handles different shardings. This snippet does 3 things:

1. Creates a **jax.Mesh** that maps our 8 TPUs into a 4x2 grid with names 'X' and 'Y' assigned to the two axes.
2. Creates matrices A and B where A is sharded along both its dimensions and B is sharded along the output dimension.
3. Compiles and performs a simple matrix multiplication that returns a sharded array.

So:
Each device = int8[128,2048]
= 0.25 MiB / 16
= 0.015 MiB = 15KB

All int8[8,2048]
× 2 × 8 × 2
= 0.25 MiB × 2
= 0.5 MiB

```
import jax
import jax.numpy as jnp

# Create our mesh! We're running on a TPU v2-8 4x2 slice with names 'X' and 'Y'.
assert len(jax.devices()) == 8
mesh = jax.make_mesh(axis_shapes=(4, 2), axis_names=('X', 'Y'))

# A little utility function to help define our sharding. A PartitionSpec is our
# sharding (a mapping from axes to names).
def P(*args):
  return jax.NamedSharding(mesh, jax.sharding.PartitionSpec(*args))

# We shard both A and B over the non-contracting dimension and A over the contracting dim.
A = jnp.zeros((8, 2048), dtype=jnp.bfloat16, device=P('X', 'Y'))
B = jnp.zeros((2048, 8192), dtype=jnp.bfloat16, device=P(None, 'Y'))

# We can perform a matmul on these sharded arrays! out_shardings tells us how we want
# the output to be sharded. JAX/XLA handles the rest of the sharding for us.
y = jax.jit(lambda A, B: jnp.einsum('BD,DF->BF', A, B), out_shardings=P('X', 'Y'))(A, B)
```

replicate this dimension

The cool thing about JAX is that these arrays behave as if they're unsharded! `B.shape` will tell us the global or logical shape (2048, 8192). We have to actually look at `B.addressable_shards` to see how it's locally sharded. We can perform operations on these arrays and JAX will attempt to figure out how to broadcast or reshape them to perform the operations. For instance, in the above example, the local shape of **A** is `[2, 1024]` and for **B** is `[2048, 4096]`. JAX/XLA will automatically add communication across these arrays as necessary to perform the final multiplication.

Computation With Sharded Arrays

If you have an array of data that's distributed across many devices and wish to perform mathematical operations on it, what are the overheads associated with sharding both the data and the computation?

Obviously, this depends on the computation involved.

- For *elementwise* operations, there is **no overhead** for operating on a distributed array.
- When we wish to perform operations across elements resident on many devices, things get complicated. Thankfully, for most machine learning nearly all computation takes place in the form of matrix multiplications, and they are relatively simple to analyze.

The rest of this section will deal with how to multiply sharded matrices. To a first approximation, this involves moving chunks of a matrix around so you can fully multiply or sum each chunk. **Each sharding will involve different communication.** For example, $A[I_X, J] \cdot B[J, K_Y] \to C[I_X, K_Y]$ can be multiplied without any communication because the *contracting dimension* (J, the one we're actually summing over) is unsharded. However, if we wanted the output unsharded (i.e. $A[I_X, J] \cdot B[J, K_Y] \to C[I, K]$), we would need to copy A or C to every device (using an *AllGather*). These two choices have different communication costs, so we need to calculate this cost and pick the lowest one.

▶ You can think of this in terms of "block matrix multiplication".

Conveniently, we can boil down all possible shardings into roughly 4 cases we need to consider, each of which has a rule for what communication we need to add

1. Case 1: neither input is sharded along the contracting dimension. *We can multiply local shards without any communication.*
2. Case 2: one input has a sharded contracting dimension. *We typically "AllGather" the sharded input along the contracting dimension.*
3. Case 3: both inputs are sharded along the contracting dimension. *We can multiply the local shards, then "AllReduce" the result.*
4. Case 4: both inputs have a non-contracting dimension sharded along the same axis. We cannot proceed without AllGathering one of the two inputs first.

You can think of these as rules that simply need to be followed, but it's also valuable to understand why these rules hold and how expensive they are. We'll go through each one of these in detail now.

Case 1: neither multiplicand has a sharded contracting dimension

Lemma: when multiplying sharded matrices, the computation is valid and the output follows the sharding of the inputs *unless* the contracting dimension is sharded or both matrices are sharded along the same axis. For example, this works fine

$$\mathbf{A}[I_X, J] \cdot \mathbf{B}[J, K_Y] \to \mathbf{C}[I_X, K_Y]$$

with no communication whatsoever, and results in a tensor sharded across both the X and Y hardware dimensions. Try to think about why this is. Basically, the computation is *independent* of the sharding, since each batch entry has some local chunk of the axis being contracted that it can multiply and reduce. Any of these cases work fine and follow this rule:

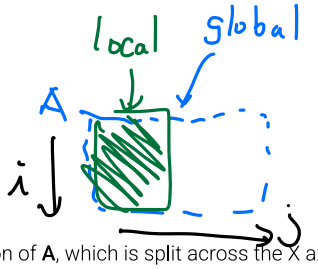
$$\begin{aligned}
\mathbf{A}[I, J] \cdot \mathbf{B}[J, K] &\to \mathbf{C}[I, K] \\
\mathbf{A}[I_X, J] \cdot \mathbf{B}[J, K] &\to \mathbf{C}[I_X, K] \\
\mathbf{A}[I, J] \cdot \mathbf{B}[J, K_Y] &\to \mathbf{C}[I, K_Y] \\
\mathbf{A}[I_X, J] \cdot \mathbf{B}[J, K_Y] &\to \mathbf{C}[I_X, K_Y]
\end{aligned}$$

Because neither **A** nor **B** has a sharded contracting dimension **J**, we can simply perform the local block matrix multiplies of the inputs and the results will *already* be sharded according to the desired output shardings. When both multiplicands have non-contracting dimensions sharded along the same axis, this is no longer true (see the invalid shardings section for details).

Case 2: one multiplicand has a sharded contracting dimension

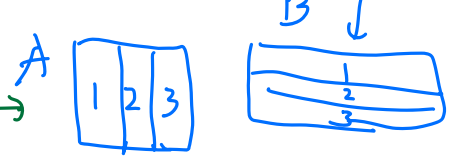
Let's consider what to do when one input **A** is sharded along the contracting **J** dimension and **B** is fully replicated:

$$\mathbf{A}[I, J_X] \cdot \mathbf{B}[J, K] \to \mathbf{C}[I, K]$$



We cannot simply multiply the local chunks of **A** and **B** because we need to sum over the full contracting dimension of **A**, which is split across the X axis. Typically, we first "**AllGather**" the shards of **A** so every device has a full copy, and only then multiply against **B:**

$$\begin{aligned}
\mathbf{AllGather}_X[I, J_X] &\to \mathbf{A}[I, J] \\
\mathbf{A}[I, J] \cdot \mathbf{B}[J, K] &\to \mathbf{C}[I, K]
\end{aligned}$$

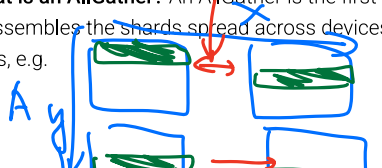


This way the actual multiplication can be done fully on each device.

Takeaway: When multiply matrices where one of the matrices is sharded along the contracting dimension, we general AllGather it first so the contraction is no longer sharded, then do a local matmul.

Note that when **B** is not also sharded along X, we could also do the local partial matmul and then sum (or *AllReduce*) the sharded partial sums, which can be faster in some cases. See Question 4 below.

What is an AllGather? An AllGather is the first core MPI communication primitive we will discuss. An AllGather *removes the sharding* along an axis and reassembles the shards spread across devices onto *each* device along that axis. Using the notation above, an AllGather removes a subscript from a set of axes, e.g.



$$\mathbf{AllGather}_{XY}(A[I_{XY}, J]) \to A[I, J]$$

We don't have to remove all subscripts for a given dimension, e.g. $A[I_{XY}, J] \to A[I_Y, J]$ is also an AllGather, just over only a single axis. Also note that we may also wish to use an AllGather to remove *non-contracting* dimension sharding, for instance in the matrix multiply:

$$A[I_X, J] \cdot B[J, K] \to C[I, K]$$

We could either AllGather **A** initially to remove the input sharding, or we can do the sharded matmul and then AllGather the result **C**.

How is an AllGather actually performed? To perform a 1-dimensional AllGather around a single TPU axis (a ring), we basically have each TPU pass its shard around a ring until every device has a copy.[2] Here is an animation:

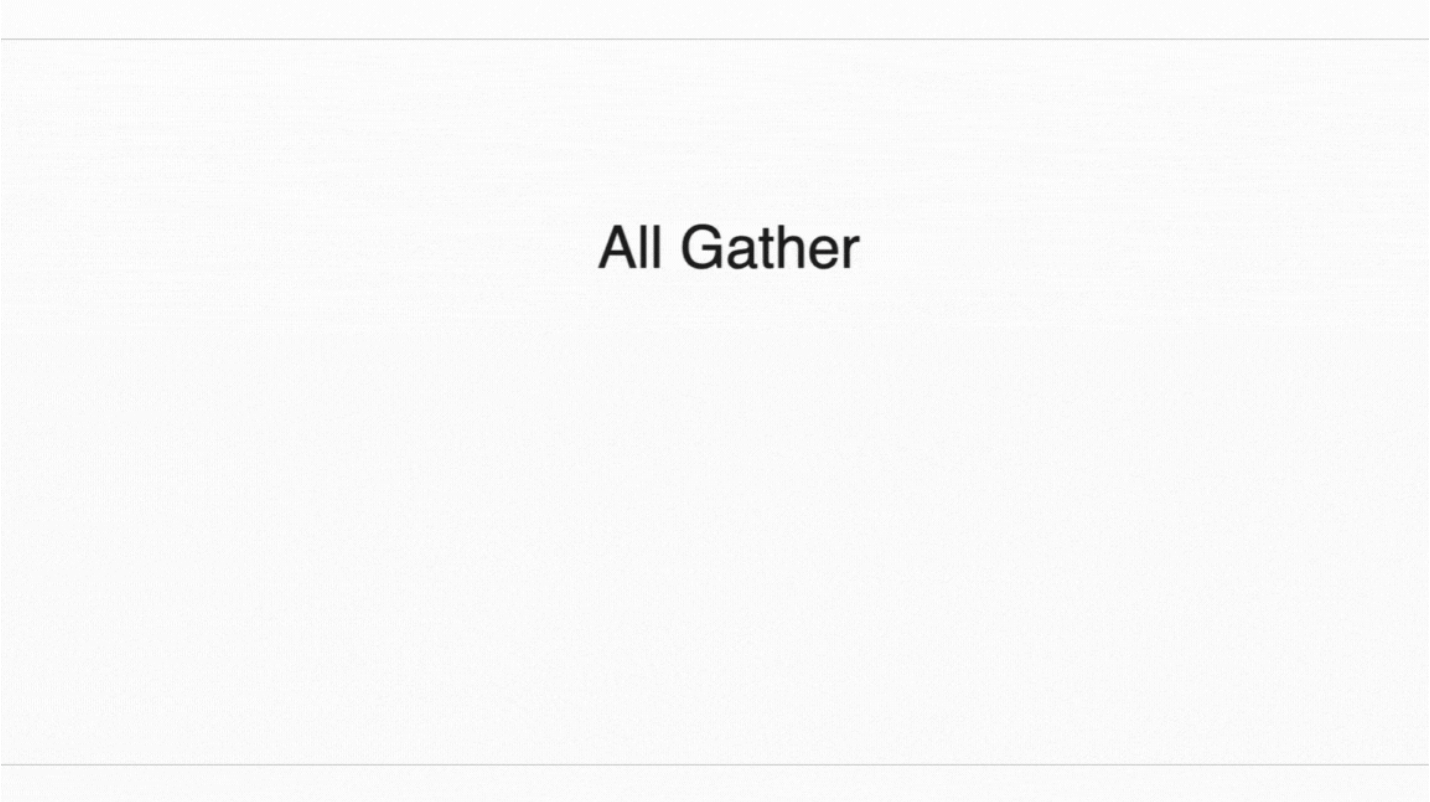


Figure: An animation showing how to perform an AllGather around a set of 8 TPU or GPU devices. Each device starts with 1 / 8th of the array and ends up with a full copy.

N devices

We can either do an AllGather in one direction or both directions (two directions is shown above). If we do one direction, each TPU sends chunks of size bytes/N over $N - 1$ hops around the ring. If we do two directions, we have $\lceil \frac{N}{2} \rceil$ hops of size $2 \cdot \text{bytes}/N$.

How long does this take? Let's take the bidirectional AllGather and calculate how long it takes. Let V be the number of bytes in the array, and X be the number of shards on the contracting dimension. Then from the above diagram, each hop sends $V/|X|$ bytes in each direction, so each hop takes

$$T_{hop} = \frac{2 \cdot V}{X \cdot W_{\text{ici}}}$$

$\left(2 \cdot \frac{V}{X}\right) / (W_{ici})$ (bidirectional)

where W_{ici} is the **bidirectional** ICI bandwidth.[3] We need to send a total of $|X|/2$ hops to reach every TPU[4], so the total reduction takes

$$T_{total} = \frac{2 \cdot V \cdot X}{2 \cdot X \cdot W_{\text{ici}}}$$

$$T_{total} = \frac{V}{W_{\text{ici}}}$$

X = # of devices = # of shards

bytes in the array

Note that this **doesn't depend on X!** That's kind of striking, because it means even though our TPUs are only locally connected, the locality of the connections doesn't matter. We're just bottlenecked by the speed of each link.

Takeaway: when performing an AllGather (or a ReduceScatter or AllReduce) in a throughput-bound regime, the actual communication time depends only on the size of the array and the available bandwidth, not the number of devices over which our array is sharded!

10^{-6} sec

A note on ICI latency: Each hop over an ICI link has some intrinsic overhead regardless of the data volume. This is typically around 1us. This means when our array A is very small and each hop takes less than 1us, we can enter a "latency-bound" regime where the calculation *does* depend on X.

▶ For the full details, click here.

$T_{hop} = \max(10^{-6}\text{sec}, \frac{2V}{X W_{ici}})$

Here is an empirical measurement of AllGather bandwidth on a TPU v5e 8x16 slice. The array is sharded across the 16 axis so it has a full bidirectional ring.

$T_{total} = \max\left(\frac{X}{2} \times 10^{-6}, \frac{V}{W_{ici}}\right)$

Sending $\frac{\text{buffer}}{4.5 \times 10^{10}} \leq 10^{-6} \Rightarrow \text{buffer} \leq \frac{4.5 \times 10^{10} \times 10^{-6}}{4.5 \times 10^{4}}$

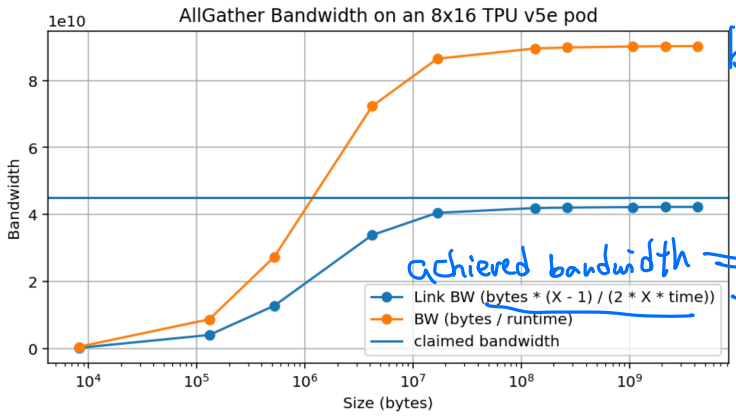


Figure: empirical bandwidth and estimated link bandwidth for TPU v5e during an AllGather. BW in orange is the actual bytes per second AllGathered, while the blue curve shows the empirical unidirectional link bandwidth calculated according to the known cost of the collective.

Note both that we achieve only about 95% of the peak claimed bandwidth (`4.5e10`) and that we achieve this peak at about 10MB, which when 16-way sharded gives us about 500kB per device (*aside: this is much better than GPUs).

What happens when we AllGather over multiple axes? When we gather over multiple axes, we have multiple dimensions of ICI over which to perform the gather. For instance, $\text{AllGather}_{XY}([B, D_{XY}])$ operates over two hardware mesh axes. This increases the available bandwidth by a factor of N_{axes}.

When considering latency, we end up with the general rule:

$$T_{total} = \max\left[\frac{T_{min} \cdot \sum_i |X_i|}{2}, \frac{V}{W_{\text{ici}} \cdot N_{\text{axes}}}\right]$$

where $\sum_i |X_i|/2$ is the length of the longest path in the TPU mesh.

Pop Quiz 2 [AllGather time]: Using the numbers from Part 2, how long does it take to perform the $\text{AllGather}_Y([E_Y, F]) \rightarrow [E, F]$ on a TPUv5e with a 2D mesh `{'X': 8, 'Y': 4}`, $E = 2048$, $F = 8192$ in bfloat16? What about with $E = 256, F = 256$?

▶ Click here for the answer.

Note: when we have a 2D mesh like `{'X': 16, 'Y': 4}`, it is not necessary for each axis to correspond to a specific *hardware* axis. This means for instance the above could describe a 4x4x4 TPU v5p cube with 2 axes on the X axis. This will come into play later when we describe data parallelism over multiple axes.

Case 3: both multiplicands have sharded contracting dimensions

The third fundamental case is when both multiplicands are sharded on their contracting dimensions, along the same mesh axis:

$$\mathbf{A}[I, J_X] \cdot \mathbf{B}[J_X, K] \rightarrow C[I, K]$$

In this case the *local* sharded block matrix multiplies are at least *possible* to perform, since they will share the same sets of contracting indices. But each product will only represent a *partial sum* of the full desired product, and each device along the **X** dimension will be left with different *partial sums* of this final desired product. This is so common that we extend our notation to explicitly mark this condition:

$$\mathbf{A}[I, J_X] \cdot_{\text{LOCAL}} \mathbf{B}[J_X, K] \rightarrow C[I, K]\{U_X\}$$

The notation $\{U_X\}$ reads "**unreduced** along X mesh axis" and refers to this status of the operation being "incomplete" in a sense, in that it will only be finished pending a final sum. The $\cdot_{\text{LOCAL}}$ syntax means we perform the local sum but leave the result unreduced.

This can be seen as the following result about matrix multiplications and outer products:

$$A \cdot B = \sum_{i=1}^{P} \underbrace{A_{:,i} \otimes B_{i,:}}_{\in \mathbb{R}^{n \times m}}$$

where ⊗ is the outer product. Thus, if TPU **i** on axis **X** has the **i**th column of **A**, and the **i**th row of **B**, we can do a local matrix multiplication to obtain $A_{:,i} \otimes B_{i,:} \in \mathbb{R}_{n \times m}$. This matrix has, in each entry, the **i**th term of the sum that **A** • **B** has at that entry. We still need to perform that sum over **P**, which we sharded over mesh axis **X**, to obtain the full **A** • **B**. This works the same way if we write **A** and **B** by blocks (i.e. shards), and then sum over each resulting shard of the result.

We can perform this summation using a full **AllReduce** across the **X** axis to remedy this:

$$A[I, J_X] \cdot_{\text{LOCAL}} B[J_X, K] \to C[I, K]\{U_X\}$$
$$\mathbf{AllReduce}_X C[I, K]\{U_X\} \to C[I, K]$$

AllReduce removes partial sums, resulting in *each* device along the axis having the same fully-summed value. AllReduce is the second of several key communications we'll discuss in this section, the first being the AllGather, and the others being ReduceScatter and AllToAll. An AllReduce takes an array with an unreduced (partially summed) axis and performs the sum by passing those shards around the unreduced axis and accumulating the result. The signature is

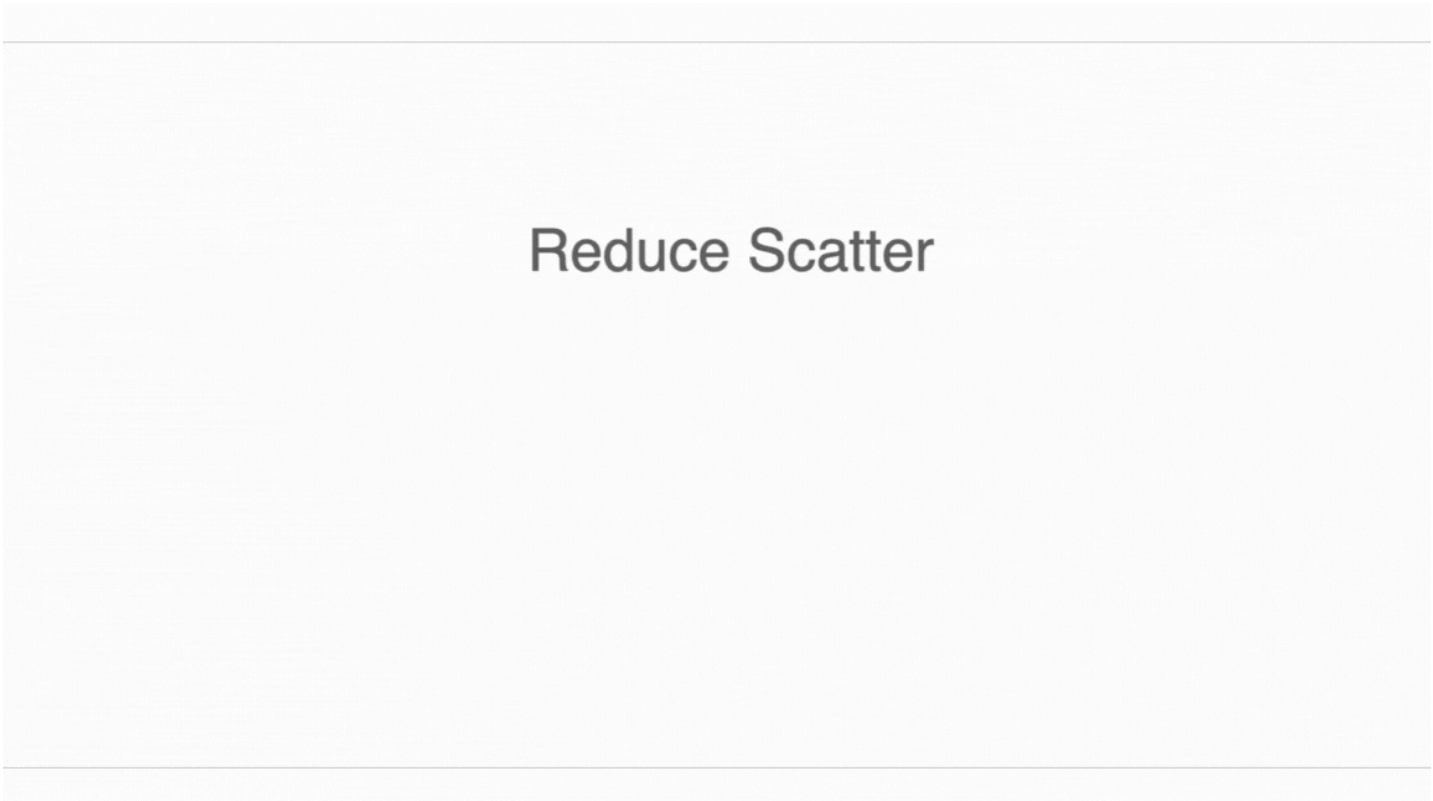
$$\mathbf{AllReduce}_Y A[I_X, J]\{U_Y\} \to A[I_X, J]$$

This means it simply removes the $\{U_Y\}$ suffix but otherwise leaves the result unchanged.

How expensive is an AllReduce? One mental model for how an AllReduce is performed is that every device sends its shard to its neighbors, and sums up all the shards that it receives. Clearly, this is more expensive than an AllGather because each "shard" has the same shape as the full array. Generally, **an AllReduce is twice as expensive as an AllGather.** One way to see this is to note that an **AllReduce** can be expressed as a composition of two other primitives: a **ReduceScatter** and an **AllGather**. Like an AllReduce, a ReduceScatter resolves partial sums on an array but results in an output 'scattered' or partitioned along a given dimension. AllGather collects all those pieces and 'unpartitions/unshards/replicates' the logical axis along that physical axis.

$$\mathbf{ReduceScatter}_{Y,J} : A[I_X, J]\{U_Y\} \to A[I_X, J_Y]$$
$$\mathbf{AllGather}_Y : A[I_X, J_Y] \to A[I_X, J]$$

What about a ReduceScatter? Just as the AllReduce removes a subscript ($F_Y \to F$ above), a ReduceScatter sums an unreduced/partially summed array and then scatters (shards) a different logical axis along the same mesh axis. $[F]\{U_Y\} \to [F_Y]$. The animation shows how this is done: note that it's very similar to an AllGather but instead of retaining each shard, we sum them together. Thus, its latency is roughly the same, excluding the time taken to perform the reduction.



The communication time for each hop is simply the per-shard bytes V/Y divided by the bandwidth W_{ici}, as it was for an AllGather, so we have

$$T_{\text{comms per AllGather or ReduceScatter}} = \frac{V}{W_{\text{ici}}}$$

$$T_{\text{comms per AllReduce}} = 2 \cdot \frac{V}{W_{\text{ici}}}$$

where W_{ici} is the bidirectional bandwidth, so long as we have a full ring to reduce over.

Case 4: both multiplicands have a non-contracting dimension sharded along the same axis

Each mesh dimension can appear at most once when sharding a tensor. Performing the above rules can sometimes lead to a situation where this rule is violated, such as:

$$A[I_X, J] \cdot B[J, K_X] \to C[I_X, K_X]$$

This is invalid because a given shard, say **i**, along dimension **X**, would have the **(i, i)**th shard of **C**, that is, a diagonal entry. There is not enough information among all shards, then, to recover anything but the diagonal entries of the result, so we cannot allow this sharding.

The way to resolve this is to AllGather some of the dimensions. Here we have two choices:

$$\begin{aligned}\mathbf{AllGather}_X A[I_X, J] &\to A[I, J] \\ A[I, J] \cdot B[J, K_X] &\to C[I, K_X]\end{aligned}$$

or

$$\begin{aligned}\mathbf{AllGather}_X B[J, K_X] &\to B[J, K] \\ A[I_X, J] \cdot B[J, K] &\to C[I_X, K]\end{aligned}$$

In either case, the result will only mention **X** once in its shape. Which one we pick will be based on what sharding the following operations need.

A Deeper Dive into TPU Communication Primitives

The previous 4 cases have introduced several "core communication primitives" used to perform sharded matrix multiplications:

1. **AllGather:** removes a subscript from a sharding, gathering the shards.
2. **ReduceScatter:** removes an "un-reduced" suffix from an array by summing shards over that axis, leaving the array sharded over a second axis.
3. **AllReduce:** removes an "un-reduced" suffix, leaving the array unsharded along that axis.

There's one more core communication primitive to mention that arises in the case of Mixture of Experts (MoE) models and other computations: the **AllToAll**.

Our final communication primitive: the AllToAll

A final fundamental collective which does not occur naturally when considering sharded matrix multiplies, but which comes up constantly in practice, is the **AllToAll** collective, or more precisely the special case of a *sharded transposition* or resharding operation. e.g.

$$\mathbf{AllToAll}_{X,J} A[I_X, J] \to A[I, J_X]$$

AllToAlls are typically required to rearrange sharded layouts between different regions of a sharded computation that don't have compatible layout schemes. They arise naturally when considering sharded mixture-of-experts models. *You can think of an AllToAll as moving a subscript from one axis to another.* Because an all to all doesn't need to replicate all of the data of each shard across the ring, it's actually *cheaper* than an AllGather (by a factor of ¼)[5].

If we generalize to an ND AllToAll, the overall cost for an array of V bytes on an AxBxC mesh is

$$T_{\text{comms per AllToAll}} = \frac{V \cdot \max(A, B, C, \dots)}{4 \cdot N \cdot W_{\text{ici}}}$$

where as usual W_{ici} is the bidirectional ICI bandwidth. For a 1D mesh, this reduces to $V/(4 \cdot W_{\text{ici}})$, which is 1 / 4 the cost of an AllReduce. In 2D, the cost actually scales down with the size of the smallest axis.

Aside: If you want a hand-wavy derivation of this fact, start with a 1D torus $\mathbb{Z}/N\mathbb{Z}$. If we pick a source and target node at random, they are on average N / 4 hops from each other, giving us a cost of $(V \cdot N)/(4 * N)$. Now if we consider an ND torus, each axis is basically independent. Each node has $1/Z$ bytes and on average has to hop its data $\max(A, B, C, \dots)/4$ hops.

More about the ReduceScatter

ReduceScatter is a more fundamental operation than it first appears, as it is actually the derivative of an AllGather, and vice versa. i.e. if in the forward pass we have:

$$\mathbf{AllGather}_X A[I_X] \rightarrow A[I]$$

Then we ReduceScatter the reverse-mode derivatives **A'** (which will in general be different on each shard) to derive the sharded **A'**:

$$\mathbf{ReduceScatter}_X A'[I]\{U_X\} \rightarrow A'[I_X]$$

Likewise, $\mathbf{ReduceScatter}_X(A[I]\{U_X\}) \rightarrow A[I_X]$ in the forward pass implies $\mathbf{AllGather}_X(A'[I_X]) \rightarrow A'[I]$ in the backwards pass.

Turning an AllReduce into an AllGather and ReduceScatter also has the convenient property that we can defer the final AllGather until some later moment. Very commonly we'd rather not pay the cost of reassembling the full matrix product replicated across the devices. Rather we'd like to preserve a sharded state even in this case of combining two multiplicands with sharded contracting dimensions:

$$A[I, J_X] \cdot B[J_X, K] \rightarrow C[I, K_X]$$

In this case, we can also perform a ReduceScatter instead of an AllReduce, and then optionally perform the AllGather at some later time, i.e.

$$\begin{aligned} A[I, J_X] \cdot_{LOCAL} B[J_X, K] &\rightarrow C[I, K]\{U_X\} \\ \mathbf{ReduceScatter}_{X,K} C[I, K]\{U_X\} &\rightarrow C[I, K_X] \end{aligned}$$

Note that ReduceScatter *introduces* a sharded dimension, and so has a natural freedom to shard along either the **I** or **K** named dimensions in this case. We generally need to choose *which* named dimension to introduce a new sharding to when using a ReduceScatter (though the choice is usually forced by the larger modeling context). This is why we use the syntax **ReduceScatter$_{X,K}$** to specify the axis to shard.

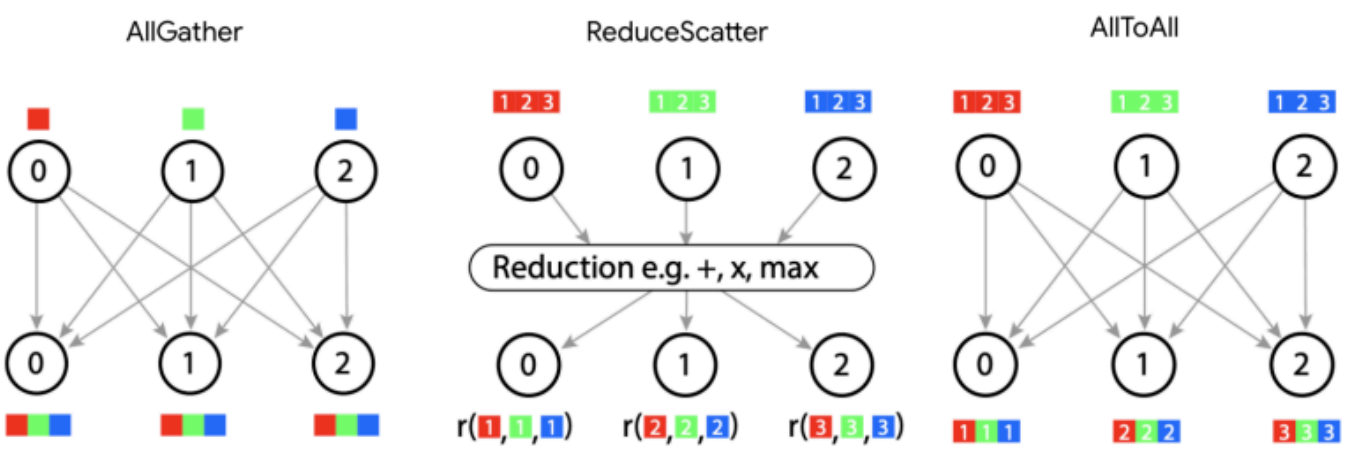
What Have We Learned?

- The sharding of an array is specified by a **Mesh** that names the physical, hardware axes of our TPU mesh and a **Sharding** that assigns mesh axis names to the logical axes of the array.

- For example, **A**[I_{XY}, J] describes an abstract array **A** with its first dimension sharded along two mesh axes X and Y. Combined with Mesh(mesh_shape=(4, 8), axis_names=('X', 'Y')) or the abbreviated Mesh({'X': 4, 'Y': 8}), this tells us our array is sharded 32 ways along the first dimension.

- **Arithmetic with sharded arrays works exactly like with unsharded arrays unless you perform a contraction along a sharded axis**. In that case, we have to introduce some communication. We consider four cases:
  1. *Neither array is sharded along the contracting dimension*: no communication is needed.
  2. *One array is sharded along the contracting dimension* (or the contracting dimensions are sharded along different axes): we AllGather one of the inputs before performing the operation.
  3. *Both arrays are identically sharded along the contracting dimension:* we multiply the shards locally then perform an AllReduce or ReduceScatter.
  4. *Both arrays are sharded along the same mesh axis along a non-contracting dimension:* we AllGather one of the inputs first.

- TPUs use roughly **4 core communication primitives**:
  1. AllGather: $[A_X, B] \rightarrow [A, B]$
  2. ReduceScatter: $[A, B]\{U_X\} \rightarrow [A, B_X]$
  3. AllToAll: $[A, B_X] \rightarrow [A_X, B]$
  4. AllReduce: $[A_X, B]\{U_Y\} \rightarrow [A_X, B]$ (technically not a primitive since it combines a ReduceScatter + AllGather)



- The cost and latency of each of these operations **doesn't depend on the size of the axis (as long as they're bandwidth bound)**, but only on the size of the input arrays and the bandwidth of the link. For a unidirectional AllGather/ReduceScatter:

$$T_{\text{comm per AllGather or ReduceScatter}} = \frac{\text{Data volume}}{\text{bandwidth}} \cdot \frac{\text{Axis} - 1}{\text{Axis}} \longrightarrow \frac{\text{Data volume}}{\text{bandwidth (bidirectional)}}$$

- An AllReduce is composed of a ReduceScatter followed by an AllGather, and thus has 2x the above cost. An AllToAll only has to pass shards part-way around the ring and is thus ¼ the cost of an AllGather. Here's a summary:

Operation	Description	Syntax	Runtime
AllGather	Gathers all the shards of a sharded array along an axis, removing a subscript.	$[A_X, B] \rightarrow [A, B]$	bytes / (bidirectional ICI bandwidth * num_axes)
ReduceScatter	Sums a partially summed array along an axis and shards it along another axis (adding a subscript).	$[A, B]\{U_X\} \rightarrow [A_X, B]$	Same as AllGather
AllReduce	Sums a partially summed array along an axis. Removes a { U_x }. Combines an AllGather and ReduceScatter.	$[A_X, B]\{U_Y\} \rightarrow [A_X, B]$	2 * AllGather
AllToAll	Gathers (replicates) an axis and shards a different dimension along the same axis.	$[A, B_X] \rightarrow [A_X, B]$	AllGather / 4 for a bidirectional ring

Some Problems to Work

Here are some instructive problems based on content in this section. We won't include all answers at the moment but we'll write up more answers as we can.

Question 1 [replicated sharding]: An array is sharded $A[I_X, J, K, \ldots]$ (i.e., only sharded across X), with a mesh `Mesh({'X': 4, 'Y': 8, 'Z': 2})`. What is the ratio of the total number of bytes taken up by A across all chips to the size of one copy of the array?

▶ Click here for the answer.

Question 2 [AllGather latency]: How long should $\text{AllGather}_X([B_X, D_Y])$ take on a TPUv4p 4x4x4 slice with mesh `Mesh({'X': 4, 'Y': 4, 'Z': 4})` if $B = 1024$ and $D = 4096$ in bfloat16? How about $\text{AllGather}_{XY}([B_X, D_Y])$? How about $\text{AllReduce}_Z([B_X, D_Y]\{U_Z\})$?

▶ Click here for the answer.

Question 3 [latency-bound AllGather]: Let's say we're performing an $\text{AllGather}_X([B_X])$ but B is very small (say 128). How long should this take on a TPUv4p 4x4x4 slice with mesh `Mesh({'X': 4, 'Y': 4, 'Z': 4})` in bfloat16? *Hint: you're probably latency bound.*

▶ Click here for the answer.

Question 4 [matmul strategies]: To perform $X[B, D] \cdot_D Y[D_X, F] \to Z[B, F]$, in this section we tell you to perform $\text{AllGather}_X(Y[D_X, F])$ and multiply the fully replicated matrices (Case 2, *Strategy 1*). Instead, you could multiply the local shards like $X[B, D_X] \cdot_D Y[D_X, F] \to Z[B, F]\{U_X\}$ (Case 4, *Strategy 2*), and then $\text{AllReduce}_X(Z[B, F]\{U_X\})$. How many FLOPs and comms does each of these perform? Which is better and why?

▶ Click here for the answer.

Question 5 [minimum latency]: Let's say I want to do a matmul $A[I, J] \cdot_J B[J, K] \to C[I, K]$ on a TPUv5p 4x4x4 with the lowest possible latency. Assume the inputs can be sharded arbitrarily but the result should be fully replicated. How should my inputs be sharded? What is the total FLOPs and comms time?

▶ Click here for the (partial) answer.

Question 6: Let's say we want to perform $A[I_X, J_Y] \cdot_J B[J_Y, K] \to C[I_X, K]$ on TPUv5e 4x4. What communication do we perform? How much time is spent on communication vs. computation?

- What about $A[I_X, J] \cdot_J B[J_X, K_Y] \to C[I_X, K_Y]$? This is the most standard setting for training where we combine data, tensor, and zero sharding.
- What about $A[I_X, J] \cdot_J B[J, K_Y] \to C[I_X, K_Y]$? This is standard for inference, where we do pure tensor parallelism (+data).

Question 7: A typical Transformer block has two matrices $B[D, F]$ and $C[F, D]$ where $F \gg D$. With a batch size B, the whole block is $C \cdot B \cdot x$ with $x[B, D]$. Let's pick $D = 8192$, $F = 32768$, and $B = 128$ and assume everything is in bfloat16. Assume we're running on a TPUv5e 2x2 slice but assume each TPU only has 300MB of free memory. How should **B, C, and the output be sharded to stay below the memory limit while minimizing overall time? How much time is spent on comms and FLOPs?**

Question 8 [challenge]: Using the short code snippet above as a template, allocate a sharded array and benchmark each of the 4 main communication primitives (AllGather, AllReduce, ReduceScatter, and AllToAll) using pmap or shard_map. You will want to use `jax.lax.all_gather`, `jax.lax.psum`, `jax.lax.psum_scatter`, and `jax.lax.all_to_all`. Do you understand the semantics of these functions? How long do they take?

Question 9 [another strategy for sharded matmuls?]: Above we claimed that when only one input to a matmul is sharded along its contracting dimension, we should AllGather the sharded matrix and perform the resulting contracting locally. Another strategy you might think of is to perform the sharded matmul and then AllReduce the result (as if both inputs were sharded along the contracting dimension), i.e. $A[I, J_X] *_J B[J, K] \to C[I, K]$ by way of

1. $C[I, K]\{U_X\} = A[I, J_X] \cdot B[J_X, K]$
2. $C[I, K] = \text{AllReduce}(C[I, K]\{U_X\})$

Answer the following:

1. Explicitly write out this algorithm for matrices $A[N, M]$ and $B[M, K]$, using indices to show exactly what computation is done on what device. Assume A is sharded as $A[I, J_X]$ across ND devices, and you want your output to be replicated across all devices.
2. Now suppose you are ok with the final result not being replicated on each device, but instead sharded (across either the N or K dimension). How would the algorithm above change?
3. Looking purely at the communication cost of the strategy above (in part (b), not (a)), how does this communication cost compare to the communication cost of the algorithm in which we first AllGather A and then do the matmul?

▶ Click here for the answer.

Question 10: Fun with AllToAll: In the table above, it was noted that the time to perform an AllToAll is a factor of 4 lower than the time to perform an AllGather or ReduceScatter (in the regime where we are throughput-bound). In this problem we will see where that factor of 4 comes from, and also see how this factor would change if we only had single-direction ICI links, rather than bidirectional ICI links.

1. Let's start with the single-direction case first. Imagine we have *D* devices in a ring topology, and If we are doing either an AllGather or a ReduceScatter, on an N x N matrix *A* which is sharded as $A[I_X, J]$ (say D divides N for simplicity). Describe the comms involved in these two collectives, and calculate the

total number of scalars (floats or ints) which are transferred across **a single** ICI link during the entirety of this algorithm.

2. Now let's think about an AllToAll, still in the single-directional ICI case. How is the algorithm different in this case than the all-gather case? Calculate the number of scalars that are transferred across a single ICI link in this algorithm.
3. You should have found that the ratio between your answers to part (a) and part (b) is a nice number. Explain where this factor comes from in simple terms.
4. Now let's add bidirectional communication. How does this affect the total time needed in the all-gather case?
5. How does adding bidirectional communication affect the total time needed in the AllToAll case?
6. Now simply explain the ratio between AllGather time and AllToAll time in a bidirectional ring.

▶ Click here for the answer.

That's it for Part 3! For Part 4 (about Transformer math), click here!

Footnotes

1. It's worth noting that we may also choose to parallelize for speed. Even if we could fit on a smaller number of chips, scaling to more simply gives us more FLOPs/s. During inference, for instance, we can sometimes fit on smaller topologies but choose to scale to larger ones in order to reduce latency. Likewise, during training we often scale to more chips to reduce the step time. [↩]
2. A GPU AllGather can also work like this, where you create a ring out of the GPUs in a node and pass the chunks around in that (arbitrary) order. [↩]
3. The factor of 2 in the numerator comes from the fact that we're using the bidirectional bandwidth. We send V/X in each direction, or $2V/X$ total. [↩]
4. technically, $\lceil X/2 \rceil$ [↩]
5. For even-sized bidirectional rings, each device will send $(N/2 + (N/2 - 1) + \ldots + 1)$ chunks right and $((N/2 - 1) + \ldots + 1)$ chunks left $= 0.5 \cdot (N/2) \cdot (N/2 + 1) + 0.5 \cdot (N/2) \cdot (N/2 - 1) = N^2/4$. The size of each chunk (aka shard of a shard) is bytes/N^2 so the per-device cost is $(\text{bytes}/N^2) \cdot N^2/4 = \text{bytes}/4$. This result scales across all devices as the total bandwidth scales with device number. [↩]

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}
}
```