How to Scale Your Model

A Systems View of LLMs on TPUs

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

Contents

Introduction

Much of deep learning still boils down to a kind of black magic, but optimizing the performance of your models doesn't have to — even at huge scale! Relatively simple principles apply everywhere — from dealing with a single accelerator to tens of thousands — and understanding them lets you do many useful things:

- Ballpark how close parts of your model are to their theoretical optimum.
- Make informed choices about different parallelism schemes at different scales (how you split the computation across multiple devices).
- Estimate the cost and time required to train and run large Transformer models.
- Design algorithms that take advantage of specific hardware affordances.
- Design hardware driven by an explicit understanding of what limits current algorithm performance.

Expected background: We're going to assume you have a basic understanding of LLMs and the Transformer architecture but not necessarily how they operate at scale. You should know the basics of LLM training and ideally have some basic familiarity with JAX. Some useful background reading might include this blog post[1] on the Transformer architecture and the original Transformer paper. Also check the list in the conclusion for more useful concurrent and future reading.

Goals & Feedback: By the end, you should feel comfortable estimating the best parallelism scheme for a Transformer model on a given hardware platform, and roughly how long training and inference should take. If you don't, email us or leave a comment! We'd love to know how we could make this clearer.

Why should you care?

Three or four years ago, I don't think most ML researchers would have needed to understand any of the content in this book. But today even "small" models run so close to hardware limits that doing novel research requires you to think about efficiency at scale.[2] **A 20% win on benchmarks is irrelevant if it comes at a 20% cost to roofline efficiency.** Promising model architectures routinely fail either because they *can't* run efficiently at scale or because no one puts in the work to make them do so.

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. This is known as "*strong scaling*". Although adding additional chips ("parallelism") usually decreases the computation time, it also comes at the cost of added communication between chips. When communication takes longer than computation we become "communication bound" and cannot scale strongly.[3] If we understand our hardware well enough to antici-

[1] `https://jalammar.github.io/illustrated-transformer/`

[2] Historically, ML research has followed something of a tick-tock cycle between systems innovations and software improvements. Alex Krizhevsky had to write unholy CUDA code to make CNNs fast but within a couple years, libraries like Theano and TensorFlow meant you didn't have to. Maybe that will happen here too and everything in this book will be abstracted away in a few years. But scaling laws have pushed our models perpetually to the very frontier of our hardware, and it seems likely that, in the near future, doing cutting edge research will be inextricably tied to an understanding of how to efficiently scale models to large hardware topologies.

[3] As your computation time decreases, you also typically face bottlenecks at the level of a single chip. Your shiny new TPU or GPU may be rated to perform 500 trillion operations-per-second, but if you aren't careful it can just as easily do a tenth of that if it's bogged down moving parameters around in memory. The interplay of per-chip computation, memory bandwidth, and total memory is critical to the scaling story.

pate where these bottlenecks will arise, we can design or reconfigure our models to avoid them.[4]

Our goal in this book is to explain how TPU (and GPU) hardware works and how the Transformer architecture has evolved to perform well on current hardware. We hope this will be useful both for researchers designing new architectures and for engineers working to make the current generation of LLMs run fast.

High-Level Outline

The overall structure of this book is as follows::

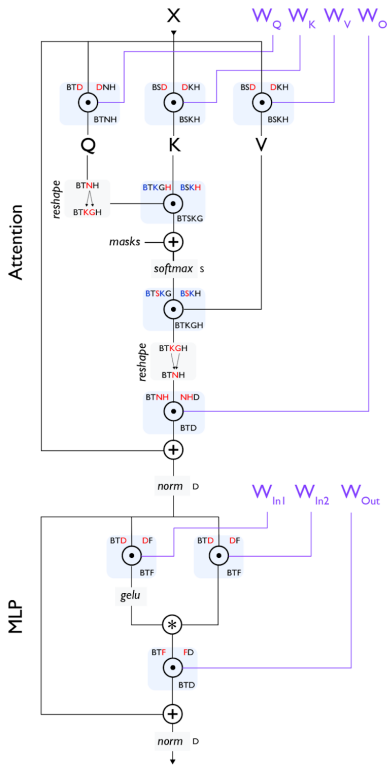
Section 1 explains roofline analysis and what factors can limit our ability to scale (communication, computation, and memory). Section 2 and Section 3 talk in detail about how TPUs and modern GPUs work, both as individual chips and — of critical importance — as an interconnected system with inter-chip links of limited bandwidth and latency. We'll answer questions like:

- How long should a matrix multiply of a certain size take? At what point is it bound by compute or by memory or communication bandwidth?
- How are TPUs wired together to form training clusters? How much bandwidth does each part of the system have?
- How long does it take to gather, scatter, or re-distribute arrays across multiple TPUs?
- How do we efficiently multiply matrices that are distributed differently across devices?

Five years ago ML had a colorful landscape of architectures — ConvNets, LSTMs, MLPs, Transformers — but now we mostly just have the Transformer. We strongly believe it's worth understanding every piece of the Transformer architecture: the exact sizes of every matrix, where normalization occurs, how many parameters and FLOPs[5] are in each part. Section 4 goes through this "Transformer math" carefully, showing how to count the parameters and FLOPs for both training and inference. This tells us how much memory our model will use, how much time we'll spend on compute or comms, and when attention will become important relative to the feed-forward blocks.

[4]Hardware designers face the inverse problem: building hardware that provides just enough compute, bandwidth, and memory for our algorithms while minimizing cost. You can imagine how stressful this "co-design" problem is: you have to bet on what algorithms will look like when the first chips actually become available, often 2 to 3 years down the road. The story of the TPU is a resounding success in this game. Matrix multiplication is a unique algorithm in the sense that it uses far more FLOPs per byte of memory than almost any other (N FLOPs per byte), and early TPUs and their systolic array architecture achieved far better perf / $ than GPUs did at the time they were built. TPUs were designed for ML workloads, and GPUs with their TensorCores are rapidly changing to fill this niche as well. But you can imagine how costly it would have been if neural networks had not taken off, or had changed in some fundamental way that TPUs (which are inherently less flexible than GPUs) could not handle.

[5]FLoating point OPs, basically the total number of adds and multiplies required. While many sources take FLOPs to mean "operations per second", we use FLOPs/s to indicate that explicitly.



symbol	dimension
B	batch
L	number of layers
T	sequence length (query)
S	sequence length (key value)
V	vocab
D	d_model, embedding dimension
F	MLP hidden dimension
H	attention head dimension
N	number of query heads
K	number of key/value heads
G	q heads per kv head = N // K

Figure 1: A standard Transformer layer with each matrix multiplication (matmul) shown as a dot inside a circle. All parameters (excluding norms) are shown in purple. Section 4 walks through this diagram in more detail.

Section 5: Training and Section 7: Inference are the core of this essay, where we discuss the fundamental question: given a model of some size and some number of chips, how do I parallelize my model to stay in the "strong scaling" regime? This is a simple question with a surprisingly complicated answer. At a high level, there are 4 primary parallelism techniques used to split models over multiple chips (**data**, **tensor**, **pipeline** and **expert**), and a number of other techniques to reduce the memory requirements (**rematerialisation**, optimizer/model sharding aka **ZeRO**, **host offload**, **gradient accumulation**). We discuss many of these here.

We hope by the end of these sections you should be able to choose among them yourself for new architectures or settings. Section 6 and Section 8 are practical tutorials that apply these concepts to LLaMA-3, a popular open-source model.

Finally, Section 9 and Section 10 look at how to implement some of these ideas in JAX and how to profile and debug your code when things go wrong.

Throughout we try to give you problems to work for yourself. Please feel no pressure to read all the sections or read them in order. And please leave feedback. For the time being, this is a draft and will continue to be revised. Thank you!

We'd like to acknowledge James Bradbury and Reiner Pope who derived many of the ideas in this doc but have since left Google.

1 Intro to Rooflines

When we run algorithms on hardware, we're bounded by three things: how fast our computer can do math (OPs/second), the bandwidth available for moving data around (bytes/second), and the total memory available to store data (bytes). These constraints let us upper and lower bound the time of a given computation.

1.1 Where Does the Time Go?

Let's start with an extremely simple question: *why does an algorithm take 50ms instead of 50s or 5ms*? What is actually happening within the model that takes substantial time and how long should we expect it to take?

Computation: A deep learning model is effectively a bunch of matrix multiplications, each composed of floating-point multiplication and addition 'operations' (FLOPs). Our accelerator speed determines how long these take to compute:

$$T_{\text{math}} = \frac{\text{Computation FLOPs}}{\text{Accelerator FLOPs/s}}$$

For instance, an NVIDIA H100 can perform about 9.89e14 bfloat16[1] FLOPs/s while a TPU v6e can perform 9.1e14 FLOPs/s. That means doing 1e12 FLOPs on an H100 will take (roughly) `1e12 / 9.89e14 = 1.01ms` and `1e12 / 9.1e14 = 1.1ms` on a TPU v6e.[2]

Communication within a chip: *Within an accelerator*, tensors need to be transferred between on-chip memory (HBM) and the compute cores. You'll see the bandwidth of this link referred to as "HBM bandwidth"[3] On an H100, this is about 3.35TB/s [4] and on TPU v6e this is about 1.6TB/s [5].

Communication between chips: When we distribute a model *across multiple accelerators*, tensors frequently need to be transferred between them. There are often a few options for this on our hardware (ICI, DCN, and PCIe), each with different bandwidths.

Whether the communication is within a chip or between chips, we measure this in bytes/s and estimate the total communication time with:

$$T_{\text{comms}} = \frac{\text{Communication Bytes}}{\text{Network/Memory Bandwidth Bytes/s}}$$

Typically (but not always), computation within a single chip can be overlapped with communication within a chip and between chips. This means **we can lower-bound training and inference time by using the maximum of computation and communication time**. We can also **upper-bound with their sum**. In practice, we optimize against the maximum as the algebra is simpler and we can usually come close to this bound by overlapping our communication and computation. If we optimize with the maximum in mind then the lower and upper bounds differ by at most a factor of 2 since $T_{\text{math}} + T_{\text{comms}} \leq 2 * \max(T_{\text{math}}, T_{\text{comms}})$. We then increase accuracy beyond this by modeling 'overlap regions' and overheads, which can be informed by profiling your specific model and target system.

$$T_{\text{lower}} = \max(T_{\text{math}}, T_{\text{comms}})$$
$$T_{\text{upper}} = T_{\text{math}} + T_{\text{comms}}$$

[1] bf16 is short for bfloat16, a 16-bit floating point format often used in ML.
[2] Note that these chips are priced differently, and this comparison does not normalize to cost.
[3] NVIDIA also calls this "memory bandwidth."
[4] `https://www.nvidia.com/en-us/data-center/h100/`
[5] `https://cloud.google.com/tpu/docs/v6e`

If we assume we can perfectly overlap communication and computation, when $T_{\text{math}} > T_{\text{comms}}$, we see full utilization from our hardware. We call this being "compute-bound". When $T_{\text{comms}} > T_{\text{math}}$, we tend to be "communication-bound" and at least some fraction of our accelerator FLOPs/s is wasted waiting for data to be passed around. One way to tell if an operation will be compute or communication-bound is to look at its "*arithmetic intensity*" or "*operational intensity*".

Definition: the arithmetic intensity of an algorithm is given by the ratio of the total FLOPs it performs to the number of bytes it needs to communicate — either within a chip or between chips.

$$\text{Arithmetic Intensity} = \frac{\text{Computation FLOPs}}{\text{Communication Bytes}}$$

Arithmetic intensity measures the "FLOPs per byte" of a given operation. To a first order, when our arithmetic intensity is high, T_{math} is large compared to T_{comms} and we typically use most of the available FLOPs. When the opposite is true, we spent more time on comms and waste FLOPs. The point where this crossover happens is the "peak arithmetic intensity" of our hardware, the ratio of peak accelerator FLOPs/s to accelerator bandwidth.

$$\begin{aligned} T_{\text{math}} > T_{\text{comms}} &\Leftrightarrow \frac{\text{Computation FLOPs}}{\text{Accelerator FLOPs/s}} > \frac{\text{Communication Bytes}}{\text{Bandwidth Bytes/s}} \\ &\Leftrightarrow \frac{\text{Computation FLOPs}}{\text{Communication Bytes}} > \frac{\text{Accelerator FLOPs/s}}{\text{Bandwidth Bytes/s}} \\ &\Leftrightarrow \text{Intensity(Computation)} > \text{Intensity(Accelerator)} \end{aligned}$$

The quantity Intensity(Accelerator) is the arithmetic intensity at which our accelerator achieves its peak FLOPs/s. **For the TPU v5e MXU, this is about 240 FLOPs/byte**[6], since the TPU can perform `1.97e14` FLOPs/s and load `8.2e11` bytes/s from HBM. That means if an algorithm has a lower arithmetic intensity than 240[7] FLOPs/byte, it will be bound by byte loading and thus we won't make good use of our hardware. Let's look at one such example:

Example (dot product): to compute the dot product of two vectors in bfloat16 precision, `x · y: bf16[N], bf16[N] → bf16[1]`, we need to load x and y from memory, each of which has $2 * N = 2N$ bytes, perform N multiplications and $N - 1$ additions, and write 2 bytes back into HBM.

$$\text{Intensity(dot product)} = \frac{\text{Total FLOPs}}{\text{Total Bytes}} = \frac{N + N - 1}{2N + 2N + 2} = \frac{2N - 1}{4N + 2} \to \frac{1}{2}$$

as $N \to \infty$. So the dot product has an arithmetic intensity of $\frac{1}{2}$ or, put another way, the dot product does 0.5 floating point operations per byte loaded. This means our arithmetic intensity is lower than that of our hardware and we will be communication-bound.[8]

[6] The MXU is the matrix multiply unit on the TPU. We specify this here because the TPU has other accelerators like the VPU that are responsible for elementwise operations that have a different peak FLOPs/s.

[7] This is only true if the algorithm loads its weights from HBM and runs in the MXU. As we'll discuss in the next section, we can sometimes store parameters in VMEM which has a much higher bandwidth. Many algorithms also run in the VPU, which has different performance characteristics.

[8] The 240 number above is not the correct comparison here since, as you will see in the next section, a dot-product is performed on the VPU and not the MXU. The TPU v5p VPU can do roughly 7e12 FLOPs / second, so its critical intensity is around 3, which means we are still somewhat comms-bound here. Either way, the fact that our intensity is low and constant means it is difficult on most hardware to be compute-bound.

1.1.1 Visualizing rooflines

We can visualize the tradeoff between memory and compute using a **roofline plot**, which plots the peak achievable FLOPs/s (throughput) of an algorithm on our hardware (the y-axis) against the arithmetic intensity of that algorithm (the x-axis). Here's an example log-log plot:

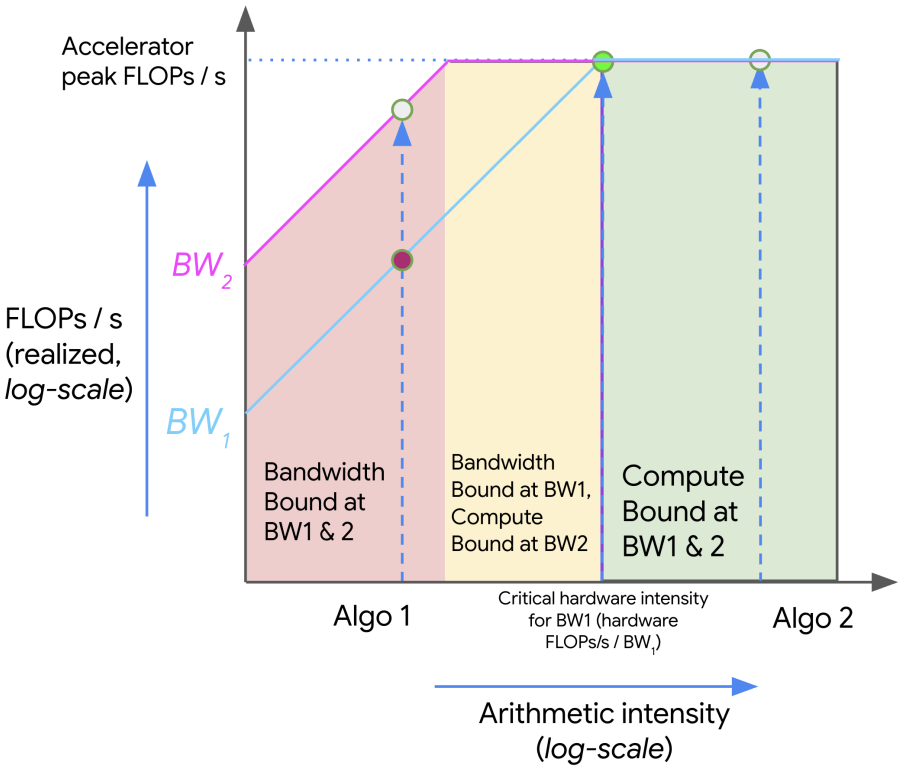


Figure 2: An example roofline plot showing two algorithms with different arithmetic intensities (Algo 1 and Algo 2) and their corresponding theoretical peak throughput under different bandwidths (BW1 and BW2). In the red area, an algorithm is bandwidth bound at both bandwidths and is wasting some fraction of the hardware's peak FLOPs/s. The yellow area is bandwidth-bound only at the lower bandwidth (BW1). The green area is compute-bound at all bandwidths. Here, we are using the peak FLOPs/s of the accelerator and increasing bandwidth or improving intensity yield no benefit."

Above, as the intensity increases (moving left to right), we initially see a linear increase in the performance of our algorithm (in FLOPs/s) until we hit the critical arithmetic intensity of the hardware, 240 in the case of the TPU v5e. Any algorithm with a lower intensity will be bandwidth (BW) bound and limited by the peak memory bandwidth (shown in red). Any algorithm to the right will fully utilize our FLOPs (shown in green). Here, Algo 1 is comms-bound and uses only a fraction of the total hardware FLOPs/s. Algo 2 is compute-bound. We can generally improve the performance of an algorithm either by increasing its arithmetic intensity or by increasing the memory bandwidth available (moving from BW1 to BW2).

1.1.2 Matrix multiplication

Let's look at our soon-to-be favorite algorithm: matrix multiplication (aka matmul). We write $X * Y \to Z$ where X has shape bf16$[B, D]$, Y has shape bf16$[D, F]$, and Z has shape bf16$[B, F]$. To do the matmul

we need to load $2DF + 2BD$ bytes, perform $2BDF$ FLOPs, and write $2BF$ bytes back.[9][10] Thus:

$$\text{Intensity(matmul)} = \frac{2BDF}{2BD + 2DF + 2BF} = \frac{BDF}{BD + DF + BF}$$

We can get a nice simplification if we assume our local "batch size" B is small relative to D and F. Then we get

$$\frac{BDF}{BD + DF + BF} \approxeq \frac{BDF}{DF} = B \tag{1}$$

$$\text{Intensity(matmul)} > \text{Intensity(TPU)} \implies B > \frac{1.97e14}{8.20e11} = 240 \tag{2}$$

This is a reasonable assumption for Transformer matmuls since for most of our models we have our local batch size in tokens $B < 1024$ but D and $F > 8000$. Thus we become compute-bound when our local batch size is greater than 240 tokens, a very simple rule!

Takeaway: for a bfloat16 matmul to be compute-bound on most TPUs, we need our local batch size in tokens to be greater than 240.

This comes with a few notable caveats we'll explore in the problems below, particularly with respect to quantization (e.g. if we quantize our activations but still do full-precision FLOPs), but it's a good rule to remember. For GPUs, this number is slightly higher (closer to 300), but the same conclusion generally holds. We'll discuss the lower-level GPU and TPU details in the next section.

1.1.3 Network communication rooflines

All the rooflines we've discussed so far have been memory-bandwidth rooflines, *all within a single chip*. This shouldn't be taken as a rule. In fact, most of the rooflines we'll care about in this book involve communication between chips: usually matrix multiplications that involve matrices sharded across multiple TPUs.

To pick a somewhat contrived example, say we want to multiply two big matrices $X \sim$ bfloat16[B, D] and $Y \sim$ bfloat16[D, F] which are split evenly across 2 TPUs/GPUs (along the D dimension). To do this multiplication (as we'll see in Section 3), we can multiply half of each matrix on each TPU (`A = X[:, :D // 2] @ Y[:D // 2, :]` on TPU 0 and `B = X[:, D // 2:] @ Y[D // 2:, :]` on TPU 1) and then copy the resulting "partial sums" to the other TPU and add them together. Say we can copy `4.5e10` bytes in each direction and perform `1.97e14` FLOPs/s on each chip. What are T_{math} and T_{comms}?

T_{math} is clearly half of what it was before, since each TPU is doing half the work, i.e.[11]

$$T_{\text{math}} = \frac{2BDF}{2 \cdot \text{Accelerator FLOPs/s}} = \frac{BDF}{1.97e14}$$

Now what about T_{comms}? This now refers to the communication time between chips! This is just the total bytes sent divided by the network bandwidth, i.e.

$$T_{\text{comms}} = \frac{2BF}{\text{Network Bandwidth}} = \frac{2BF}{4.5e10}$$

[9] Technically we perform $BF \times (2D - 1)$ FLOPs but this is close enough. This comes from BDF multiplications and $BF * (D - 1)$ additions. Section 4 has more details.

[10] Although the output of a matmul is technically float32 we usually cast down to bfloat16 before copying back to HBM.

[11] We're ignoring the FLOPs required to add the two partial sums together (another DF additions), but this is basically negigible.

Therefore we become compute-bound (now with respect to the inter-chip network) when Intensity(matmul (2-chips)) > Intensity(TPU w.r.t. inter-chip network) or equivalently when $\frac{BDF}{2BF} = \frac{D}{2} > \frac{1.97e14}{4.5e10} = 4377$ or $D > 8755$. Note that, unlike before, the critical threshhold now depends on D and not B! Try to think why that is. This is just one such example, but we highlight that this kind of roofline is critical to knowing when we can parallelize an operation across multiple TPUs.

1.2 Worked problems

Exercise 1.1 *[int8 matmul]*

Say we want to do $X[B, D] \cdot_D Y[D, F] \to Z[B, F]$ in int8 precision (1 byte per parameter) instead of bfloat16.[12]

1. How many bytes need to be loaded from memory? How many need to be written back to memory?
2. How many total OPs are performed?
3. What is the arithmetic intensity?
4. What is a roofline estimate for T_{math} and T_{comms}? What are reasonable upper and lower bounds for the runtime of the whole operation?

Assume our HBM bandwidth is `8.1e11` bytes/s and our int8 peak OPs/s is `3.94e14`.

Exercise 1.2 *[int8 matmul + bf16 matmul]*

In practice we often do different weight vs. activation quantization, so we might store our weights in very low precision but keep activations (and compute) in a higher precision. Say we want to quantize our weights in int8 but keep activations (and compute) in bfloat16. At what batch size do we become compute bound? Assume `1.97e14` bfloat16 FLOPs/s.

Hint: this means specifically `bfloat16[B, D] * int8[D, F] -> bfloat16[B, F]` *where B is the "batch size".*

Exercise 1.3

For the problem above, make a roofline plot of peak FLOPs vs. B for several values of D and F.

Exercise 1.4

What if we wanted to perform int8[B, D] $*_D$ int8[B, D, F] $\to$ int8[B, F] where we imagine having a different matrix for each batch element. What is the arithmetic intensity of this operation?

Exercise 1.5 *[Memory Rooflines for GPUs]*

Using the spec sheet provided by NVIDIA for the H100[13], calculate the batch size at which a matrix multiplication will become compute-bound. *Note that the Tensor Core FLOPs numbers are twice the true value since they're only achievable with structured sparsity.*

[12] Here and throughout we'll use the notation $A \cdot_D B$ to indicate that the multiplication is performing a contraction over the D dimension. This is an abuse of einsum notation.

[13] `https://www.nvidia.com/en-us/data-center/h100/`

2 All About TPUs

This section is all about how TPUs work, how they're networked together to enable multi-chip training and inference, and how this affects the performance of our favorite algorithms. There's even some good stuff for GPU users too!

2.1 What is a TPU?

A TPU is basically a compute core that specializes in matrix multiplication (called a TensorCore) attached to a stack of fast memory (called high-bandwidth memory or HBM). Here's a diagram:

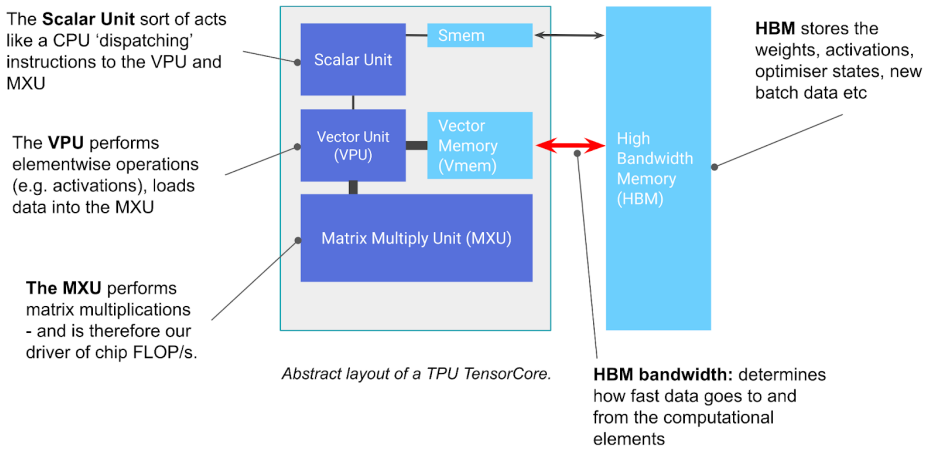


Figure 3: the basic components of a TPU chip. The TensorCore is the gray left-hand box, containing the matrix-multiply unit (MXU), vector unit (VPU), and vector memory (VMEM).

You can think of the TensorCore as basically just being a really good matrix multiplication machine, but it has a few other functions worth noting. The TensorCore has three key units:

- The **MXU** (Matrix Multiply Unit) is the core of the TensorCore. For most TPU generations, it performs one `bfloat16[8,128] @ bf16[128,128] -> f32[8,128]` matrix multiply[1] every 8 cycles using a systolic array (see Appendix B for details).
  - This is about `5e13` bf16 FLOPs/s per MXU at 1.5GHz on TPU v5e. Most TensorCores have 2 or 4 MXUs, so e.g. the total bf16 FLOPs/s for TPU v5e is `2e14`.
  - TPUs also support lower precision matmuls with higher throughput (e.g. each TPU v5e MXU can do `4e14` int8 OPs/s).
- The **VPU** (Vector Processing Unit) performs general mathematical operations like ReLU activations or pointwise addition or multiplication between vectors. Reductions (sums) are also performed here. Appendix C provides more details.
- **VMEM** (Vector Memory) is an on-chip scratchpad located in the TensorCore, close to the compute units. It is much smaller than HBM (for example, 128 MiB on TPU v5e) but has a much higher bandwidth to the MXU. VMEM operates somewhat like an L1/L2 cache on CPUs but is much larger and programmer-controlled. Data in HBM needs to be copied into VMEM before the TensorCore can do any computation with it.

[1] TPU v6e (Trillium) has a 256x256 MXU, while all previous generations use 128x128

TPUs are very, very fast at matrix multiplication. It's mainly what they do and they do it well. TPU v5p, one of the most powerful TPUs to date, can do `2.5e14` bf16 FLOPs / second / core or `5e14` bf16 FLOPs / sec / chip. A single pod of 8960 chips can do 4 exaflops / second. That's a lot. That's one of the most powerful supercomputers in the world. And Google has a lot of them. [2]

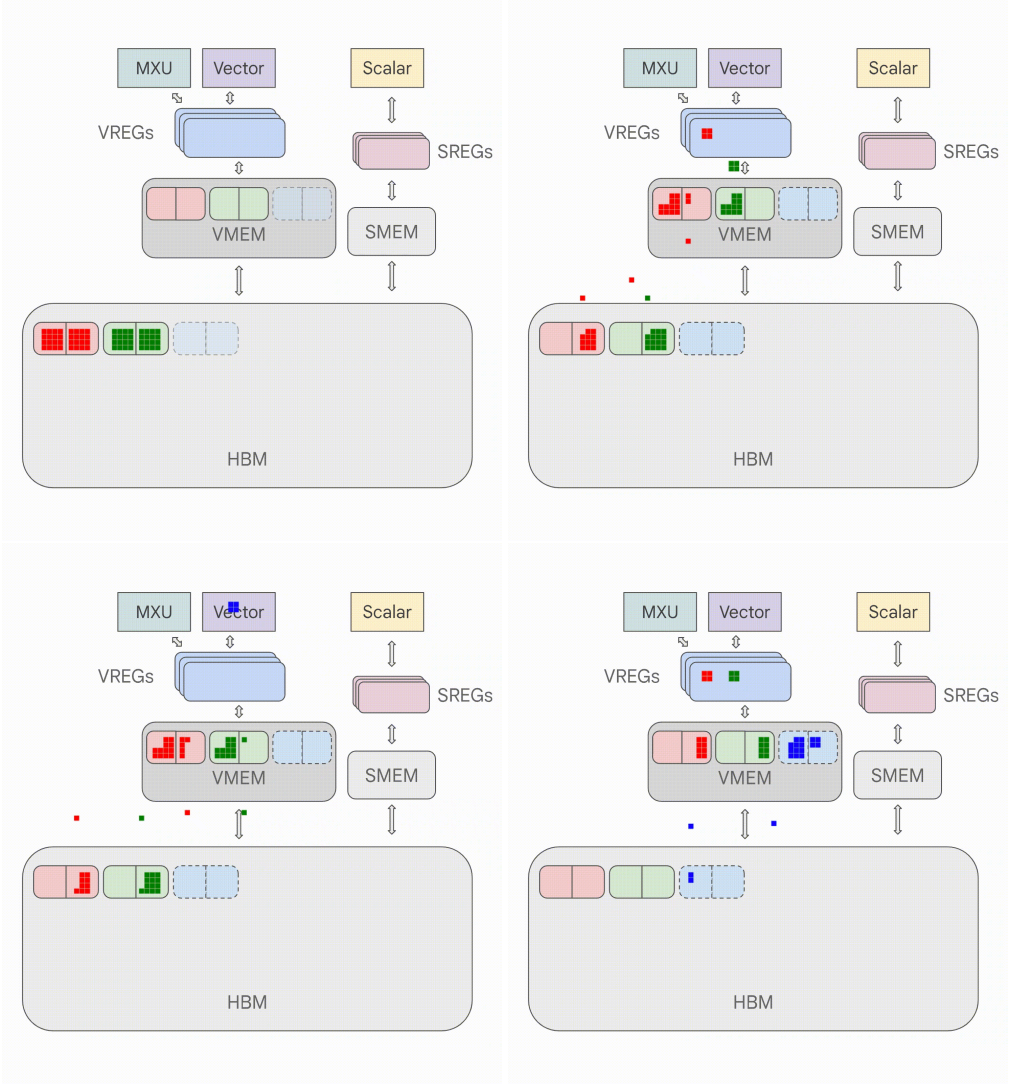
The diagram above also includes a few other components like SMEM and the scalar unit, which are used for control flow handling and are discussed briefly in Appendix C, but aren't crucial to understand. On the other hand, HBM is important and fairly simple:

- **HBM** (High Bandwidth Memory) is a big chunk of fast memory that stores tensors for use by the TensorCore. HBM usually has capacity on the order of tens of gigabytes (for example, TPU v5e has 16GiB of HBM).
  - When needed for a computation, tensors are streamed out of HBM through VMEM (see below) into the MXU and the result is written from VMEM back to HBM.
  - The bandwidth between HBM and the TensorCore (through VMEM) is known as "HBM bandwidth" (usually around 1-2TB/sec) and limits how fast computation can be done in memory-bound workloads.

Generally, all TPU operations are pipelined and overlapped. To perform a matmul $X \cdot A \to Y$, a TPU would first need to copy chunks of matrices A and X from HBM into VMEM, then load them into the MXU which multiplies chunks of 8x128 (for X) and 128x128 (for A), then copy the result chunk by chunk back to HBM. To do this efficiently, the matmul is pipelined so the copies to/from VMEM are overlapped with the MXU work. This allows the MXU to continue working instead of waiting on memory transfers, keeping matmuls compute-bound, not memory-bound.

Here's an example of how you might perform an elementwise product from HBM. These are excerpts from an animation found in the online version of the book:

[2] TPUs, and their systolic arrays in particular, are such powerful hardware accelerators because matrix multiplication is one of the few algorithms that uses $O(n^3)$ compute for $O(n^2)$ bytes. That makes it very easy for an ordinary ALU to be bottlenecked by compute and not by memory bandwidth.

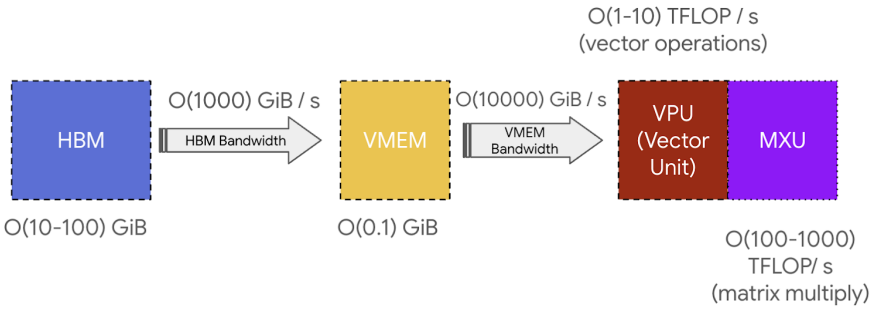


A matmul would look nearly identical except it would load into the MXU instead of the VPU/Vector unit, and the loads and stores would occur in a different order, since the same weight chunk is used for multiple chunks of activations. You can see chunks of data streaming into VMEM, then into the VREGs (vector registers), then into the Vector Unit, then back into VMEM and HBM. As we're about to see, if the load from HBM to VMEM is slower than the FLOPs in the Vector Unit (or MXU), we become "bandwidth bound" since we're starving the VPU or MXU of work.

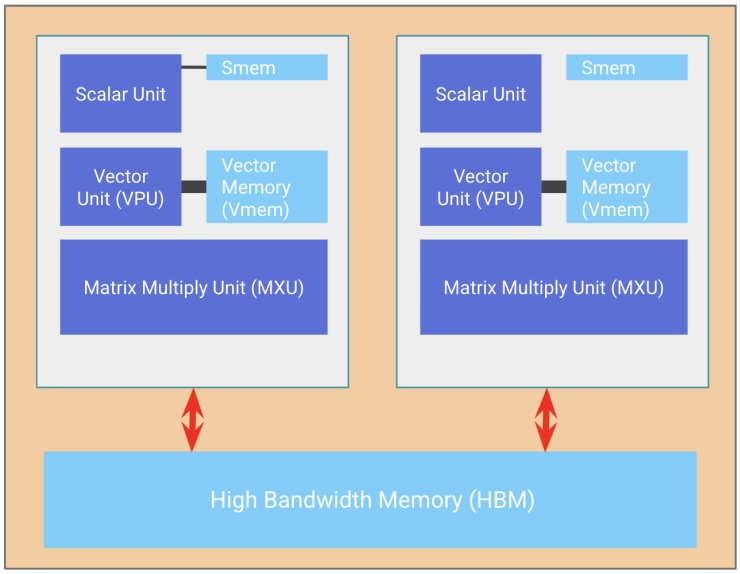
Takeaway: TPUs are very simple. They load weights from HBM into VMEM, then from VMEM into a systolic array which can perform around 200 trillion multiply-adds per second. The HBM ↔ VMEM and VMEM ↔ systolic array bandwidths set fundamental limits on what computations TPUs can do efficiently.

VMEM and arithmetic intensity: VMEM is much smaller than HBM but it has a much higher bandwidth to the MXU. As we saw in Section 1, this means if an algorithm can fit all its inputs/outputs in VMEM, it's much less likely to hit communication bottlenecks. This is particularly helpful when a computation has

poor arithmetic intensity: VMEM bandwidth is around 22x higher than HBM bandwidth which means an MXU operation reading from/writing to VMEM requires an arithmetic intensity of only 10-20 to achieve peak FLOPs utilization. That means if we can fit our weights into VMEM instead of HBM, our matrix multiplications can be FLOPs bound at much smaller batch sizes. And it means algorithms that fundamentally have a lower arithmetic intensity can still be efficient. VMEM is just so small this is often a challenge.[3]



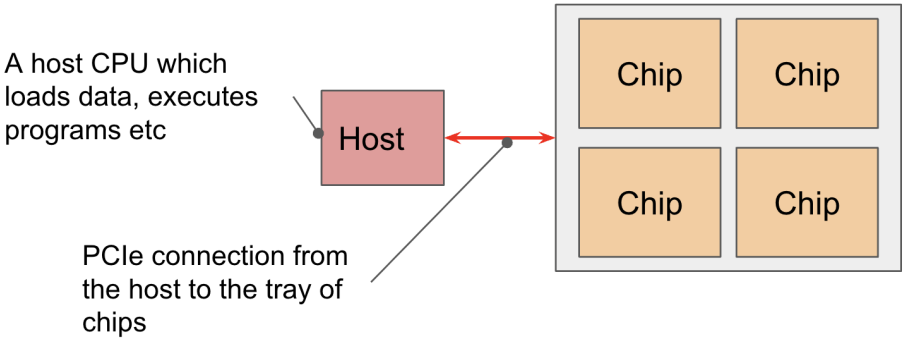
A TPU chip typically (but not always) consists of two TPU cores which share memory and can be thought of as one large accelerator with twice the FLOPs (known as a "megacore" configuration). This has been true since TPU v4. Older TPU chips they have separate memory and are regarded as two separate accelerators (TPU v3 and older). Inference-optimized chips like the TPU v5e only have one TPU core per chip.



Chips are arranged in **sets of 4 on a 'tray'** connected to a **CPU host via PCIe network.** This is the format most readers will be familiar with, 4 chips (8 cores, though usually treated as 4 logical megacores) exposed through Colab or a single TPU-VM. For inference chips like the TPU v5e, we have 2 trays per host, instead of 1, but also only 1 core per chip, giving us 8 chips = 8 cores. [4]

[3]We sometimes talk about VMEM prefetching, which refers to loading weights ahead of time in VMEM so we can mask the cost of loading for our matmuls. For instance, in a normal Transformer we can sometimes load our big feed-forward weights into VMEM during attention, which can hide the cost of the weight load if we're memory bandwidth bound. This requires our weights to be small enough or sharded enough to fit a single layer into VMEM with space to spare.

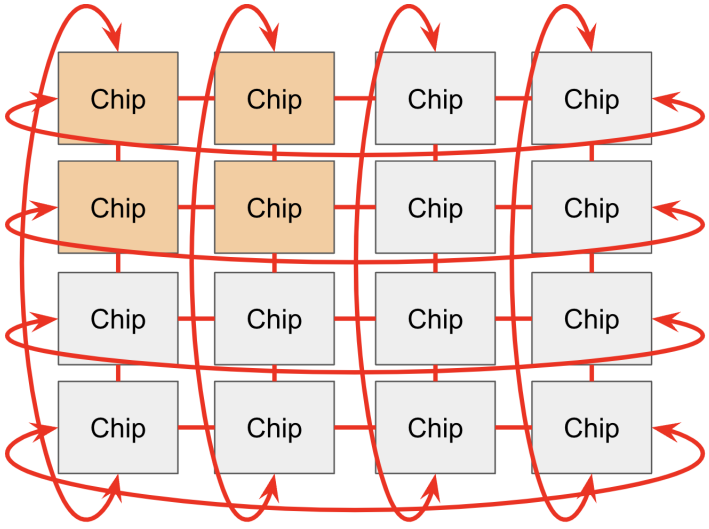
[4]On Cloud TPU VMs, each tray is exposed as part of a separate VM, so there are once again 4 cores visible.



PCIe bandwidth is limited: Like the HBM ↔ VMEM link, the CPU ↔ HBM PCIe connection has a specific bandwidth that limits how quickly you can load from host memory to HBM or vice-versa. PCIe bandwidth for TPU v4 is 16GB / second each way, for example, so close to 100x slower than HBM. We *can* load/offload data into the host (CPU) RAM, but not very quickly.

2.2 TPU Networking

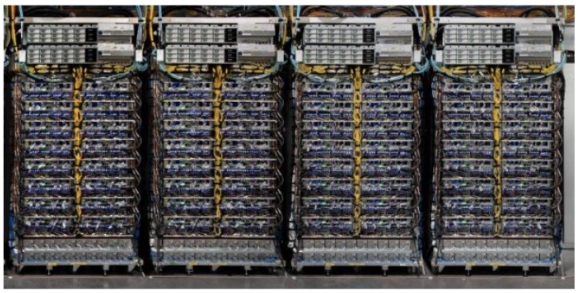
Chips are connected to each other through the ICI network in a Pod. In older generations (TPU v2 and TPU v3), inference chips (e.g. TPU v5e), and Trilium (TPU v6e), ICI ("inter-chip interconnects") connects the 4 nearest neighbors (with edge links to form a 2D torus). TPU v4 and TPU v5p are connected to the nearest 6 neighbors (forming a 3D torus). Note these connections do **not** go through their hosts, they are direct links between chips.



The toroidal structure reduces the maximum distance between any two nodes from N to $N/2$, making communication much faster. TPUs also have a "twisted torus" configuration that wraps the torus in a Mobius-strip like topology to further reduce the average distance between nodes.

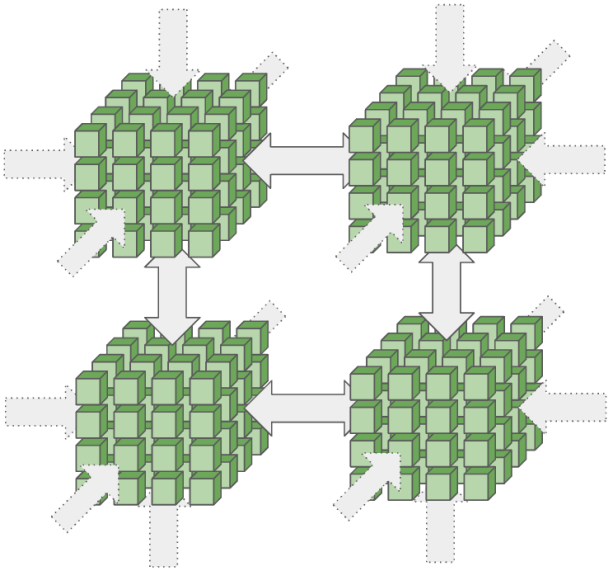
TPU pods (connected by ICI) can get really big: the maximum pod size (called a superpod) is `16x16x16` for TPU v4 and `16x20x28` for TPU v5p. These large pods are composed of reconfigurable cubes of `4x4x4` chips

connected by optical wraparound links[5] that we can reconfigure to connect very large topologies.

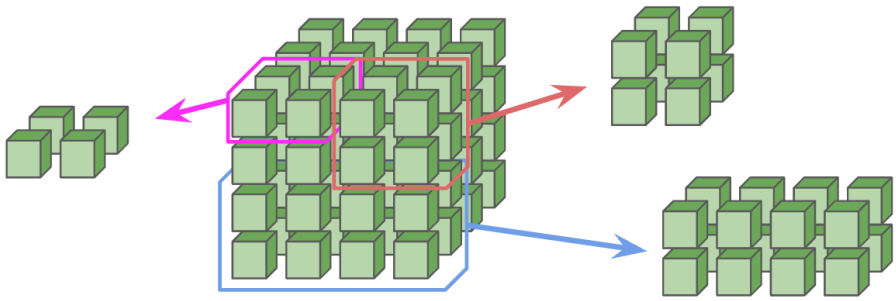


Each rack corresponds to 4x4x4 TPUv4s

These can be configured into arbitrary
4ix4jx4k toruses via optical interconnects



Smaller topologies (e.g. `2x2x1`, `2x2x2`) can also be requested, albeit with no wraparounds. This is an important caveat, since it typically doubles the time of most communication. Any multiple of a full cube (e.g. `4x4x4` or `4x4x8`) will have wraparounds provided by the optical switches.[6]

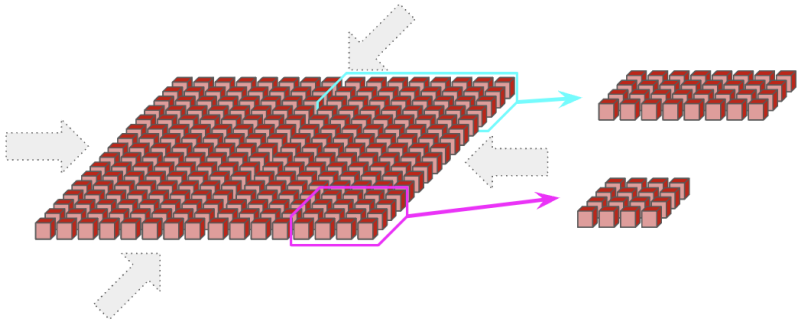


Smaller slices can also be requested.

TPU v5e and Trillium pods consist of a single `16x16` 2D torus with wraparounds along any axis of size 16 (meaning an `8x16` has a wraparound on the long axis). TPUs v5e and v6e (Trillium) cannot expand beyond a 16x16 torus but pods can still communicate with each other over standard data-center networking (DCN), which connects TPU hosts to each other. Again, smaller topologies can be requested without wraps on dims < 16.

[5]The optical switch is simply a reconfigurable connection with the same ICI bandwidth. It just lets us connect cubes while retaining a wraparound link.

[6]Note that a `2x2x4` won't have any wraparounds since they are provided by the optical switches which are only available on a full cube. A TPU v5e 8x16 *will* have a wraparound on the longer axis, however, since it doesn't use reconfigurable optical networking.



This nearest-neighbor connectivity is a key difference between TPUs and GPUs. GPUs are connected with a hierarchy of switches that approximate a point-to-point connection between every GPU, rather than using local connections like a TPU. Typically, GPUs within a node (8 GPUs for H100 or as many as 500 for B200) are directly connected, while larger topologies require O(log(N)) hops between each GPU. On the one hand, that means GPUs can send arbitrary data within a node in a single low-latency hop. On the other hand, TPUs are dramatically cheaper (since NVLink switches are expensive) and simpler to wire together, and can scale to much larger topologies because the number of links per device and the bandwidth per device is constant.

ICI is very fast relative to DCN, but is still slower than HBM bandwidth. For instance, a TPU v5p[7] has:

- `2.5e12` bytes/s (2.5 TB/s) of HBM bandwidth per chip.
- `9e10` bytes/s (90[8] GB/s) of ICI bandwidth per axis, with 3 axes per chip.
- `2.5e10` bytes/s (25 GB/s) of DCN (egress) bandwidth per host. Since we typically have 8 TPUs per host, this is really closer to `3.1e9` bytes / s / chip.

This means that when we split models across multiple chips, we need to be careful to avoid bottle-necking the MXU with slower cross-device communication.

Multi-slice training: A set of ICI-connected TPUs is called a **slice**. Different slices can be connected between each other using DCN, for instance to link slices on different pods. Since DCN is a much slower connection than ICI, one should try to limit how much our computation has to wait for data from DCN. DCN is host-to-host, so to transfer buffers from TPU to TPU over DCN, we first need to transfer over PCIe to the host, then egress over the network, then ingress over the target host network, then over PCIe into HBM.

2.3 Key Takeaways

- TPUs are simple and can in most cases be thought of as a matrix multiply unit connected to memory (super fast), other chips over ICI (rather fast), and the rest of the datacenter over DCN (somewhat fast).
- Communication is limited by our various network bandwidths in order of speed:
  - HBM bandwidth: Between a TensorCore and its associated HBM.

[7] `https://cloud.google.com/tpu/docs/v5p#system_architecture`

[8] The page above lists 100 GB/s of bandwidth, which is slightly different from what's listed here. TPU ICI links have slightly different bandwidths depending on the operation being performed. You can generally use the numbers in this doc without worry.

  - ICI bandwidth: Between a TPU chip and its nearest 4 or 6 neighbors.
  - PCIe bandwidth: Between a CPU host and its associated tray(s) of chips.
  - DCN bandwidth: Between multiple CPU hosts, typically hosts not connected by ICI.

- **Within a slice, TPUs are only connected to their nearest neighbors via ICI.** This means communication over ICI between distant chips in a slice needs to hop over the intervening chips first.
- **Weight matrices need to be padded to at least size 128** (256 on TPU v6) in all dimensions to fill up the MXU (in fact, smaller axes are padded to 128).
- **Lower precision matrix multiplication tends to be faster.** TPUs can do int8 or int4 FLOPs roughly 2x/4x faster than bfloat16 FLOPs for generations that support it. VPU operations are still performed in fp32.
- To avoid bottlenecking the TPU compute unit, we need to **make sure the amount of communication across each channel is proportional to its speed.**
- **Here are some specific numbers for our chips:**

Model	Pod size	Host size	HBM capacity/chip	HBM BW/chip (bytes/s)	FLOPs/s/chip (bf16)	FLOPs/s/chip (int8)
TPU v3	32×32	4×2	32GB	9.0e11	1.4e14	1.4e14
TPU v4p	16×16×16	2×2×1	32GB	1.2e12	2.75e14	2.75e14
TPU v5p	16×20×28	2×2×1	96GB	2.8e12	4.59e14	9.18e14
TPU v5e	16×16	4×2	16GB	8.1e11	1.97e14	3.94e14
TPU v6e	16×16	4×2	32GB	1.6e12	9.20e14	1.84e15

Host size refers to the topology of TPUs connected to a single host (e.g. TPU v5e has a single CPU host connected to 8 TPUs in a 4x2 topology). And here are interconnect figures:

Model	ICI BW/link (one-way, bytes/s)	ICI BW/link (bidi, bytes/s)
TPU v3	1e11	2e11
TPU v4p	4.5e10	9e10
TPU v5p	9e10	1.8e11
TPU v5e	4.5e10	9e10
TPU v6e	9e10	1.8e11

We include both one-way (unidirectional) bandwidth and bidi (bidirectional) bandwidth since unidirectional bandwidth is more true to the hardware but bidirectional bandwidth occurs more often in equations involving a full ring. [9]

[9] By bidi (bidirectional) bandwidth we mean the total bytes that can be sent along a single link in both directions, or equally, the total number of outgoing bytes from a single TPU along a particular axis, assuming we can use both links efficiently. This is true when we have a functioning ring, AKA when we have a wraparound connection on the particular axis. This occurs on inference chips when we have a full 16 axis, or on training chips (v*p) when we have an axis which is a multiple of 4. We prefer to use the bidirectional bandwidth because it appears frequently in calculations involving bidirectional comms.

PCIe bandwidth is typically around `1.5e10` bytes / second per chip [10], while DCN bandwidth is typically around `2.5e10` bytes / second per host. We include both unidirectional and bidirectional bandwidth for completeness. Typically bidirectional bandwidth is the more useful number when we have access to a full wraparound ring, while one-way bandwidth is more true to the hardware.

2.4 Worked problems

These numbers are a little dry, but they let you make basic roofline estimates for model performance. Let's work a few problems to explain why this is useful. You'll see more examples in Section 3.

Exercise 2.1 *[bounding LLM latency]*

Say you want to sample from a 200B parameter model in bf16 that's split across 32 TPU v4p. How long would it take to load all the parameters from HBM into the systolic array? *Hint: use the numbers above.*

Exercise 2.2 *[TPU details]*

Consider a full TPU v5e pod. How many total CPU hosts are there? How many TPU TensorCores? What is the total FLOPs/s for the whole pod? What is the total HBM? Do the same exercise for TPU v5p pod.

Exercise 2.3 *[PCIe operational intensity]*

Imagine we're forced to store a big weight matrix A of type bfloat16$[D, F]$, and a batch of activations x of type bfloat16$[B, D]$ in host DRAM and want to do a matrix multiplication on them. This is running on a single host, and we're using a single TPU v6e chip attached to it. You can assume $B \ll D$, and $F = 4D$ (we'll see in future chapters why these are reasonable assumptions). What is the smallest batch size B we need to remain FLOPs bound over PCIe? Assume PCIe bandwidth of `1.5e10` bytes / second.

Exercise 2.4 *[general matmul latency]*

Let's say we want to multiply a weight matrix int8[16384, 4096] by an activation matrix of size int8[B, 4096] where B is some unknown batch size. Let's say we're on 1 TPUv5e to start.

1. How long will this multiplication take as a function of B? *Hint: it may help to calculate how long it will take to load the arrays from HBM and how long the multiplication will actually take. Which is bottlenecking you?*
2. What if we wanted to run this operation out of VMEM? How long would it take as a function of B?

Exercise 2.5 *[ICI bandwidth]*

Let's say we have a TPU v5e `4x4` slice. Let's say we want to send an array of type `bfloat16[8, 128, 8192]` from `TPU{0,0}` to `TPU{3, 3}`. Let's say the per-hop latency for TPU v5e is $1\mu s$.

1. How soon will the first byte arrive at its destination?
2. How long will the total transfer take?

[10] Trillium (TPU v6e) has 32GB/s, about 4x higher than v5.

Exercise 2.6 *[pulling it all together (hard)]*

Imagine you have a big matrix **A**: `int8[128 * 1024, 128 * 1024]` sharded evenly across a TPU v5e 4x4 slice but offloaded to host DRAM on each chip. Let's say you want to copy the entire array to `TPU{0, 0}` and multiply it by a vector `bf16[8, 128 * 1024]`. How long will this take? *Hint: use the numbers above.*

2.5 Appendix

2.5.1 Appendix A: Let's talk about GPUs

Compared to TPUs, GPUs have a simpler communication model and a more complicated programming model.

Overview of the compute model:

- GPUs are conceptually similar to TPUs: they also function as an accelerator attached to a CPU. Many components are roughly analogous:

TPU	GPU
Tensor Core	SM ('Streaming Multiprocessor')
HBM	DRAM
VPU	Tensor Cores
VMEM	L1 Cache
ICI	NVLink/NVSwitch

- Compared to TPUs, GPUs have many more 'streaming multiprocessors' (an H100 has about 140), each of which can be seen as analogous to a TensorCore (which a TPU only has 1-2 of). Having more SMs makes computation more flexible (since each can do totally independent work) but also makes the hardware more complex to reason about.
- Each SM in an H100 has about 1024 CUDA Cores which perform SIMD scalar work (like a TPU VPU) and a small L1 cache used to speed data access and for register spilling. A section of the memory used for the L1 cache can also be declared as shared memory allowing access from any thread in the thread-block, and is used for user-defined caches, parallel reductions and synchronization, etc (similar to VMEM on a TPU).
- GPUs also have an additional L2 cache that is shared by all SMs. Unlike VMEM, this is hardware managed and optimizing cache hits is often important for performance.

Networking:

- Primary difference is that NVIDIA GPUs are typically in 'cliques' of 8-256 GPUs via switches (NVLink → NVSwitch), which allow for point-to-point communication between any GPU within that 'clique', but that means communication between more than 256 is significantly slower - this means training on more than 256 typically requires pipeline parallelism to scale, which is more complex (by contrast, PaLM was trained on two cliques of 3072 TPU chips each).

- For common neural net operations such as AllReduce, all-to-all connections do not hold an advantage (as the same communication patterns must occur regardless), but it does allow for storing MoE models across more GPUs and transmitting the experts around more efficiently.
- Each GPU requires a switch that costs similar to the GPU itself, making on chip interconnect like ICI cheaper.
- NVIDIA deep learning performance[11]
- NVSwitch[12]
- Very different Tensor Parallelism / Pipeline Parallelism transition point!

2.5.2 Appendix B: How does a systolic array work?

At the core of the TPU MXU is a `128x128` systolic array (`256x256` on TPU v6e). When fully saturated the systolic array can perform one `bfloat16[8,128] @ bf16[128x128] -> f32[8,128]`[13] multiplication per 8 clock cycles.

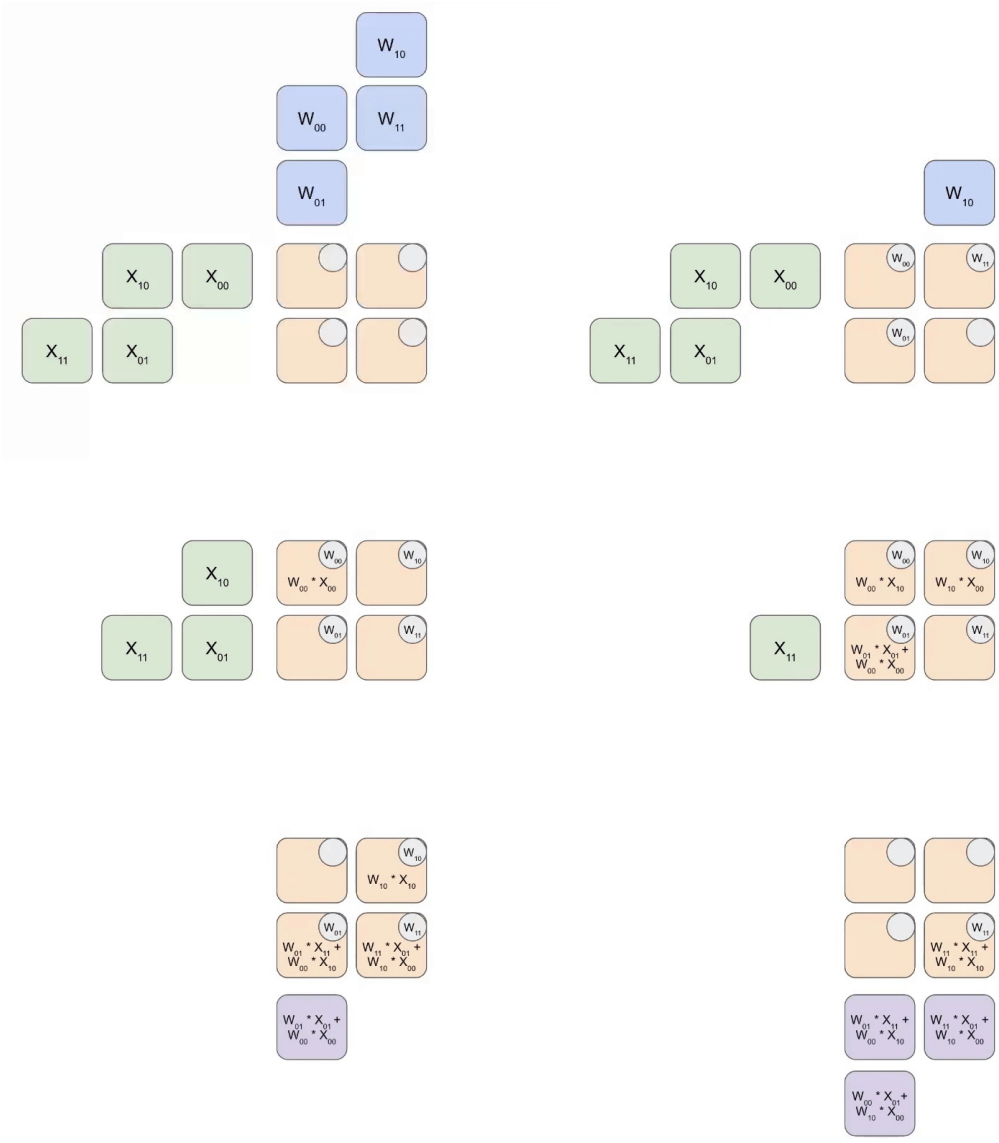
- At its core, the systolic array is a 2D `128x128` (`= 16,384`) grid of ALUs each capable of performing a multiply and add operation.
- Weights (**W**, the `128x128` input) are passed down from above (called the RHS) while inputs (**X**, the `8x128` input) are passed in from the left (called the LHS).

Here is a simplified animation of multiplying a set of weights (blue) with a set of activations (green). You'll notice that the weights (RHS) are partially loaded first, diagonally, and then the activations are fed in, also diagonally. In each frame below, we multiply all the overlapped green and blue units, sum the result with any residual passed in from above, and then pass the result in turn down one unit.

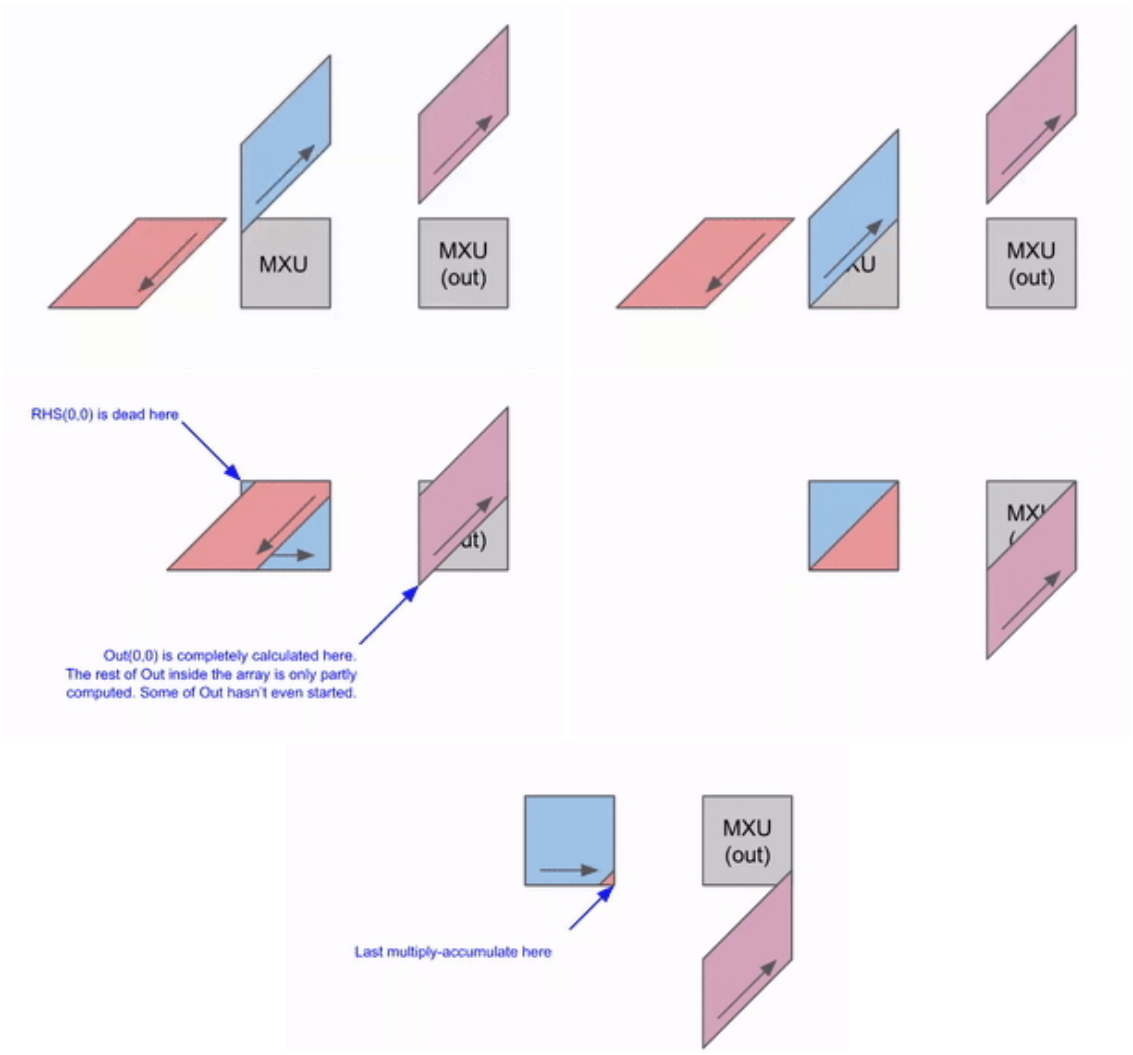
[11] `https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html#gpu-arch`

[12] `https://www.nvidia.com/en-au/data-center/nvlink/`

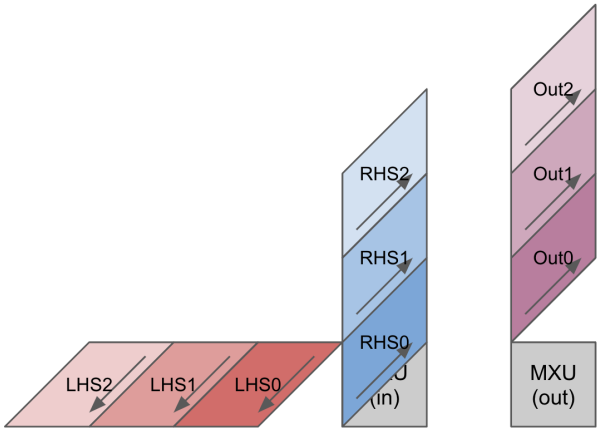
[13] If you are not familiar with this notation, it means: multiplying a `8x128` matrix with bfloat16 elements by a `128x128` matrix with bfloat16 elements and storing the results in a `8x128` matrix with float32 elements.



Here's a more general version of this animation showing the output being streamed out of computation:



Here's a diagram showing how this can be pipelined across multiple RHS and LHS arrays:



There is an initial pipeline bubble as the weights (RHS) and activations (LHS) are loaded. After that initial bubble, new inputs and weights can be loaded in without an additional bubble.

We can efficiently pipeline this to multiply large matrices without too large a pipeline bubble. With that said, it's important that our matrices have shapes larger than the side dimension of the MXU, which is generally 128x128. Some TPUs (since TPU v3) have multiple MXUs, either 2 for TPU v3 and 4 for TPU v4/5, so we need to ensure tiling dimensions are larger than 128 * number of MXUs.

Trillium (TPU v6e) has a `256x256` systolic array, which means it can perform 4x more FLOPs / cycle. This also means the dimensions of your tensors needs to be twice as large to utilize the MXU fully.

This blog post[14] has another excellent animation of a systolic array multiplication for a fixed weight matrix.

2.5.3 Appendix C: TPU internals

Scalar Core

The TPU scalar core processes all of the instructions and executes all of the transfers from HBM into vector memory (VMEM). The scalar core is also responsible for fetching instructions for the VPU, MXU and XLU components of the chip. One side-effect of this is that each core of the TPU is only capable of creating one DMA request per cycle.

To put this in context, a single 4 scalar core controls a VPU consisting of 2048 ALUs, 4 MXUs, 2 XLUs, and multiple DMA engines. The highly skewed nature of control per unit compute is a source of hardware efficiency, but also limits the ability to do data dependent vectorization in any interesting way.

VPU

The TPU vector core consists of a two dimensional vector machine (the **VPU**) that performs vector operations like vadd (vector addition) or vmax (elementwise max) and a set of vector registers called **VREGs** that hold data for the VPU and MXU. The VPU is effectively a 2D vector arithmetic unit of shape `(8, 128)` where the 128 dimension is referred to as a lane and the dimension of 8 is referred to as a sublane. Each (lane, sublane) pair on v4 contains 2 standard floating-point and integer ALUs. From a software point-of-view, this creates the appearance of a 8x128 vector unit with a total of 2048 floating point adders in v4. TPU v4 has 32 VREGs of size `(8, 128)` which the VPU loads from and writes to.

The VPU executes most arithmetic instructions in one cycle in each of its ALUs (like vadd or vector add) with a latency of 2 cycles, so e.g. in v5 you can add 4 pairs of f32 values together from VREGs in each cycle. A typical VPU instruction might look like `{v2 = vadd.8x128.f32 v0, v1}` where v0 and v1 are input VREGs and v2 is an output VREG.

All lanes and sublanes execute the same program every cycle in a pure SIMD manner, but each ALU can perform a different operation. So we can e.g. process 1 vadd and 1 vsub in a single cycle, each of which operates on two full VREGs and writes the output to a third.

[14] `https://fleetwood.dev/posts/domain-specific-architectures#google-tpu`

3 Sharded Matrices and How to Multiply Them

Here we'll explain how the biggest ML models are split (or "sharded") across multiple 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.

3.1 Partitioning Notation and Collective Operations

When we train an LLM on ten thousand TPUs, 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**, so we have to split them up.[1] We call this "*sharding*" or "*partitioning*" our arrays.
Here's an example 2D array **A** sharded across 4 TPUs:

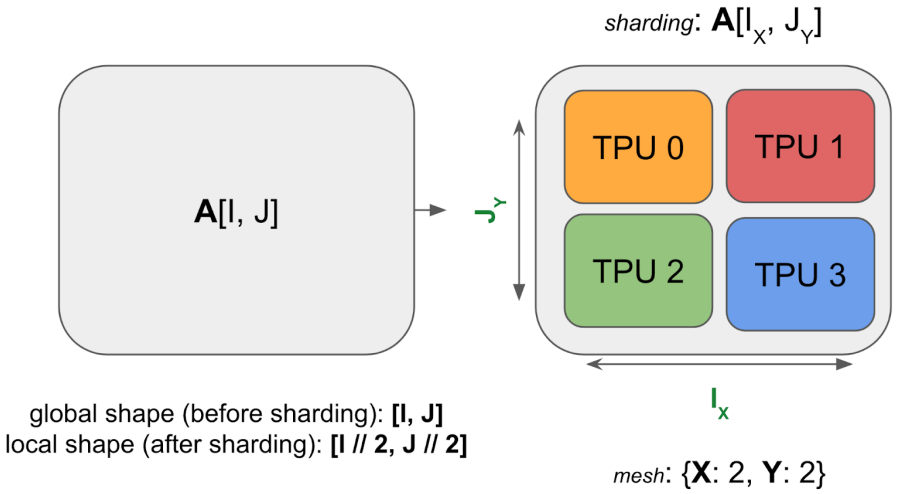


Figure 7: An example array of shape **A**[I, J] gets sharded across 4 devices. Both dimensions are evenly sharded across 2 devices with a sharding **A**[I_X, J_Y]. Each TPU holds 1/4 of the total memory.

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.

3.1.1 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**.

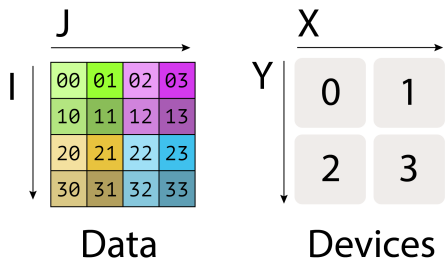
Example (the diagram above): For the above diagram, we have:

[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.

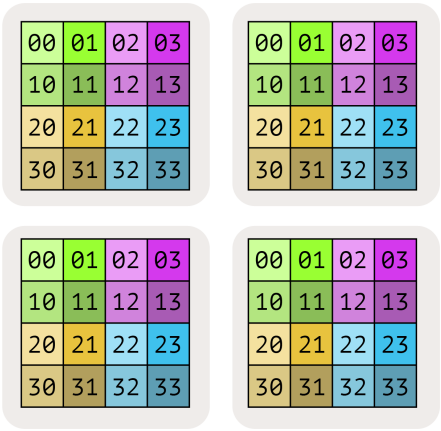
- **Sharding:** $A[I_X, J_Y]$, which tells us to shard the first axis, I, along the mesh axis X, and the second axis, J, along the mesh axis Y. This sharding tells us that each shard holds $1/(|X| \cdot |Y|)$ of the array.
- **Mesh:** the device mesh above `Mesh(devices=((0, 1), (2, 3)), axis_names=('X',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

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.
Example (2D sharding across 1 axis): $A[I_{XY}, J]$ shards the first dimension (I) along both the X and Y hardware axes. The number of bytes per device is the same as the previous sharding but the local shape is different. It is now $(|I|/(|X| \cdot |Y|), |J|)$.

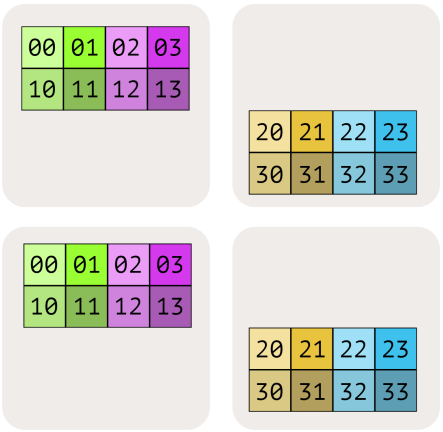
Visualizing these shardings: Let's try to visualize these shardings by looking at a 2D array of data split over 4 devices:



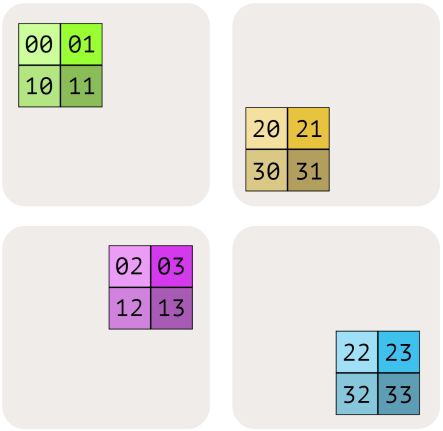
We write the *fully-replicated* form of the matrix simply as $A[I, J]$ with no sharding assignment. This means that *each* device contains a full copy of the entire matrix.



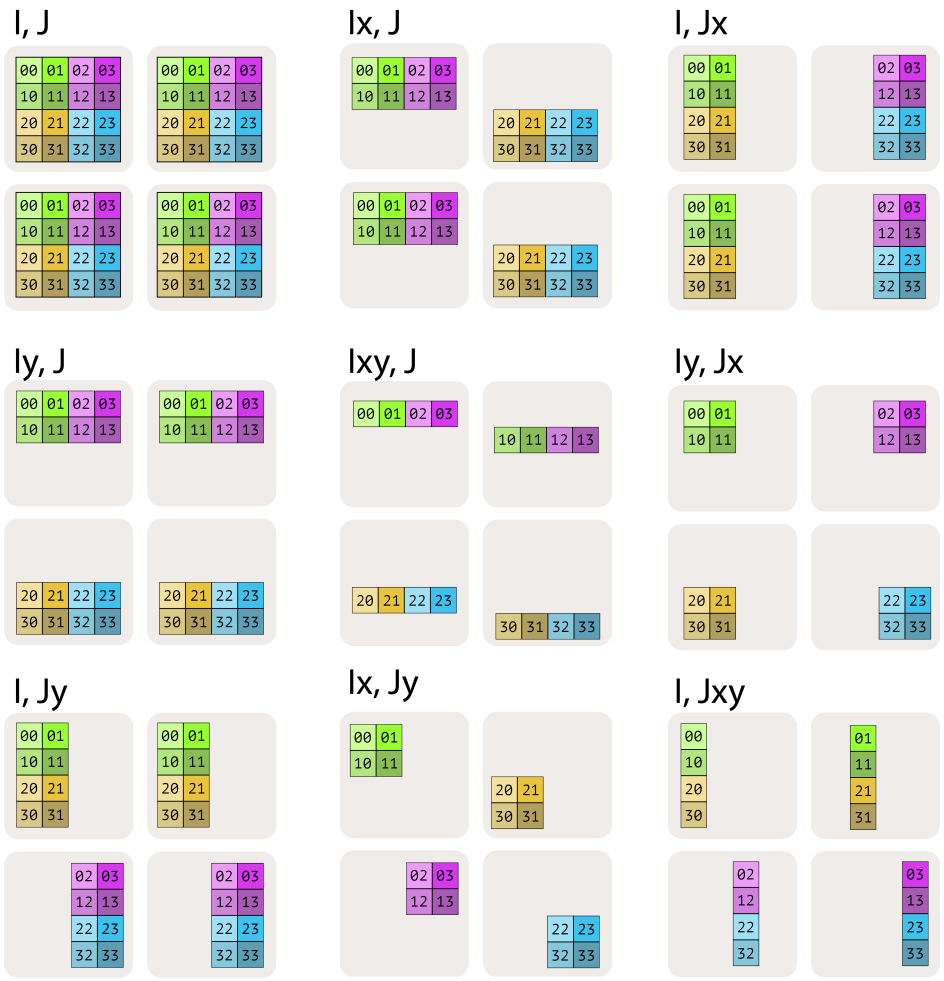
When we wish to indicate that one of these dimensions has been partitioned across a mesh axis, then we indicate so using a mesh-axis subscript. For instance $A[I_X, J]$ would mean that the **I** logical axis has been partitioned across the **X** mesh dimension, but that the **J** dimension is *not* partitioned,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



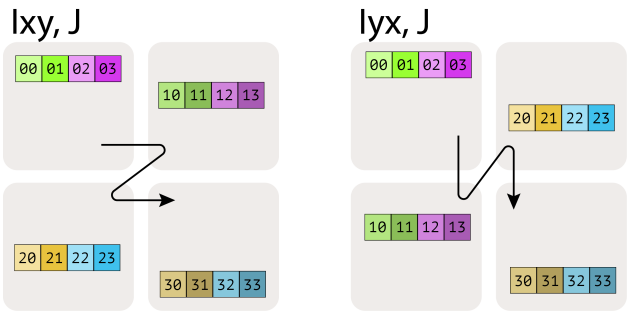
$A[I_X, J_Y]$ means that the **I** logical axis has been partitioned across the **X** mesh axis, and that the **J** dimension has been partitioned across the **Y** mesh axis.



We illustrate the other possibilities in the figure below:



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.



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?

Answer: Our array **A** is sharded over X and and Y and replicated over Z, so per device it has shape `int8[128 / (2 * 8), 2048] = int8[8, 2048]`, with size `8 * 2048 = 16,384` bytes. Because it's replicated over Z, while within a Z-plane it's fully sharded over X and Y, there's one copy of it per Z-plane, and 2 such planes, so the total size (across all devices) is `128 * 2048 * 2 = 512kiB` total.

3.1.2 A quick aside: how would we describe this in code?

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 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.

```
import jax
import jax.numpy as jnp
import jax.sharding as shd

# 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 shd.NamedSharding(mesh, shd.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.
compiled = jax.jit(lambda A, B: jnp.einsum('BD,DF->BF', A, B), out_shardings=P('X', 'Y')).lower
    (A, B).compile()
y = compiled(A, B)
```

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.

3.2 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. 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".

First let's recall the concept of a "block matrix", or a nested matrix of matrices:

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} = \left(\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} a_{02} & a_{03} \\ a_{12} & a_{13} \end{bmatrix} \right) = \begin{pmatrix} \mathbf{A_{00}} & \mathbf{A_{01}} \\ \mathbf{A_{10}} & \mathbf{A_{11}} \end{pmatrix}$$

Matrix multiplication has the nice property that when the matrix multiplicands are written in terms of blocks, the product can be written in terms of block matmuls following the standard rule:

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix} = \begin{pmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{pmatrix} \tag{3}$$

What this means is that implementing distributed matrix multiplications reduces down to moving these sharded blocks over the network, performing local matrix multiplications on the blocks, and summing their results. **The question then is what communication to add, and how expensive it is**.

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:

- **Case 1:** neither input is sharded along the contracting dimension. *We can multiply local shards without any communication.*
- **Case 2:** one input has a sharded contracting dimension. *We typically "AllGather" the sharded input along the contracting dimension.*
- **Case 3:** both inputs are sharded along the contracting dimension. *We can multiply the local shards, then "AllReduce" the result.*

- **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 partitioned tensors, the computation is valid and the output follows the sharding of the inputs *unless* the contracting dimension is sharded or both tensors have a non-contracting dimension 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:

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

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 Case 4 for details).

Case 2: neither multiplicand has a sharded contracting dimension

Let us consider the simple case of the distributed matrix multiply of **A** sharded in the contracting J dimension against a fully replicated **B**:

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

We cannot simply perform local matrix multiplies of the local **A**, **B** blocks against one another as we're missing the full data from the contracting axis of **A**. Typically, we first "**AllGather**" the shards of **A** together locally, and only then multiply against **B**:

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

AllGathers *remove 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.

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

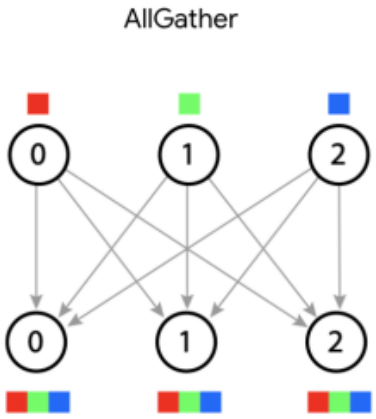
We also don't have to remove all subscripts for a given dimension, e.g. $A[I_{xy}, J] \rightarrow A[I_y, J]$ is also an AllGather, just over only a single axis.

Note that we may also wish to use an AllGather to remove *non-contracting* dimension sharding, for instance the matrix multiply:

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

We would similarly AllGather along **X** to remove the output sharding, however in this case we have the freedom of doing so before or after the matrix multiply, unlike in the case of AllGathering the contracting dimension, where we are forced to do so before performing the matrix multiply.

How is an AllGather actually performed? To perform an AllGather along a single axis, we need to pass all the shards around the axis until every device has a copy. Figure 1 shows an example. Each of the 8 devices starts with 1 / 8th of the array and ends up with all copies. One efficient way to do this is to have each device pass its shard around the sharding dimension ring, either in one direction or both directions. If we do one direction, it takes $N - 1$ hops of size total size$/N$ per-link, otherwise we have $\lceil \frac{N}{2} \rceil$ hops of size $2 \cdot$ total size$/N$ per link.



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

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

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

$$\begin{aligned} T_{total} &= \frac{2 \cdot V \cdot |X|}{2 \cdot |X| \cdot W_{\text{ICI}}} \\ &= \frac{V}{W_{\text{ICI}}} \end{aligned}$$

[2]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.

[3]technically, $\lceil |X|/2 \rceil$

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!

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 $|X|$.

Let T_{min} be the minimum time for a single hop. Then

$$T_{hop} = \max\left[T_{min}, \frac{2 \cdot V}{|X| \cdot W_{\text{ICI}}}\right]$$
$$T_{total} = \max\left[\frac{T_{min} \cdot |X|}{2}, \frac{V}{W_{\text{ICI}}}\right]$$

since we perform $|X|/2$ hops. For large reductions or gathers, we're solidly bandwidth bound. We're sending so much data that the overhead of each hop is essentially negligible. But for small arrays (e.g. when sampling from a model), this isn't negligible, and the ICI bandwidth isn't relevant. We're bound purely by latency. Another way to put this is that given a particular TPU, e.g. TPU v5e with `4.5e10` unidirectional ICI bandwidth, sending any buffer under `4.5e10 * 1e-6 = 45kB` will be latency bound.

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}.

In general we have

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

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 $\text{AllGather}_Y([E_Y, F])$ $\square$ [E, F] on a TPUv5e with a 2D mesh `{'X': 8, 'Y': 4}`, $E = 2048$, $F = 8192$ in bfloat16? How about with $E = 256$, $F = 256$.

Answer: Let's start by calculating some basic quantities:

1. TPU v5e has 4.5e10 bytes/s of unidirectional ICI bandwidth for each of its 2 axes.
2. In bfloat16 for (a), we have $A[E_Y, F]$ so each device holds an array of shape bfloat16[512, 8192] which has 512 * 8192 * 2 = 8.4MB. The total array has size 2048 * 8192 * 2 = 34MB.

For part (1), we can use the formula above. Since we're performing the AllGather over one axis, we have $T_{\text{comms}} = 34e6/9e10 = 377\mu s$. To check that we're not latency-bound, we know over an axis of size 4,

we'll have at most 3 hops, so our latency bound is something like 3us, so we're not close. However, TPU v5e only has a wraparound connection when one axis has size 16, so here *we actually can't do a fully bidirectional AllGather*. We have to do 3 hops for data from the edges to reach the other edge, so in theory we have more like $T_{\text{comms}} = 3 * 8.4e6/4.5e10 = 560\mu s$. An actual profile from the authors[4] shows $680\mu s$, which is reasonable since we're likely not getting 100% of the theoretical bandwidth! *For part (2)* each shard has size `64 * 256 * 2 = 32kB. 32e3 / 4.5e10 = 0.7us`, so we're latency bound. Since we have 3 hops, this will take roughly 3 * 1us = 3us. In practice, it's closer to 8us.[5]

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:

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

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:

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

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 $\otimes$ 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:

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

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:

$$\textbf{AllReduce}_Y\, A[I_x, J]\{U_Y\} \to A[I_x, J]$$

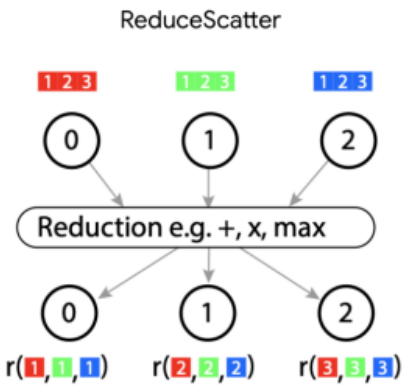
[4] https://imgur.com/a/RkvpRGQ
[5] https://imgur.com/a/HZLQmYs

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.

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

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 divided by the bandwidth, 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:

$$\textbf{AllGather}_X A[I_X, J] \to A[I, J]$$
$$A[I, J] \cdot B[J, K_X] \to C[I, K_X]$$

or

$$\textbf{AllGather}_X B[J, K_X] \to B[J, K]$$
$$A[I_X, J] \cdot B[J, K] \to C[I_X, K]$$

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.

3.3 A Deeper Dive into TPU Communicaton 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**.

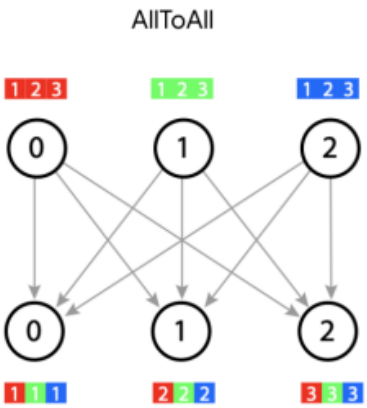
3.3.1 Our final communication primitive: the AllToAll

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.

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

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).[6]

[6] For even-sized bidirectional rings, each device will send $(N/2 + (N/2 - 1) + ... + 1)$ chunks right and $((N/2 - 1) + ... + 1)$ 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 (bytes$/N^2) \cdot N^2/4 =$ bytes$/4$. This result scales across all devices as the total bandwidth scales with device number.



3.3.2 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:

$$\textbf{AllGather}_X A[I_X] \to A[I]$$

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

$$\textbf{ReduceScatter}_X A'[I]\{U_X\} \to A'[I_X]$$

Likewise, $\text{ReduceScatter}_X(A[I]\{U_X\}) \to A[I_X])$ in the forward pass implies $\text{AllGather}_X(A'[I_X]) \to A'[I]$ 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:

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

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.

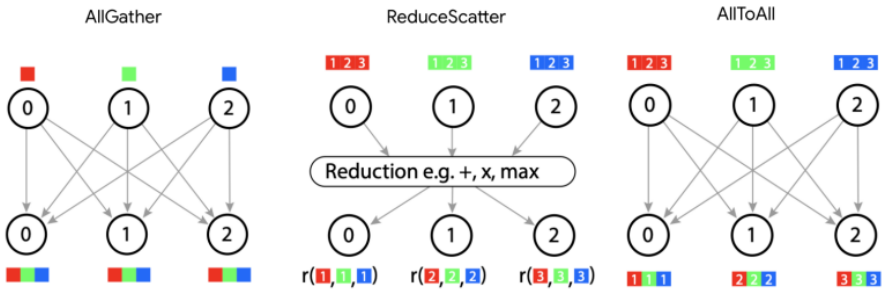
$$A[I, J_X] \cdot_{\text{LOCAL}} B[J_X, K] \to C[I, K]\{U_X\}$$
$$\textbf{ReduceScatter}_{X,K} C[I, K]\{U_X\} \to C[I, K_X]$$

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 $\textbf{ReduceScatter}_{X,K}$ to specify the axis to shard.

3.4 Key Takeaways

- 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:
  - *Neither array is sharded along the contracting dimension:* no communication is needed.
  - *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.
  - *Both arrays are identically sharded along the contracting dimension:* we multiply the shards locally then perform an AllReduce or ReduceScatter.
  - *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:**
  - AllGather: $[A_X, B] \rightarrow [A, B]$
  - ReduceScatter: $[A, B]\{U_X\} \rightarrow [A, B_X]$
  - AllToAll: $[A, B_X] \rightarrow [A_X, B]$
  - 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.	$[A_X, B] \to [A, B]$	bytes / (bidirectional ICI bandwidth * num_axes)
ReduceScatter	Sums a partially summed array along an axis and shards it along another axis (adding a subscript).	$[A, B]\{U_X\} \to [A_X, B]$	Same as AllGather
AllReduce	Sums a partially summed array along an axis. Removes a $\{U_x\}$. Combines an AllGather and ReduceScatter.	$[A_X, B]\{U_Y\} \to [A_X, B]$	2 * AllGather
AllToAll	Gathers (replicates) an axis and shards a different dimension along the same axis.	$[A, B_X] \to [A_X, B]$	AllGather/4 for a bidirectional ring

3.5 Worked problems

Exercise 3.1 *[replicated sharding]*

An array is sharded $A[I_X, J, K, ...]$ (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?

Exercise 3.2 *[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 $\text{AllReduce}_Z([B_X, D_Y]\{U_Z\})$?

Exercise 3.3 *[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.*

Exercise 3.4 *[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?

Exercise 3.5 *[minimum latency]*

Let's say I want to do a matmul $A[B, D] \cdot_D B[D, F] \to C[B, F]$ on a TPUv5p 4x4x4 with the lowest possible latency. How should my inputs be sharded? What is the total FLOPs and comms time?

Exercise 3.6

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?

1. 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.
2. What about $A[I_X, J] \cdot_J B[J, K_Y] \to C[I_X, K_Y]$? This is standard for inference, where do pure tensor parallelism (+ data).

Exercise 3.7

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?**

Exercise 3.8 *[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?

Exercise 3.9 *[another strategy for sharded matmuls?]*

In case 2, 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?

Exercise 3.10 *[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.

4 Transformers

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

4.1 Counting Dots

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

array	shape
x	[P]
y	[P]
A	[N P]
B	[P M]

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

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

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

4.1.1 Forwards and reverse FLOPs

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

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

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

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

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

is again $2NPM$ FLOPs since $\partial L / \partial C$ is a (co-)vector of size $[N, M]$. While this quantity isn't the derivative wrt. a parameter, it's used to compute derivatives for previous layers of the network (e.g. just as $\partial L / \partial C$ is used to compute $\partial L / \partial B$ above).

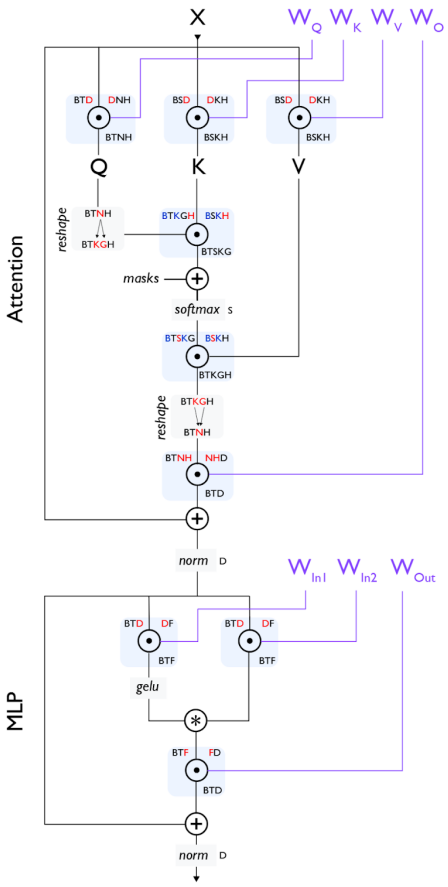
Adding these up, we see that **during training, we have a total of $6NPM$ FLOPs**, compared to $2NPM$ during inference: $2NPM$ in the forward pass, $4NPM$ in the backward pass. Since PM is the number of parameters in the matrix, this is the simplest form of the famous $6 * \text{num parameters} * \text{num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

4.2 Transformer Accounting

Transformers are the future. Well, they're the present at least. Maybe a few years ago, they were one of many architectures. But today, it's worth knowing pretty much every detail of the architecture. We won't reintroduce the architecture but this blog[1] and the original Transformer paper[2] may be helpful references.

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

[1] https://jalammar.github.io/illustrated-transformer/
[2] https://arxiv.org/abs/1706.03762



symbol	dimension
B	batch
L	number of layers
T	sequence length (query)
S	sequence length (key value)
V	vocab
D	d_model, embedding dimension
F	MLP hidden dimension
H	attention head dimension
N	number of query heads
K	number of key/value heads
G	q heads per kv head = N // K

Figure 8: this diagram shows one layer of a standard Transformer and flows from top-to-bottom. We use a single-letter convention to describe the shapes and layouts of arrays in a Transformer, again showing contracting dimensions in red, and batched dimensions in blue. In a given operation, the input shape is given on top-left and the parameter shape is given on the top-right, with the resulting shape below, e.g. $[B, T, D]$ is the input shape for the gating einsum and $[D, F]$ is the weight shape.

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

Note 2 [MHA attention]: With self-attention, T and S are the same but for cross-attention they may be different. With vanilla Multi-Head Attention (MHA), N and K are the same while for Multi-Query Attention (MQA)[4] K=1 and for Grouped MQA (GMQA)[5] K merely has to divide N.

[3] https://arxiv.org/abs/2002.05202
[4] https://arxiv.org/abs/1911.02150
[5] https://arxiv.org/abs/2305.13245

4.3 Global FLOPs and Params Calculation

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

MLPs

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

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

Attention

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

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

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

the batched dims in blue:

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

Other Operations

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

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

General rule of thumb for Transformer FLOPs

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

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

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

Fractional cost of attention with context length

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

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

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

4.4 Miscellaneous Math

4.4.1 Sparsity and Mixture-of-Experts

We'd be remiss not to briefly discuss Mixture of Experts (MoE) models which replace the single dense MLP blocks in a standard Transformer with a set of independent MLPs that can be dynamically routed between. To a first approximation, **an MoE is just a normal dense model with E MLP blocks per layer**, instead of just one. Each token activates k of these experts, typically $k = 2$. This increases the parameter count by $O(E)$, while multiplying the total number of activated parameters per token by k, compared with the dense version.

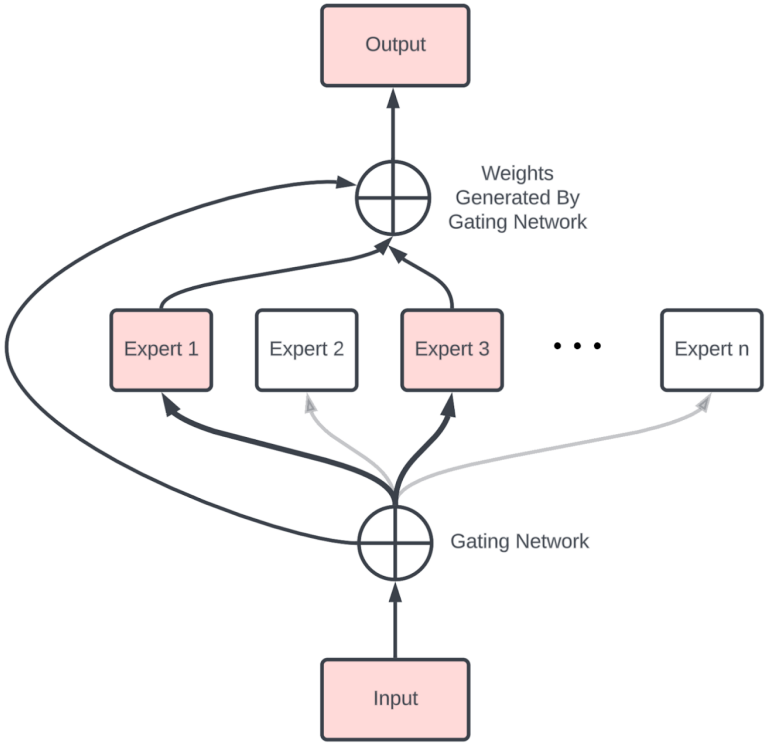


Figure 9: an example MoE layer with n experts. The gating expert routes each token to k of them, and the output of those k MLPs get summed. Our parameter count is n times the size of each expert, but only k are used for each token. Source: https://deepgram.com/learn/mixture-of-experts-ml-model-guide

Compared to a dense model, an MoE introduces new comms, primarily two AllToAlls (one before and one after the MoE block) that route tokens to the correct expert and bring them back to their home device.[6] However as we saw in the previous section, the cost of each AllToAll is only 1/4 that of a comparable AllGather along a single axis (for a bidirectional ring).

4.4.2 Gradient checkpointing

Backpropagation as an algorithm trades memory for compute. Instead of a backward pass requiring $O(n^2_{\text{layers}})$ FLOPs, **it requires $O(n_{\text{layers}})$ memory**, saving all intermediate activations generated during the forward

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

pass. While this is better than quadratic compute, it's incredibly expensive memory-wise: a model with $B * T = 4M$ (4M total tokens per batch), $L = 64$, and $D = 8192$ that avoids all unnecessary backward pass compute would have to save roughly $2 * 20 * B * T * D * L = 84TB$ of activations in bfloat16. 20 comes from (roughly) counting every intermediate node in the Transformer diagram above, since e.g.

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

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

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

This by no means comprehensive. When using JAX, these are typically controlled by `jax.remat`/`jax.checkpoint`.

4.4.3 Key-Value (KV) caching

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

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

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

4.5 Key Takeaways

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

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

- The parameter count of the MLP block dominates the total parameter count and the MLP block also dominates the FLOPs budget as long as the sequence length $T < 8D$
- The total FLOPs budget during training is well approximated by $6 \cdot$ num_params $\cdot$ num_tokens for reasonable context lengths.
- During inference, our KV caches are roughly $2 \cdot S \cdot L \cdot N \cdot H$ per cache, although architectural modifications can often reduce this.

4.6 Worked Problems

Exercise 4.1

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

Exercise 4.2

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

Exercise 4.3

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

Exercise 4.4

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

Exercise 4.5

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

Exercise 4.6

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

Exercise 4.7

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

[7] `https://arxiv.org/pdf/2412.19437v1`

Exercise 4.8

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

4.7 Appendix

4.7.1 Appendix A: How does Flash Attention work?

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

1. As we noted in Section 4, even though this is quadratic, the attention FLOPs only dominated when $S > 8 \cdot D$, and especially during training the memory of a single attention matrix is small compared to all of the weights and activation checkpoints living in memory, especially when sharded.
2. We don't need to materialize the full attention matrix in order to compute attention! We can compute local sums and maxes and avoid ever materializing more than a small chunk of the array. While the total FLOPs is still quadratic, we drastically reduce memory pressure.

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

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

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

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

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

$$L^1 = \sum_i \exp(Q \cdot K_i^1 - \max_j Q \cdot K_j^1)$$
$$L^2 = \sum_i \exp(Q \cdot K_i^2 - \max_j Q \cdot K_j^1)$$

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

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

where:

$$M^1 = \max_j Q \cdot K_j^1$$
$$M^2 = \max_j Q \cdot K_j^2$$

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

Algorithm 1 FlashAttention

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

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

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

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

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

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

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

During the backpropagation of this gradient to Q and K

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

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

local contraction along the feature **depth** dimension.

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

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

5 Training

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.

5.1 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*". This is difficult to achieve 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*.[1]

In this section, we'll discuss four common parallelism schemes: (pure) data **parallelism**, **fully-sharded data parallelism** (FSDP / ZeRO sharding),**tensor 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.[2] We'll use the following notation to simplify calculations throughout this section.

Notation	Meaning (model parameters)
D	d_{model} (the hidden dimension/residual stream dim)
F	d_{ff} (the feed-forward dimension)
B	Batch dimension (total number of tokens in the batch)
T	Sequence length
L	Number of layers in the model

Notation	Meaning (hardware characteristic
C	FLOPS per chip
W	Network bandwidth (bidirectional, often subscripted; e.g. W_{ICI})
X	Number of chips along a mesh axis
Y	Number of chips along an alternate mesh axis
Z	Number of chips along a third mesh axis

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:

[1] 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.

[2] 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.

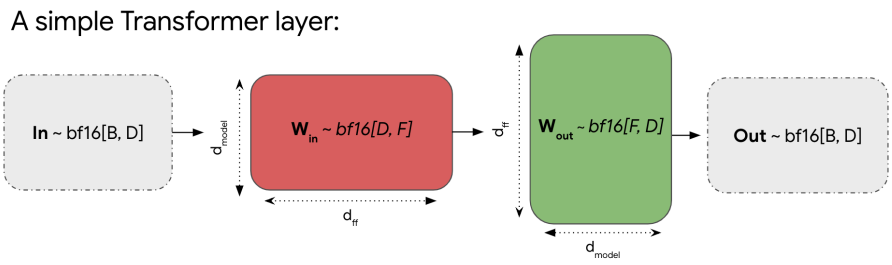


Figure 10: a simplified Transformer layer. We treat each FFW block as a stack of two matrices W_{in}: `bf16[D, F]` (up-projection) and W_{out}: `bf16[F, D]` (down-projection) with an input **In**: `bf16[B, D]`.

Here 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_{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:*

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

5.1.1 Data Parallelism

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

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 performs an **AllReduce on their gradients in order to synchronize them before updating the parameters.**

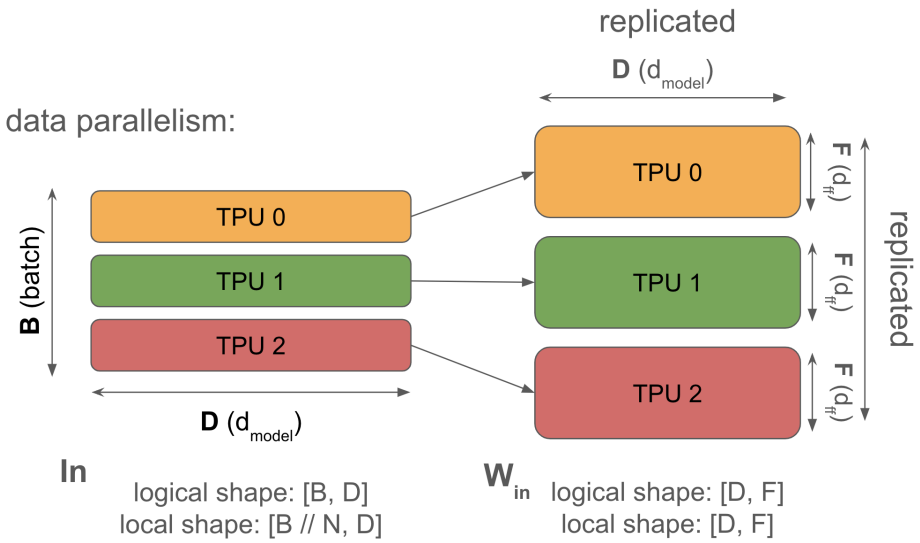


Figure 11: a diagram of pure data parallelism (forward pass). Our activations (left) are fully sharded along the batch dimension and our weights are fully replicated, so each TPU has an identical copy of the weights. This means the total memory of our weights is increased by a factor of N, but no communication is required on the forward-pass

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.

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)*

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[3], the largest model we can fit has TPU memory$/10$ parameters, so e.g. on a TPUv5p pod 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 num_params = HBM per device$/10$. For TPU v5p this is roughly 9B parameters.[4]

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_{ici} = **bidirectional** network bandwidth, and X = number of shards across which the batch is partitioned[5]. Let's calculate the time required to perform the relevant matmuls, T_{math}, and the required communication time T_{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_{ici}; specifically the AllReduce time is $2 \cdot \text{total bytes}/W_{\text{ici}}$. Since we need to AllReduce for both W_{in} and W_{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:

$$T_{\text{comms}} = \frac{2 \cdot 2 \cdot 2 \cdot D \cdot F}{W_{\text{ici}}}.$$

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

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

[3] 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.

[5] We assume this partitioning is done over an ICI mesh, so the relevant network bandwidth is W_{ici}

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

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

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

$$\frac{B}{X} > \frac{C}{W_{\text{ici}}}.$$

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_{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 $B > C/W_{\text{ici}}$ 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_{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 on context parallelism: throughout this section, we use B to refer to the total batch size in tokens. Clearly, however, our batch is made up of K sequences of T tokens each, so how can we do this? As far as the MLP is concerned, *tokens are tokens*! It doesn't matter if they belong to the same batch or two different batches. 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.

5.1.2 Fully-Sharded Data Parallelism (FSDP)

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

Fully-sharded data parallelism (often called FSDP or ZeRO-sharding) 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.**

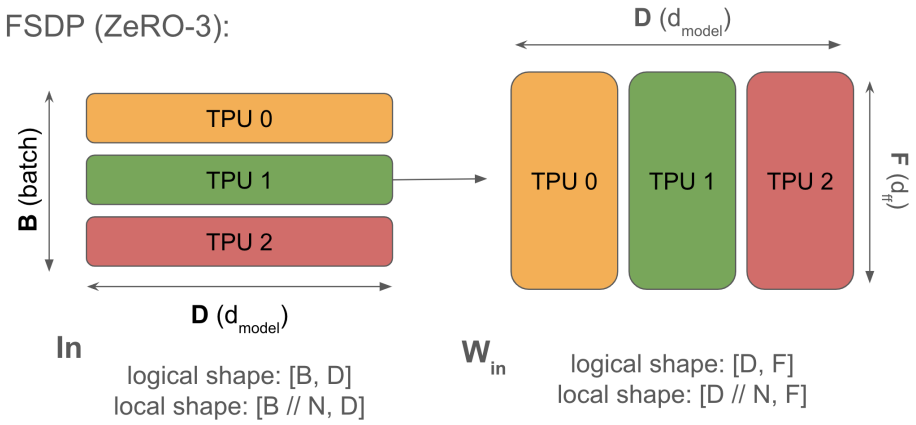


Figure 12: FSDP shards the contracting dimension of Win and the output dimension of Wout along the data dimension. This reduces memory but (from Section 3) requires us to gather the weights for W before we perform the matmul. Note that the activations (left) *are not sharded along the contracting dimension*, 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 algorithm:

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[6], 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:

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

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_{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 $2550/n_{\text{axes}}$.

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 approxi-**

[6]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.

mately 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.

5.1.3 Tensor Parallelism

Syntax: $\text{In}[B, D_Y] \cdot_D W_{in}[D, F_Y] \cdot_F W_{out}[F_Y, D] \rightarrow \text{Out}[B, D_Y]$ (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. This can unlock a smaller efficient batch size per pod. The figure below shows an example of a single matrix sharded in this way:

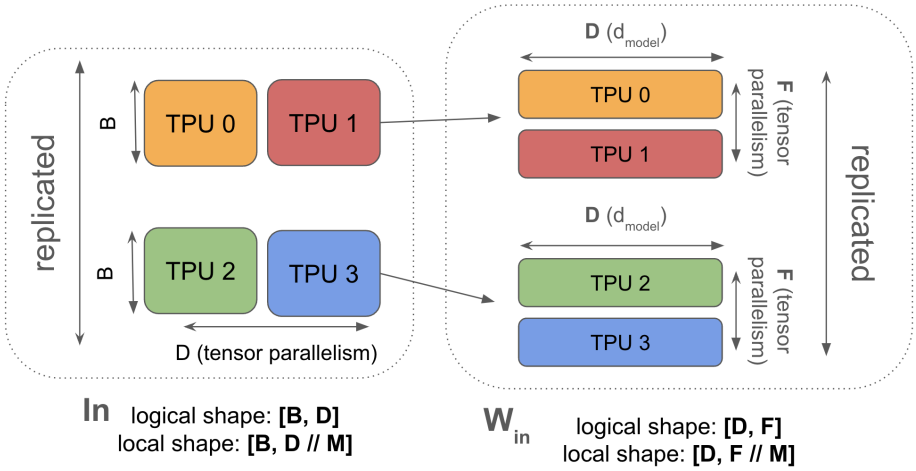


Figure 13: an example of basic tensor parallelism. Since we're only sharding our activations over Y (unlike in FSDP where we shard over X), we replicate our activations over X. Using our standard syntax, this is **A**[B, D_Y] * **B**[D, F_Y] -> **C**[B, F_Y]. Because we're only sharding over one of the contracting dimensions, we typically AllGather the activations **A** before the matmul.

As noted, **$\text{In}[B, D_Y] *_D W_{in}[D, F_Y] *_F W_{out}[F_Y, D]$ $\text{Out}[B, D_Y]$ 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 model 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 model 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.

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

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

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

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

$$\frac{4 \cdot B \cdot D \cdot F}{Y \cdot C} > \frac{2 \cdot 2 \cdot (B \cdot D)}{W_{\text{ici}}} \frac{F}{Y \cdot C} > \frac{1}{W_{\text{ici}}}$$

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

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

Takeaway: model parallelism becomes communication bound when $Y > n_{\text{axes}} * F/2550$. For most models this is between 8 and 16-way model parallelism.

Note that this doesn't depend on the precision of the computation, since e.g. for int8, on TPUv5p, $C_{\text{int8}}/W_{\text{ICI}}$ 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 TPUv4p with LLaMA 3-70B with $D = 8192$, $F \approx 30,000$, we can comfortably do 8-way model parallelism, but will be communication bound on 16 way model parallelism. The required F for model 8 way model sharding is 20k.
- For Gemma 7B, $F \approx 50k$, so we become communication bound with 19-way model parallelism. That means we could likely do 16-way and still see good performance.

5.1.4 Mixed FSDP and Tensor Parallelism

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

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.

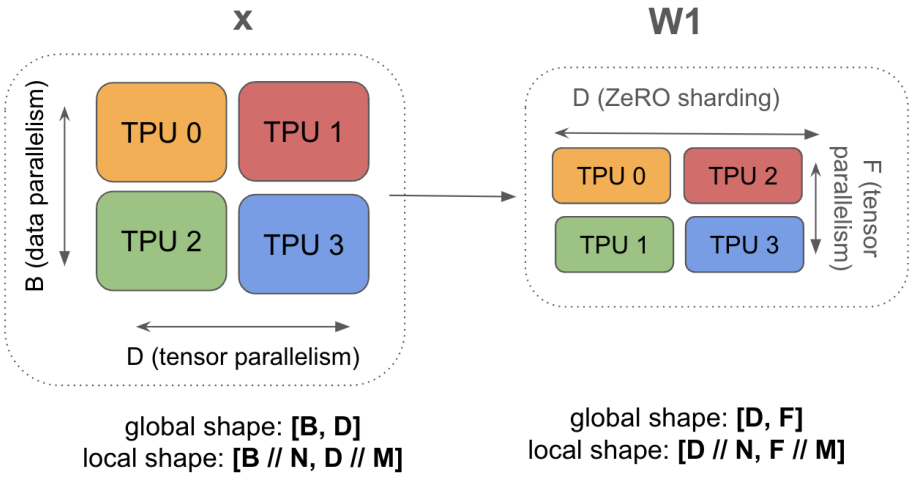


Figure 14: a diagram combining FSDP and tensor parallelism. Unlike the other cases, there is no duplication of model parameters.

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!

Mixed FSDP/model parallelism:

Forward pass: need to compute Loss[B]
1. In[B_X, D] = **AllGather**$_Y$(In[B_X, D_Y]) *(on critical path)*
2. W_{in}[D, F_Y] = **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. W2[F_Y, D] = **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] = **ReduceScatter**$_Y$(Out[B_X, D] {U_Y}) *(on critical path)*
7. Loss[B_X] = ...

Backward pass: need to compute dW_{out}[F_Y, D_X], dW_{in}[D_X, F_Y]
1. dOut[B_X, D_Y] = ...
2. dOut[B_X, D] = **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] = **ReduceScatter**$_X$(dW_{out}[F_Y, D] {U_X})
5. W_{out}[F_Y, D] = **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] = **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] = **ReduceScatter**$_X$(dW_{in}[D, F_Y] {U_X})
10. W_{in}[D, F_Y] = **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] = **ReduceScatter**$_Y$(dIn[B_X, D] {U_Y}) *(on critical path)*

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

- Model parallelism performs $\mathbf{AllGather}_Y([B_X, D_Y])$ 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 MP 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 MP 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

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

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

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 simplifications: 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 do not overlap comms on the X and Y axis. Under the second assumption, the total comms time is

$$T_{\text{comms}} = T_{\text{FSDP comms}} + T_{\text{MP comms}}.$$

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_{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:

$$T_{\text{comms}}(X) = \frac{F \cdot X}{N \cdot M_X} + \frac{B}{X \cdot M_Y}$$

Differentiating this expression wrt X and setting the derivative equal to zero gives the optimal value X_{opt}:

$$\frac{d}{dX} T_{\text{comms}}(X_{opt}) = \frac{F}{N \cdot M_X} - \frac{B}{X_{opt}^2 \cdot M_Y} \rightarrow$$

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

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), $B = 48,000$, $F = 32,768$, 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:

$$T_{\text{FSDP comms}} + T_{\text{MP comms}} < T_{\text{math}}$$

which gives us:

$$\frac{2 \cdot 2 \cdot D \cdot F}{Y \cdot W_{\text{ici}} \cdot M_X} + \frac{2 \cdot 2 \cdot B \cdot D}{X \cdot W_{\text{ici}} \cdot M_Y} < \frac{2 \cdot 2 \cdot B \cdot D \cdot F}{N \cdot C}$$

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

$$\frac{F}{Y \cdot M_X} + \frac{B}{X \cdot M_Y} < \frac{B \cdot F}{N \cdot \alpha}$$

Plugging in our calculated X_{opt} into the equation above (and noting that $Y_{opt} = N / X_{opt}$) results in the following condition on the batch size B:

$$\sqrt{\frac{4 \cdot B \cdot F}{M_X \cdot M_Y \cdot N}} < \frac{B \cdot F}{N \cdot \alpha}$$

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:

$$\frac{T_{\text{math}}}{T_{\text{comms}}} = \frac{\sqrt{BF}\sqrt{M_X M_Y}}{2\alpha\sqrt{N}}.$$

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

$$\frac{B}{N} > \frac{4\alpha^2}{M_X M_Y F}$$

See Appendix C for an alternate derivation of this relation. 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 $B/N > 400$. This roughly wins us a factor of two 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 $2 \cdot 2550^2/F$. This lets us handle a batch of as little as 400 per chip, which is roughly a factor of two smaller than we could achieve with just FSDP.

Below we plot the ratio of FLOPs to comms time for mixed FSDP + MP, comparing it both to only model parallelism 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 400 and 850, a mixed FSDP + MP strategy is required in order to be compute-bound.

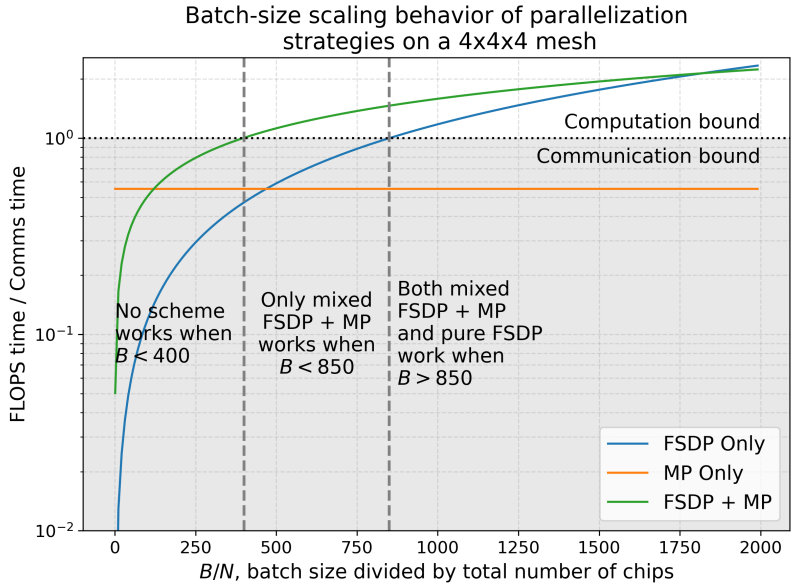


Figure 15: Ratio of FLOPs to comms time for optimal mixed FSDP/MP on a TPUv5p 4x4x4 slice with F=30k. As expected, model parallelism has a fixed ratio with batch size; ideal mixed FSDP + MP scales with $\sqrt{B}$, and FSDP scales with B. However, in intermediate batch size regimes, only FSDP + MP achieves 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.

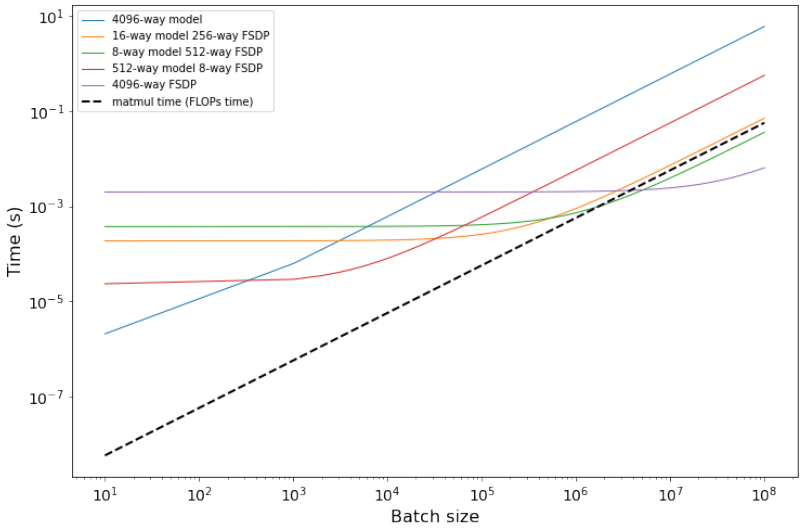
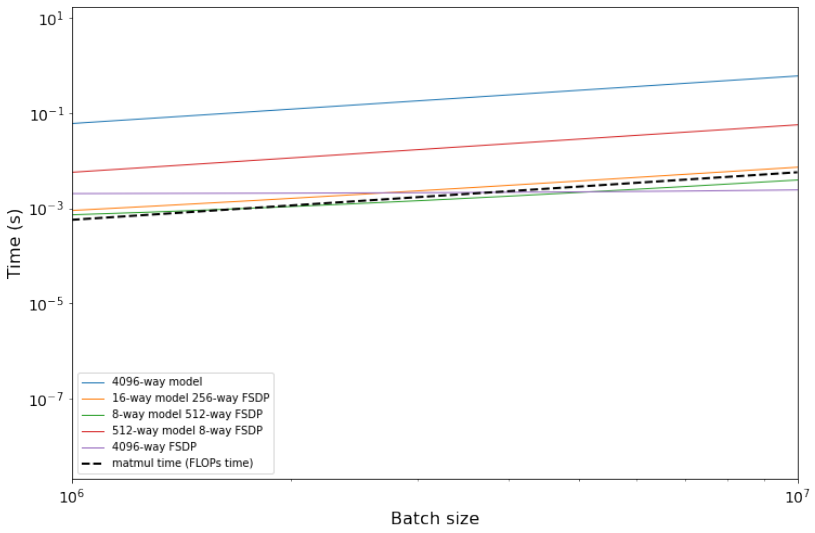


Figure 16: Time taken for communication with different parallelism schemes. The black dashed line is the time taken by the matrix multiplication FLOPs, so any curve above this line is comms-bound. We note that all strategies become comms-bound below batch size `1.5e6`, which is in line with our expected `4096 * 2 * 2550^2 / (8192 * 4) = 1.6e10`.

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 `1.6e10`, as predicted.

Zooming in, we can see that devoting two axes to FSDP, and using the optical switches to reconfigure the topology to have an 8-long axis for model sharding will give us the lowest communication volume between 1M and 6M batch size per slice, while pure FSDP combination is best between 6M and 100M. This agrees with our calculations above!



In the online version of this book, there is an interactive animation to play with this, showing the total compute time and communication time for different batch sizes.

5.1.5 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 ($W_{in}[L_Z, D_X, F_Y]$ 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.

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(0, num_layers, -1):
  _, f_vjp = jax.vjp(layer_fn, intermediates[i + 1], 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 showing their "bubble-free" pipeline schedule:

Figure 17: 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.

5.1.6 Scaling Between Pods

Let's take a step back and look at a specific example, say training LLaMA-3 70B on TPU v5p. LLaMA-3 70B has $F \approx 30,000$. From the above sections, we know the following:

- We'll be ICI bound when we do model parallelism greater than $Y > n_{\text{axes}} * F/2550 \cong n_{\text{axes}} * 11$.
- Pure FSDP becomes ICI bound when we have a batch size $< 2550/n_{\text{axes}}$. Here that means if we wanted to train with BS=2M, we'd at most be able to use ≈ 2400 chips, which is roughly a quarter of a TPU v5p pod.
- Mixed FSDP + model parallelism becomes ICI bound when we have batch size $< 2 \cdot 2550^2/30,000 = 432$, so this lets us scale to roughly 9k chips! However, the maximum size of a TPU v5p pod is 8k chips, and beyond that we have to scale to lower-bandwidth data-center networking (DCN).

So this gives us a nice recipe to fit on a single pod with BS=3.5M. We'd use the equation above, which gives roughly X (FSDP) = 1024 and Y (MP) = 8. If the model was larger, there would be room to expand the model sharding to 16. We have a bit of room to drop the batch size as low as BS=1.5M on that pod and still be compute bound, but we're close to the lower bound there.

To go larger than one pod, we need to scale over DCN. Because DCN has lower bandwidth, it's typically too slow to do much useful FSDP. Instead, we do pure data parallelism over the DCN axis and FSDP within a pod. Lets calculate whether the Data Center Network (DCN) holds up.

With pure data parallelism over DCN, we need to sync the weights and optimizer states during each step (as the model completes its backward pass we need to complete the AllReduce). We can actually just borrow the math from the pure data parallelism section above which tells us that we become comms bound when the per pod batch size $< C_{\text{pod}}/W_{\text{DCN}}$ where the RHS here is the total compute and total bandwidth for the entire pod.

- Our total DCN ingress+egress bandwidth is `2.5e10` per host, with 4 chips per host. This gives us 2000 hosts in the slice, and a total of `5e13` bytes of bandwidth.
- C_{pod} here is the pod size times the per-chip compute, which is `8k * 4.5e14 = 3.8e18` FLOPs.

As before, we become bottlenecked when $T_{\text{math}} < T_{\text{comms}}$ which happens when our per pod batch size $< C/W_{\text{DCN}} = 3.8e18/5e13 = 76,000$ (our pod level DCN operational intensity). For LLaMA-3, that's not going to be a problem since our per-pod batch size is much higher than that, but it could become an issue if we were to train on smaller slices (e.g. v5e).

> **Takeaway:** This means we can scale fairly arbitrarily across pods, so e.g. with 10 pods of 8960 chips we could do a global batch size of about 40M tokens on 89,600 chips, training LLaMA-3 70B in about 2 days.

5.2 Key Takeaways

- 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, model 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.
Model Parallelism (aka Megatron, Tensor)	Activations are sharded along d_{model}, weights are sharded along d_{ff}, activations are gathered before W_{in}, the result reduce-scattered after W_{out}
Mixed FSDP + Model Parallelism	Both of the above, where FSDP gathers the model sharded weights.

And here are the "formulas" for each method:

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

- 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 model parallelism.

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

- 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 batch size per shard $< C/W$, the arithmetic intensity of the network. For ICI this is 2,550 and for DCN this is 75,000. This can be increased with more parallel axes.
- Model 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 + model parallelism allows us to drop the batch size to as low as $2 \cdot 2550^2/F \approx 400$. This is fairly close to the point (200) where we become HBM bandwidth bound anyway.

- Data parallelism across pods requires a minimum batch size per pod of roughly 75,000 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.

5.3 Worked Problems

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

hyperparam	value
n_{layers} (L)	40
d_{model} (D)	5,120
$\text{ffw}_{\text{multiplier}}$ (F // D)	2.7
n_{heads} (N)	40
$n_{\text{kv\_heads}}$ (K)	40
d_{qkv} (H)	128
$n_{\text{embeddings}}$ (V)	32,000

Exercise 5.1

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.*

Exercise 5.2

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).*

Exercise 5.3

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 + model 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?

Exercise 5.4 *[Scaling LLAMA 70B to more chips]*

What if we wanted to drop to batch size 1M? How does this affect the answers to question 3? What about batch size 10M?

5.4 Appendix

5.4.1 Appendix A - More stuff about FSDP

Here's a nice extra figure showing how FSDP shards parameters/gradients. The rows are, in order, pure data parallelism, ZeRO-1/2/3. There's not much reason not to do ZeRO-3 since it has effectively the same communication load.

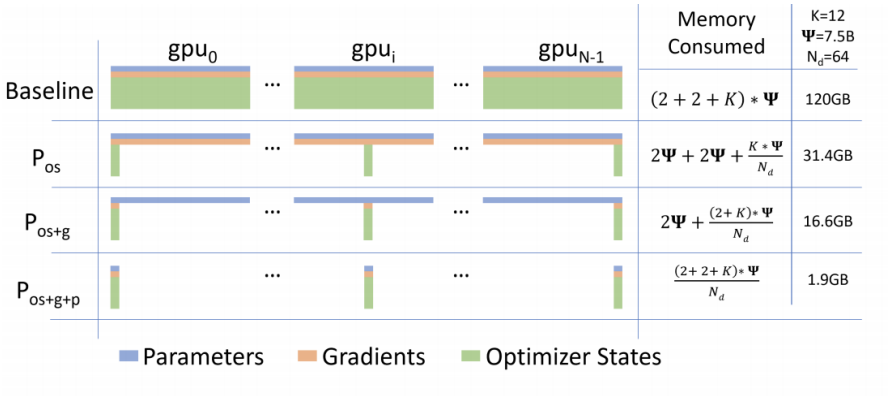


Figure 1: Comparing the per-device memory consumption of model states, with three stages of *ZeRO*-DP optimizations. Ψ denotes model size (number of parameters), K denotes the memory multiplier of optimizer states, and N_d denotes DP degree. In the example, we assume a model size of $\Psi = 7.5B$ and DP of $N_d = 64$ with $K = 12$ based on mixed-precision training with Adam optimizer.

5.4.2 Appendix B - Deriving the comms necessary for the backward passes

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} = dTmp[B, F] $*_B$ Tmp[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.

5.4.3 Appendix C - Alternate derivation of the batch size constraint for mixed FSDP + model parallelism

Above we derived that when using a combination of FSDP + model parallelism, we can be compute-bound when

$$\frac{B}{N} > \frac{4\alpha^2}{M_X M_Y F}$$

Here we present an alternate derivation of this fact. We start by setting the communication time equal to the computation time, and look for a condition which makes this equality impossible.

$$\frac{F}{Y \cdot M_X} + \frac{B}{X \cdot M_Y} = \frac{B \cdot F}{N \cdot \alpha}$$

Since $XY = N$, we can rewrite in terms of X:

$$\frac{FX}{N \cdot M_X} + \frac{B}{X \cdot M_Y} = \frac{B \cdot F}{N \cdot \alpha}$$

or

$$X^2 \frac{F}{N \cdot M_X} + \frac{B}{M_Y} - X \frac{B \cdot F}{N \cdot \alpha} = 0$$

As this is a quadratic in X, the point at which we'll have no solutions is the point at which the discriminant becomes zero. This occurs when

$$B^2 \cdot F^2 \cdot M_X^2 \cdot M_Y^2 - 4 \cdot \alpha^2 \cdot F \cdot B \cdot N \cdot M_Y \cdot M_X = 0$$

or by simplifying:

$$B \cdot F \cdot M_X \cdot M_Y - 4 \cdot \alpha^2 \cdot N = 0$$

which gives us:

$$B = \frac{4 \cdot \alpha^2 \cdot N}{F \cdot M_X \cdot M_Y}$$

so our total batch size divided by the total number of chips cannot drop below:

$$\frac{4\alpha^2}{F \cdot M_X \cdot M_Y}$$

as we had derived above.

6 Training LLAMA 3 on TPUs

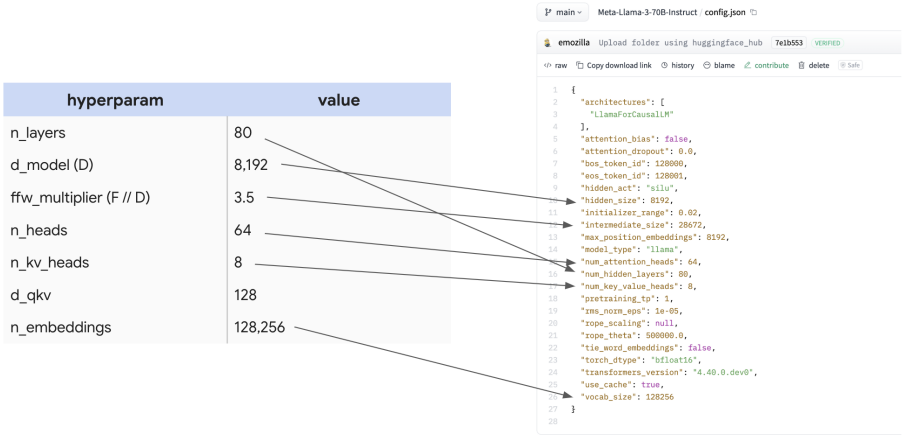
Our goal in this section is to apply results from the previous section to a very practical problem: training the LLaMA 3 family (herd) of models. Unlike the previous sections we want you to do a lot of this work yourself. For this reason, we've hidden the answers to each section so you can try to answer it first. Try grabbing a pen and doing by hand!

6.1 What does LLAMA 3 look like?

The LLaMA-3 model family includes 3 main models: LLaMA 3 8B, 70B, and 405B. We'll mostly focus on 70B, and leave 8B and 405B for you to explore in the problem section at the end. Here's the architecture for LLaMA 3-70B, taken from the LLaMA HuggingFace page[1].

hyperparam	value
n_{layers} (L)	80
d_{model} (D)	8,192
d_{ff} (F)	28,672
n_{heads} (N)	64
$n_{\text{kv\_heads}}$ (K)	8
d_{qkv} (H)	128
$n_{\text{embeddings}}$ (V)	128,256

To highlight how easy this is to find, here's the config itself, along with a mapping:



It's useful to make a big table with these numbers for many different open-source LLMs, so you can quickly compare the design decisions they've made.

[1] https://huggingface.co/meta-llama/Meta-Llama-3-70B/blob/main/config.json

6.2 Counting parameters and FLOPs

Question: From this table, can we calculate the LLaMA 3-70B parameter count? Let's apply the content of Section 4 and see if we can get 70B!

param	formula	count
FFW params	d_model * d_ff * 3 (for gelu + out-projection) * n_layers	8,192 * 8,192 * 3.5 * 3 * 80 = **56.3e9**
Vocab params	2 (input and output embeddings) * n_embeddings * d_model	2 * 128,256 * 8,192 = **2.1e9**
Attention params	n_layers * [2 (for q embedding and concatenated output projection) * d_model * n_heads * d_qkv + 2 (for k and v) * d_model * n_kv_heads * d_qkv]	80 * (2 * 8,192 * 64 * 128 + 2 * 8,192 * 8 * 128) = **12e9**
		56.3e9 + 2.1e9 + 12e9 = **70.4e9**

That's great! We get the number we expect. You'll notice as expected that the FFW parameters totally dominate the overall parameter count, although attention is non-trivial.

Takeaway: The 3 big weight matrices in the MLP block are so much larger than all the other arrays in the Transformer that we can typically almost ignore all other parameters when reasoning about model memory or FLOPs. For LLaMA 3-70B, they represent 56B of 70B parameters.

Let's dig into some basic questions for LLAMA 3-70B. We'll ask some random but useful questions and try to answer them.

Let's look at FLOPs now! *Remember the general rules for training from Section 4*

Question: How many FLOPs does LLaMA-3 perform per token per training step? *This helps us determine how expensive the whole training process will be.*

Answer: As shown in Section 4, we do roughly $6 \cdot$param count FLOPs per token, so here that's roughly `6 * 70e9 = 4.2e11` FLOPs / token. That's about half a TFLOP per token per step. Assuming we're compute-bound, this should take roughly `4.2e11 / 4.59E+14 = 1ms` on a single TPU v5p chip, assuming perfect FLOPs utilization.

Question: LLAMA 3 was trained for about 15 trillion tokens. How many FLOPs is that total?

Answer: That's easy, it's just `4.2e11 * 15e12 = 6.3e24 FLOPs` total. 6.3 yottaFLOPs. That's a lot! On a single TPU this would take `6.3e24 / 4.59E+14 = 435 years`. That's also a lot!

Question: Let's say we wanted to train on a full TPU v5p pod with 16x20x28 = 8960 chips. How long would this take to train at 40% MFU in bfloat16, assuming we are compute-bound?

Answer: We know that each TPU v5p can perform `4.59e14` FLOPs / second. At 40% MFU, this will take about `T = 6.3e24 / (8960 * 4.59e14 * 0.4) = 3.8e6 seconds`. **This is about 44 days!** That's fairly reasonable, assuming we can actually achieve 40% MFU.

Question: LLaMA 3-70B was pretrained with a batch size of about 4M tokens. How many TPUs do we need at minimum to train with this batch size? *You can assume bfloat16 parameters and float32 optimizer state, and that you checkpoint gradients 4 times per layer.*

While this isn't that relevant of a question, it gives us a ballpark for the minimum compute resources to train a model like this yourself.

Answer: This question is primarily asking about memory usage, since that's the only strict constraint on available compute. During training, we have three primary uses of HBM: model parameters, optimizer state, and gradient checkpoints. If we assume bfloat16 weights, float32 optimizer state, and a *very* conservative gradient checkpointing scheme (4 times per layer), we have:

Params	2 * 70GB	140GB
Optimizer State	8 * 70GB	560GB
Gradient Checkpoints	2 * 8192 * 4e6 * 4 * 80	20.9TB
Total	-	21.6TB

The total here is about 21.6TB. You notice that gradient checkpointing strongly dominates the memory picture, even with a very conservative checkpointing scheme. We could technically go to 1 checkpoint per layer, or do microbatching, but this is a reasonable picture. With these assumptions, since each TPU v5p has 96GB of HBM, we need `21.6e12 / 96e9 = 225` TPUs. That's not very much actually!

Why wouldn't we do this? Well, because it would take us `44 days * 8960 / 225 = 1752 days` to train. That's nearly four years. **That's a lot.** Still, this makes it clear that we're using these large clusters not because we're bound by memory but rather because we need the extra FLOPs.

Question: Under the same assumptions as the question above, if we use 8960 TPU v5p chips, how much memory will we use per-chip?

Answer: Our total memory is still about 21.6TB, so per-chip we'll be using about 2.4GB per chip, which is basically nothing. If we did much more aggressive checkpointing, e.g. 12 checkpoints per layer, we'd still only be at 8GB per chip. We're nowhere near being memory bound during training at these scales.

Takeaway: It is technically possible to train even very large models on very small topologies, with the caveat that they will likely take a long time. Being able to calculate the total FLOPs of a training run allows us to ballpark its training time by assuming a modest MFU and a known topology.

6.3 How to shard LLAMA 3-70B for training

Let's stick to our setting from above and say we want to train LLaMA 3-70B with 4M token batch size (1024 sequences of length 4096 per batch) on a TPU v5p pod of 8960 chips. Let's discuss what the best sharding

strategy is for this model.

Question: Under the assumptions above, can we train our model with FSDP alone? To start, let's say we can't do any sequence/context parallelism. *This should be the first idea you have, since it's simple and will introduce no extra communication if it works.*

Answer: This answer will be a little pedantic. As noted above, LLaMA 3-70B is initially trained with sequences of length 4K, so the batch size of 4M tokens gives us a *sequence batch size* of 1024. That means we can only really do pure data parallelism/FSDP up to 1024 chips *because that's how many sequences we have to do data parallelism over*. So the answer in the simple sense of "fully data parallelism with no extra communication" is no. The next question will answer a slightly less pedantic version of this.

Question: Let's relax the requirement of not doing any sequence sharding. If we allow ourselves to do FSDP over both the batch *and* sequence axes, can we train LLaMA 3-70B with only FSDP on 8960 chips?

Answer: Now that we're allowing ourselves to do sequence/context parallelism as well, we can scale up way more. First let's calculate our per-device batch size. If we do 8960-way FSDP, we end with a per-TPU batch size of `4 * 1024 * 1024 / 8960 = 468 tokens`. We know from the previous section that we become ICI-bound by FSDP when per device batch size $< 2550/n_{\text{axes}}$. Since we can dedicate 3 axes here with a full 3D pod, this would give us a lower bound of 850, which we're well below. **So the answer is no, even with 3 axes. We would be solidly communication-bound.**

Question: Now let's look at mixed tensor parallelism and FSDP. Does there exist some combination that lets us remain compute-bound? What amount of FSDP and tensor parallelism should we do if so?

Answer: First let's check the discriminant to see if this will even fit. We know that we'll be comms-bound if our per-chip batch size is less than $2 \cdot 2550^2/F = 453$. As we saw above, we're slightly above this. So that's great! The approximate equation we cooked up in the previous section is

$$X \geqslant \frac{B}{2550} + \frac{\sqrt{B^2 \cdot F^2 - 2 \cdot 2550^2 \cdot F \cdot B \cdot N}}{2550 \cdot F}$$

for mixed FSDP + tensor parallelism. Plugging in our numbers above, this is

$$X \geqslant \frac{4.19e6}{2550} + \frac{\sqrt{4.19e6^2 \cdot 28672^2 - 2 \cdot 2550^2 \cdot 28672 \cdot 4.19e6 \cdot 8960}}{2550 \cdot 28672}$$

which is roughly

$$X \geqslant 1643 + 284 = 1927$$

Rounding to a reasonable multiple of 2, that gives us roughly 2048-way FSDP and 4-way model parallelism. That should work well!

Takeaway: We can train LLaMA-3 with a 4M token batch size on a full TPU v5p pod with a mixture of data parallelism (1024-way), sequence parallelism (2-way), and tensor parallelism (4-way) without being communication-bound. We will be comms-bound if we try to do pure FSDP or FSDP + sequence parallelism. The equations we've cooked up in the previous section are very practical.

6.4 Worked Problems

Exercise 6.1 *[Scaling LLAMA 70B to more chips]*

Say we want to train LLAMA 3-70B on 4 pods with the same batch size. What parallelism scheme would we use? Would we be compute or communication bound? Roughly how long would it take to train? *Make sure to use the correct roofline bound.*

Exercise 6.2 *[LLAMA 405B]*

1. Using the LLAMA 3-405B config[2], write a table with all the key hyperparameters as above. How many total parameters does this model have? How many FLOPs per training step? How many FLOPs do we perform if we train for 15T tokens?
2. Assume we want to train on 8 TPU v5p pods. What parallelism scheme would we use? How long would training take? Would be compute or comms bound?

[2] https://huggingface.co/meta-llama/Llama-3.1-405B/blob/main/config.json

7 All About Transformer Inference

Performing inference on a Transformer can be very different from training. Partly this is because inference adds a new factor to consider: latency. In this section, we will go all the way from sampling a single new token from a model to efficiently scaling a large Transformer across many slices of accelerators as part of an inference engine.

7.1 The Basics of Transformer Inference

So you've trained a Transformer, and you want to use it to generate some new sequences. *At the end of the day, benchmark scores going up and loss curves going down are only proxies for whether something interesting is going to happen once the rubber hits the road!*[1]

Sampling is conceptually simple. We put a sequence in and our favorite Transformer will spit out $\log p($ next token$_i$|previous tokens), i.e. log-probabilities for all possible next tokens. We can sample from this distribution and obtain a new token. Append this token and repeat this process and we obtain a sequence of tokens which is a continuation of the prompt.

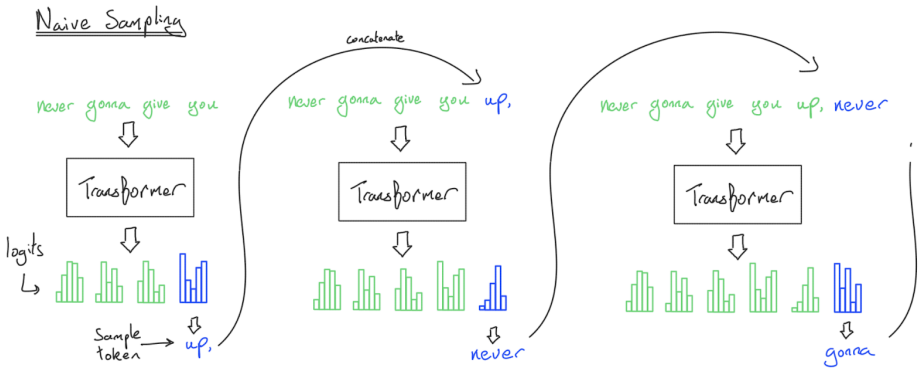


Figure 18: naive sampling from a Transformer. The blue logits give us a distribution over the next token that we can sample from. Note that each step re-processes the entire prefix, leading to an $O(N^2)$ runtime for the algorithm.

We have just described the naive implementation of Transformer sampling, and while it works, **we never do it in practice** because we are re-processing the entire sequence every time we generate a token. This algorithm is $O(n^2$ on the FFW and $O(n^3)$ on the attention mechanism to generate n tokens!

How do we avoid this? Instead of doing the full forward pass every time, it turns out we can save some intermediate activations from each forward pass that let us avoid re-processing previous tokens. Specifically, since a given token only attends to previous tokens during dot-product attention, we can simply write each token's key and value projections into a new data structure called a **KV cache**. Once we've saved these key/value projections for past tokens, future tokens can simply compute their $q_i \cdot k_j$ products without performing any new FLOPs on the earlier tokens. Amazing!

[1]Historically, you can do a surprising amount of research on Transformers without ever touching inference — LLM loss, multiple choice benchmarks can be run efficiently without a proper KV cache or generation loop implementation. This meant, especially in research codebases, there's often a lot of low hanging fruits in the inference codepath.

With this in mind, inference has two key parts:

1. Prefill: Given a long prompt, we process all the tokens in the prompt at the same time and save the resulting activations (specifically, the key-value projections) in a **"KV cache"**. We also save the logits for the last token.
2. Generation: Given a KV cache and the previous logits, we incrementally sample one token from the logits, feed that token back into the Transformer, and produce a new set of logits for the next step. We also append the KV activations for that new token to the KV cache. We repeat this until we hit a special `<EOS>` token or reach some maximum length limit.

Here's a diagram of sampling with a KV cache:

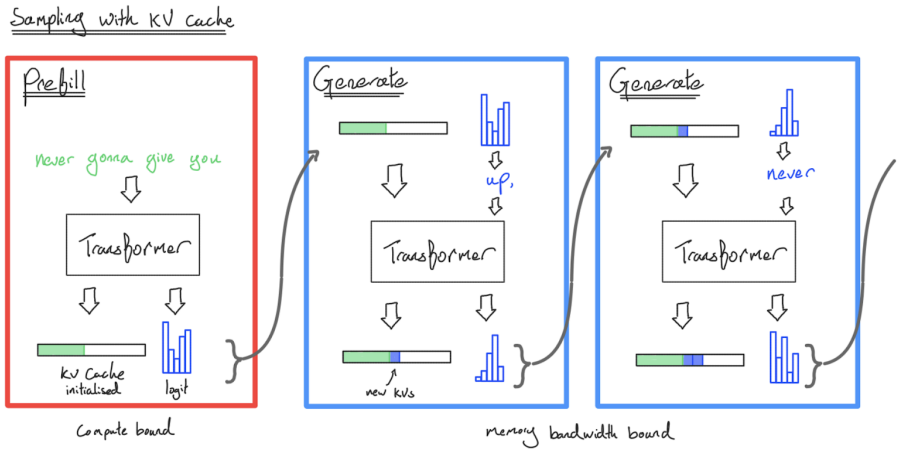


Figure 19: diagram of efficient Transformer sampling with a KV cache. Prefill processes our prompt and saves all the per-token key-value activations in a cache. Generation takes this cache (and the last-token logits), samples a new token, and passes that new token through the model, attending to the KV cache and saving the new key-value projections back to the cache. This is an $O(N)$ algorithm in the MLP block.

By sampling with a KV cache, we've reduced our time complexity to generate n tokens to (n) on the FFW and $O(n^2)$ on the attention, since we never reprocess a previous token. However, many forward passes are still needed to generate a sequence — that's what's happening when you query Gemini or ChatGPT and the result streams back to you. Every token is (usually) a separate (but partially cached) Transformer call to a massive model.

We will soon see that prefill and generation are very different beasts —– Transformer inference is two tasks in disguise! Compared to training, the KV cache is also a novel and significant source of complexity.

7.1.1 What do we actually want to optimize?

Before we proceed further, it's worth highlighting one aspect of inference that's totally new: latency. While during training we only care about throughput (total tokens processed per second), during inference we have to worry about how fast we're producing tokens (both the **Time To First Token (TTFT)** and the **per-token latency**). For example:

- **Offline batch inference** for evals and data generation only cares about bulk cost of inference and is blind to the latency of individual samples.

- **Chat interfaces/streaming tasks** need to run cheaply at scale while having low TTFT and generating tokens fast enough to exceed human reading speed.
- **Edge inference** (e.g. llama.cpp on your laptop) only needs to service one user at a time at the lowest possible latency, potentially with heavy hardware constraints.

Maximizing hardware utilization is still critical and helps with cost and TTFT, but unlike training, it does not *necessarily* translate to better experience for individual users in all contexts. Many optimizations at the accelerator, systems and model architectural level make tradeoffs between latency, throughput, context length and even model quality.

7.1.2 A more granular view of the Transformer

So far we've mostly treated a Transformer as a stack of feedforward blocks. While this is often reasonable from a FLOPs and memory standpoint, it's not sufficient to properly model inference.[2] As we saw in Part 4, the major components of a Transformer forward pass are:

1. **A bunch of linear operations**, including the MLP (W_{in}, W_{out}) and the attention QKV projections and output projections (W_Q, W_K, W_V, and W_O). These all involve reading parameters and a batch of activations from HBM, doing some FLOPs, and writing the result back to HBM.
2. **Dot-product attention**. We need to read a batch of key-value projections and a batch of query activations from HBM, do a couple inner products and some softmax ops, and write the attention result back to HBM.
3. **Everything else**, including applying layer norms, activation functions, tokens sampling, updating KV caches, and positional embeddings. These do take some FLOPs, but are dominated by, or fused into, the above.

For the next couple sections, we're going to look at each of these in the context of prefill and generation and ask what is likely to bottleneck our performance. Are we compute-bound or memory-bound? We want to emphasize how different the answers will be for prefill versus generation.

7.1.3 Linear operations: what bottlenecks us?

All our linear operations are conceptually the same, whether they live in the MLP block or attention. Their arithmetic intensity depends on the batch size. We did this math in Section 1 but it's worth repeating. Let's look at a single matrix multiply of a bf16[B, D] batch by a bf16[D, F] matrix. This could be the big MLP block or one of the smaller attention projections (W_Q, W_K, W_V, W_O). To do this matrix multiplication, we need to load both of these arrays from HBM into the MXU, do the multiplicaton, then write the result back to HBM. As before, we have:

$$T_{\text{math}} = \frac{\text{total FLOPs}}{\text{TPU FLOPs/s}} = \frac{2BDF}{\text{TPU FLOPs/s}}$$

$$T_{\text{comms}} = \frac{\text{total bytes}}{\text{TPU HBM memory bandwidth}} = \frac{2BD + 2FD + 2BF}{\text{HBM bandwidth}}$$

[2] One thing you'll notice throughout this section is that inference is much less forgiving than training. We typically have far fewer FLOPs, less opportunity for batching, and a much greater sensitivity to latency. KV caches dramatically complicate inference as well.

The TPU can overlap these by loading as it does the compute, so to be compute-bound, we need $T_{\text{math}} \geq T_{\text{comms}}$, or:

$$\frac{2BDF}{2BD + 2DF + 2BF} \geq \frac{\text{TPU FLOPs/s}}{\text{HBM Bandwidth}} = \frac{1.97E + 14}{8.20E + 11} = 240$$

where the RHS is the arithmetic intensity of our hardware. Now let's assume D and F are very large compared to B (usually our batches are at most 500 and D and $F > 10k$), we can simplify the denominator by using the fact that $2BD + 2DF + 2BF \approxeq 2DF$ which gives us

$$\frac{2BDF}{2BD + 2DF + BF} \approxeq \frac{2BDF}{2DF} \geq \frac{\text{TPU FLOPs/s}}{\text{HBM Bandwidth}}$$
$$= \frac{1.97E + 14}{8.20E + 11} \implies B \geq 240 = B_{\text{crit}}$$

Takeaway: To be compute-bound on any matrix multiplication, our total token batch size must be greater than B_{crit}, which depends on the hardware and quantization. For bfloat16 activations on TPU v5e, this is 240 tokens. This applies to any simple matmul in our Transformer (e.g. the MLP block or the attention projections).

During training, we'll have a high intensity during all our matrix multiplications because we reuse the same weights over a very large batch. **That high arithmetic intensity carries over to prefill, since user prompts are typically hundreds if not thousands of tokens long.** As we saw before, the hardware arithmetic intensity of a TPUv5e is 240, so if a sequence longer than 240 tokens is fed into a dense model running on this hardware at bf16, we would expect to be compute-bound and all is well. Prompts shorter than this can technically be batched together to achieve higher utilization, but this is typically not necessary.

Takeaway: during prefill, all matrix multiplications are basically always compute-bound. Therefore, simply maximizing hardware utilization or MFU (Model FLOPs Utilization) is enough to maximize throughput-per-chip (cost) and latency (in the form of TTFT). Unless prompts are extremely short, batching at a per-prompt level only adds latency for a small improvements in prefill throughput.

However, during generation, for each request, we can only do our forward passes one token at a time since there's a sequential dependency between steps! Thus we can only (easily) achieve good utilization by batching multiple requests together, parallelizing over the batch dimension. We'll talk about this more later, but actually batching many concurrent requests together without affecting latency is hard. For that reason, **it is much harder to saturate the hardware FLOPs with generation.**

Takeaway: Our total token batch size must be greater than B_{crit} for generation to be compute-bound on the linear/feed-forward operations (240 for bf16 params on TPU v5e). Because generation happens serially, token-by-token, this requires us to batch multiple requests together, which is hard!

It's worth noting just how large this is! Generate batch size of 240 means 240 concurrent requests generating at once, and 240 separate KV caches for dense models. That means this is difficult to achieve in practice, except in some bulk inference settings. In contrast, pushing more than 240 tokens through during a prefill is pretty routine, though some care is necessary as sparsity increases.

Note that this exact number will differ on the kind of quantization and hardware. Accelerators often can supply more arithmetic in lower precision. For example, if we have int8 parameters but do our computation in bfloat16, the critical batch size drops to 120. With int8 activations and int8 params, it jumps back up to 240 since the TPUv5e can supply 400 TOPs/s of int8 x int8.

7.1.4 What about attention?

Things get more complicated when we look at the dot-product attention operation, especially since we have to account for KV caches. Let's look at just one attention head with pure multi-headed attention. In a single Flash Attention fusion, we[3]:

1. Read the Q activations of shape bf16[B, T, D] from HBM.
2. Read the KV cache, which is a pair of bf16[B, S, D] tensors from HBM.
3. Perform $2BSTD$ FLOPs in the QK matmul. With Flash Attention, we don't need to write the bf16[B, S, T] attention matrix back into HBM.
4. Perform $2BSTD$ in the attention AV matmul.
5. Write the resulting bf16[B, T, D] tensor back into HBM.

Putting it all together, we get:

$$\text{Multiheaded Attention Arithmetic Intensity} = \frac{4BSTD}{4BSD + 4BTD} = \frac{ST}{S+T}$$

For prefill, $S = T$ since we're doing self-attention, so this simplifies to $T^2/2T = T/2$. This is great because it means **the arithmetic intensity of attention during prefill is** $O(T)$. That means it's quite easy to be compute-bound for attention. As long as our sequence length is fairly large, we'll be fine!

But since generation has a trivial sequence dim, and the B and D dims cancel, we can make the approximation:

$$S >> T = 1 \implies \frac{ST}{S+T} \approx 1$$

This is bad, since it means we cannot do anything to improve the arithmetic intensity of attention during generation. We're doing a tiny amount of FLOPs while loading a massive KV cache. **So we're basically always memory bandwidth-bound during attention!**

> **Takeaway:** during prefill, attention is usually compute bound for any reasonable sequence length (roughly > 480 tokens) while during generation our arithmetic intensity is low and constant, so we are always memory bandwidth-bound.

Why is this, conceptually? Mainly, we're compute-bound in linear portions of the model because the parameters (the memory bandwidth-heavy components) are reused for many batch items. However, every batch item has its own KV cache, so a bigger batch size means more KV caches. We will almost *always* be memory

[3]We're simplifying a fair bit here by ignoring the non-matmul FLOPs in applying the softmax, masks etc. They should be overlapped with computation or HBM reads, but it can be non-trivial to do on certain TPU generations. Whese details don't change the main message, which is that KV caches are usually memory bound.

bound here unless the architecture is adjusted aggressively.

This also means you will get diminishing returns on throughput from increasing batch size once params memory becomes comparable to KV cache memory. The degree to which the diminishing returns hurt you depends on the ratio of parameter to KV cache bytes for a single sequence, i.e. roughly the ratio $2DF/SHK$. Since $HK \approx D$, this roughly depends on the ratio of F to S, the sequence length. This also depends on architectural modifications that make the KV cache smaller (we'll say more in a moment).

7.1.5 Theoretical estimates for LLM latency and throughput

From this math, we can get pretty good bounds on the step time we should aim for when optimizing. **(Note: if there is one thing we want to the reader to take away from this entire chapter, it's the following).** For small batch sizes during generation (which is common), we can lower-bound our per-step latency by assuming we're memory bandwidth bound in both the attention and MLP blocks:

$$\text{Theoretical Min Step Time} = \frac{\text{Batch Size} \times \text{KV Cache Size} + \text{Parameter Size}}{\text{Total Memory Bandwidth}}$$

Similarly, for throughput:

$$\text{Theoretical Max Tokens/s} = \frac{\text{Batch Size} \times \text{Total Memory Bandwidth}}{\text{Batch Size} \times \text{KV Cache Size} + \text{Parameter Size}}$$

Eventually, as our batch size grows, FLOPs begin to dominate parameter loading, so in practice we have the more general equation:

$$\text{Theoretical Step Time (General)} = \underbrace{\frac{\text{Batch Size} \times \text{KV Cache Size}}{\text{Total Memory Bandwidth}}}_{\text{Attention (always bandwidth-bound)}} + \underbrace{\max\left(\frac{2 \times \text{Batch Size} \times \text{Parameter Count}}{\text{Total FLOPs/s}}, \frac{\text{Parameter Size}}{\text{Total Memory Bandwidth}}\right)}_{\text{MLP (can be compute-bound)}} \quad (4)$$

where the attention component (left) is never compute-bound, and thus doesn't need a FLOPs roofline. These are fairly useful for back-of-the-envelope calculations, e.g.

Pop Quiz: Assume we want to take a generate step with a batch size of 4 tokens from a 30B parameter dense model on TPU v5e 4x4 slice in int8 with bf16 FLOPs, 8192 context and 100 kB / token KV caches. What is a reasonable lower bound on the latency of this operation? What if we wanted to sample a batch of 256 tokens?

Answer: in int8, our parameters will use 30e9 bytes and with the given specs our KV caches will use `100e3 * 8192 = 819MB` each. We have 16 chips, each with `8.1e11` bytes/s of bandwidth and `1.97e14` bf16 FLOPs/s. From the above equations, since we have a small batch size, we expect our step time to be at least `(4 * 819e6 + 30e9) / (16 * 8.1e11) = 2.5 ms`. At 256 tokens, we'll be well into the compute-bound regime for our MLP blocks, so we have a step time of roughly `(256 * 819e6) / (16 * 8.1e11) + (2 * 256 * 30e9) / (16 * 1.97e14) = 21ms`.

As you can see, there's a clear tradeoff between throughput and latency here. Small batches are fast but don't utilize the hardware well. Big batches are slow but efficient. Here's the latency-throughput Pareto frontier calculated for some older PaLM models (from the ESTI paper[4]):

[4] https://arxiv.org/pdf/2211.05102

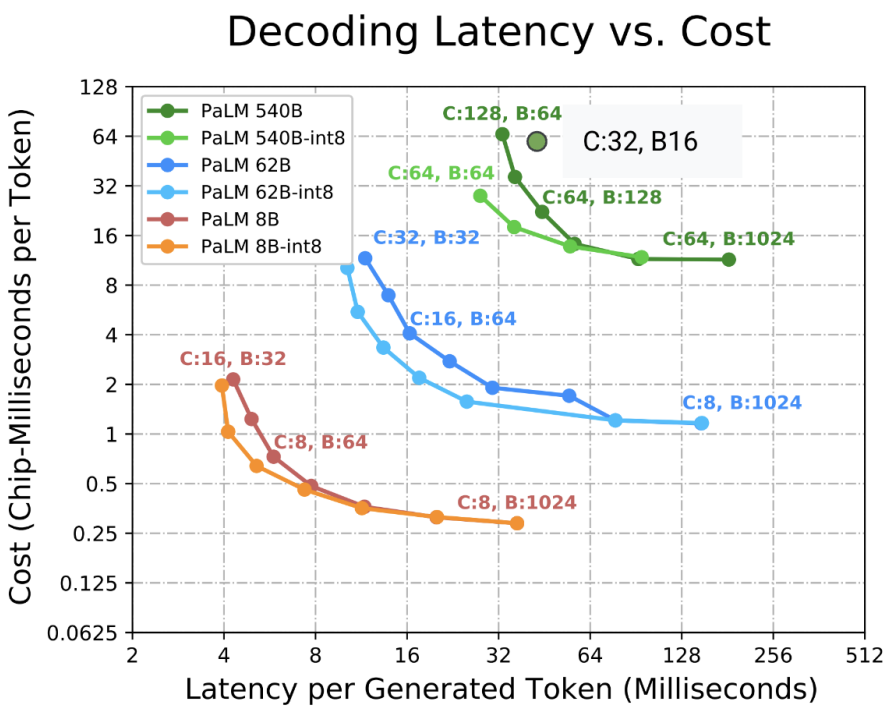


Figure 20: Pareto frontier of cost (read: throughput) versus latency for several PaLM models. Note how chip count (C) and batch size (B) moves you along the Pareto frontier, with the exception of the green dot (C:32 B:16 for PaLM 540B) where the available memory prevented the setup from supporting a good batch size and caused throughput to suffer. Note how throughput generally tends to flatten around after the batch size 240. int8 weights offers a better latency-throughput pareto optimal, but not a better max throughput.

Not only do we trade off latency and throughput with batch size as knob, we may also prefer a larger topology to a smaller one so we can fit larger batches if we find ourselves limited by HBM. The next section explores this in more detail.

Takeaway: If you care about generate throughput, use the largest per-chip batch size possible. Any per-chip batch size about the TPU arithmetic intensity (B_{crit}, usually 120 or 240) will maximize throughput. You may need to increase your topology to achieve this. Smaller batch sizes will allow you to improve latency at the cost of throughput.

This is all quite theoretical. In practice we often don't quite see a sharp roofline for a few reasons:

- Our assumption that HBM reads will be perfectly overlapped with FLOPs is not realistic, since our compiler (XLA) is fallible.
- For sharded models, XLA also often fails to efficiently overlap the ICI communication of our model-sharded matrix multiples with the FLOPs themselves, so we often start taking a latency hit on linears over $BS = 32$.
- Batch sizes larger than the theoretical roofline will still see some improvement in throughput because of imperfect overlapping, but this is a good heuristic.

7.1.6 What about memory?

We've spent some time looking at bandwidth and FLOPs, but not at memory. The memory picture looks a lot different at inference time, thanks to our new data structure, the KV cache. For this section, let's pick a real model (LLaMA 2-13B) to demonstrate how different things look:

hyperparam	value
n_{layers} (L)	40
d_{model} (D)	5,120
$\text{ffw}_{\text{multiplier}}$ (F // D)	2.7
n_{heads} (N)	40
$n_{\text{kv\_heads}}$ (K)	40
d_{qkv} (H)	128
$n_{\text{embeddings}}$ (V)	32,000

What's using memory during inference? Well, obviously, our parameters. Counting those, we have:

param	formula	count
FFW params	d_model * d_model * ffw_multiplier * 3 (for gelu + out-projection) * n_layers	5,120 * 5,120 * 2.7 * 3 * 40 = **8.5e9**
Vocab params	2 (input and output embeddings) * n_embeddings * d_model	2 * 32,000 * 5,120 = **0.3e9**
Attention params	n_layers * [2 (for q embedding and concatenated output projection) * d_model * n_heads * d_qkv + 2 (for k and v) * d_model * n_kv_heads * d_qkv]	80 * (2 * 5,120 * 40 * 128 + 2 * 5,1220 * 40 * 128) = **4.2e9**

Adding these parameters up, we get 8.5e9 + 4.2e9 + 0.3e9 = **13e9 total parameters**, just as expected. As we saw in the previous sections, during training we might store our parameters in bfloat16 with an optimizer state in float32. That may use around 100GB of memory. That pales in comparison to our gradient checkpoints, which can use several TBs.

How is inference different? During inference, we store one copy of our parameters, let's say in bfloat16. That uses 26GB — and in practice we can often do much better than this with quantization. There's no optimizer state or gradients to keep track of. Because we don't checkpoint (keep activations around for the backwards pass), our activation footprint is negligible for both prefill[5] and generate. If we prefill 8k tokens, a single activation only uses around [6] of memory. Longer prefills can be broken down into many smaller forward passes, so it's not a problem for longer contexts either. Generation use even fewer tokens than that, so activations are negligible.

The main difference is the KV cache. These are the keys and value projections for all past tokens, bounded in size only by the maximum allowed sequence length. The total size for T tokens is

$$\text{KV cache size} = 2 \cdot \text{bytes per float} \cdot H \cdot K \cdot L \cdot T$$

[5] Particularly thanks to Flash Attention, which avoids materializing our attention matrix

[6] 8,192 x 5,120 x 2 bytes = 80MB

where H is the dimension of each head, K is the number of KV heads, L is the number of layers, and the 2 comes from storing both the keys and values.

This can get big very quickly, even with modest batch size and context lengths. For LLaMA-13B, a KV cache for a single 8192 sequence at bf16 is

$$8192\ (T) \times 40\ (K) \times 128\ (H) \times 40\ (L) \times 2\ \text{(bytes)} \times 2 = 6.7\text{GB}$$

Just 4 of these exceed the memory usage of our parameters! To be clear, LLaMA 2 was not optimized for KV cache size at longer contexts (it isn't always this bad, since usually K is much smaller, as in LLaMA-3), but this is still illustrative. We cannot neglect these in memory or latency estimates.

7.1.7 Modeling throughput and latency for LLaMA 2-13B

Let's see what happens if we try to perform generation perfectly efficiently at different batch sizes on 8xTPU v5es, up to the critical batch size (240) derived earlier for maximum theoretical throughput.

Batch Size	1	8	16	32	64	240
KV cache Memory (GiB)	6.7	53.6	107.2	214.4	428.8	1608
Total Memory (GiB)	32.7	79.6	133.2	240.4	454.8	1634
Theoretical Step Time (ms)	4.98	12.13	20.30	36.65	69.33	249.09
Theoretical Throughput (tokens/s)	200.61	659.30	787.99	873.21	923.13	963.53

8x TPU v5es gives us 128GiB of HBM, 6.5TiB/s of HBM bandwidth (0.82TiB/s each) and 1600TF/s of compute.

For this model, increasing the batch size does give us better throughput, but we suffer rapidly diminishing returns. We OOM beyond batch size 16, and need an order of magnitude more memory to go near 240. A bigger topology can improve the latency, but we've hit a wall on the per chip throughput.

Let's say we keep the total number of params the same, but magically make the KV cache 5x smaller (say, with 1:5 GMQA, which means we have 8 KV heads shared over the 40 Q heads — see next section for more details).

Batch Size	1	8	16	32	64	240
KV cache Memory (GiB)	1.34	10.72	21.44	42.88	85.76	321.6
Total Memory (GiB)	27.34	36.72	47.44	68.88	111.76	347.6
Theoretical Step Time (ms)	4.17	5.60	7.23	10.50	17.04	52.99
Theoretical Throughput (tokens/s)	239.94	1,429.19	2,212.48	3,047.62	3,756.62	4,529.34

With a smaller KV cache, we still have diminishing returns, but the theoretical throughput per chip continues to scale up to batch size 240. We can fit a much bigger batch of 64, and latency is also consistently better at all batch sizes. The latency, maximum throughput, and maximum batch size all improve dramatically! In fact, later LLAMA generations used this exact optimization – LLAMA-3 8B has 32 query heads and 8 KV

heads.

> **Takeaway:** In addition to params, the size of KV cache has a lot of bearing over the ultimate inference performance of the model. We want to keep it under control with a combination of architectural decisions and runtime optimizations.

7.2 Tricks for Improving Generation Throughput and Latency

Since the original Attention is All You Need paper[7], many techniques have been developed to make the model more efficient, often targeting the KV cache specifically. Generally speaking, a smaller KV cache makes it easier to increase batch size and context length of the generation step without hurting latency, and makes life easier for the systems surrounding the Transformer (like request caching). Ignoring effects on quality, we may see:

Grouped multi-query attention (aka GMQA, GQA): We can reduce the number of KV heads, and share them with many Q heads in the attention mechanism. In the extreme case, it is possible to share a single KV head across all Q heads. This reduces the KV cache by a factor of the Q:KV ratio over pure MHA, and it has been observed that the performance of models is relatively insensitive to this change.

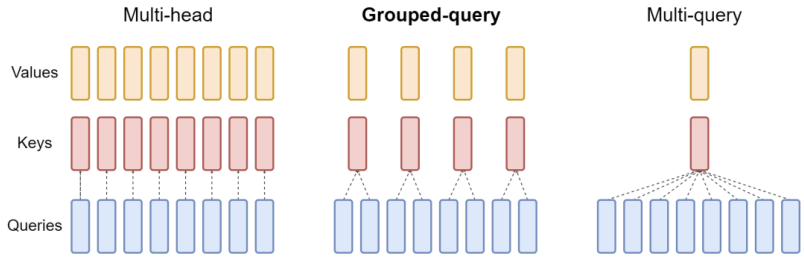


Figure 2: Overview of grouped-query method. Multi-head attention has H query, key, and value heads. Multi-query attention shares single key and value heads across all query heads. Grouped-query attention instead shares single key and value heads for each *group* of query heads, interpolating between multi-head and multi-query attention.

This also effectively increases the arithmetic intensity of the attention computation (see Question 4 in Section 4).

Mixing in some local attention layers: Local attention caps the context to a small to moderately sized max length. At training time and prefill time, this involves masking the attention matrix to a diagonal strip instead of a triangle. This effectively caps the size of the max length of the KV cache for the local layers. By mixing in some local layers into the model with some global layers, the KV cache is greatly reduced in size at contexts longer than the local window.

Sharing KVs across layers: The model can learn to share the same KV caches across layers in some pattern. Whilst this does reduce the KV cache size, and provide benefits in increasing batch size, caching, offline storage etc. shared KV caches may need to be read from HBM multiple times, *so it does not necessarily improve the step time*.

[7] https://arxiv.org/abs/1706.03762

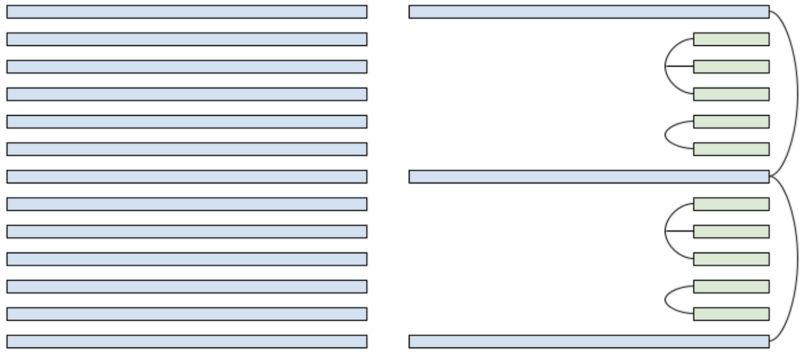


Figure 1. Left: Standard transformer design where every attention is global attention. Right: The attention design in our production model. Blue boxes indicate global attention, green boxes indicate local attention, and curves indicate KV-sharing. For global attention layers, we share KV across multiple non-adjacent layers. This illustration depicts only a subset of the layers in the full model.

Figure 21: **Left:** Multiple layers of pure global attention. **Right:** An example of some global/local interleaving pattern with sharing with adjacent layers. Source: Character.ai blog.

Quantization: Inference is usually less sensitive to the precision of parameters and KVs. By quantizing the parameters and KV cache (e.g. to int8, int4, fp8 etc.), we can save on memory bandwidth on both, decrease the batch size required to reach the compute roofline and save memory to run at bigger batch sizes. Quantization has the added advantage that even if the model was not trained with quantization it can often be applied post training.

Using ragged HBM reads and Paged Attention: We allocated 8k of context for each KV cache in the calculations above but it is often not necessary to read the entire KV cache from memory — requests have a wide range of length distributions and don't use the max context of the model, so we can often implement kernels (e.g. Flash Attention variants) that only read the non-padding part of the KV cache.

Paged Attention is a refinement upon this that stores KV caches in OS-style page tables and mostly avoids padding the KV caches altogether. This adds a lot of complexity but means every batch only uses as much memory as it needs. This is a runtime optimization, so again it is indifferent to architecture.

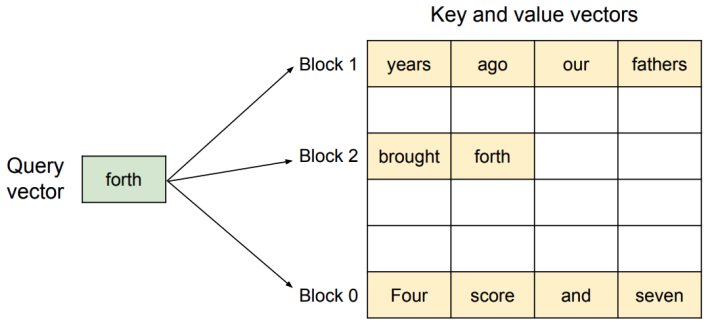


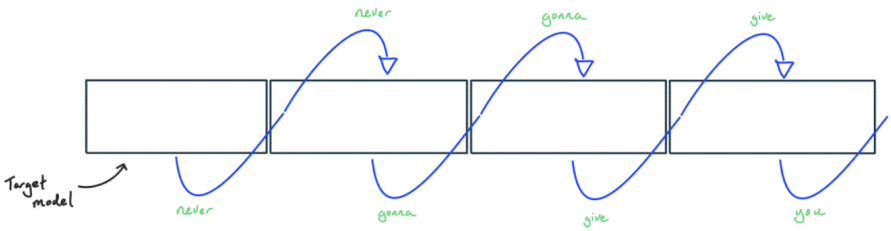
Figure 5. Illustration of the PagedAttention algorithm, where the attention key and values vectors are stored as non-contiguous blocks in the memory.

Figure 22: During generation, a single token (forth) attends to multiple KV cache blocks/pages. By paging the KV cache, we avoid loading or storing more memory than we need to. Taken from the PagedAttention paper.

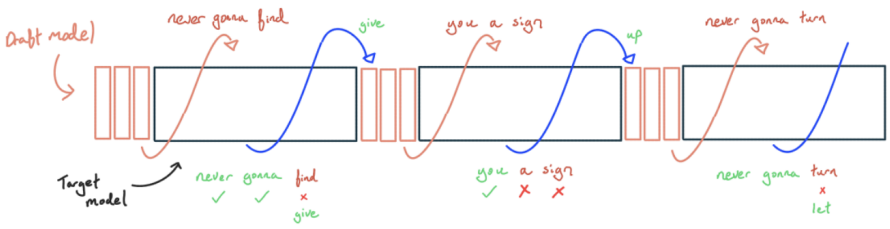
Big Picture: All told, these KV cache optimizations can reduce KV cache sizes by over an order of magnitude compared to a standard MHA Transformer. This can lead to an order-of-magnitude improvement in the overall cost of the Transformer.

7.2.1 Speculative Sampling

When we *really* care about end to end latency, there is one extra trick we can employ called speculative sampling. As a recap, we usually generate tokens from a large Transformer one by one:



With speculative sampling, we use a smaller, cheaper model to generate tokens and then check the result with the big model. This is easiest to understand with *greedy decoding*:



1. We sample greedily from some smaller, cheaper model. Ideally we use a model trained to match the larger model, e.g. by distillation, but it could be as simple as simply using n-grams or token matching a small corpus of text.
2. After we've generated K tokens, we use the big model to compute the next-token logits for all the tokens we've generated so far.
3. Since we're decoding greedily, we can just check if the token generated by the smaller model has the highest probability of all possible tokens. If one of the tokens is wrong, we take the longest correct prefix and replace the first wrong token with the correct token, then go back to (1). If all the tokens are correct, we can use the last correct logit to sample an extra token before going back to (1).

Why is this a latency win? This scheme still requires us to do the FLOPs-equivalent of one forward pass through the big model for every token, but because we can batch a bunch of tokens together, we can do all these FLOPs in one forward pass and take advantage of the fact that we're *not compute-bound* to score more tokens for free.

Every accepted token becomes more expensive in terms of FLOPs on average (since some will be rejected, and we have to call a draft model), but we wring more FLOPs out of the hardware, and the small model is cheap, so we win overall. Since everything has been checked by the big model, we don't change the sampling distribution at all (though the exact trajectory will differ for non-greedy).

For normal autoregressive sampling the token/s is the same as the step time. We are still beholden to the theoretical minimum step time according to the Arithmetic Intensity section here (in fact, Speculative Sampling step times are usually quite a bit slower than normal autoregressive sampling, but because we get more than 1 token out per step on average we can get much better tokens/s).

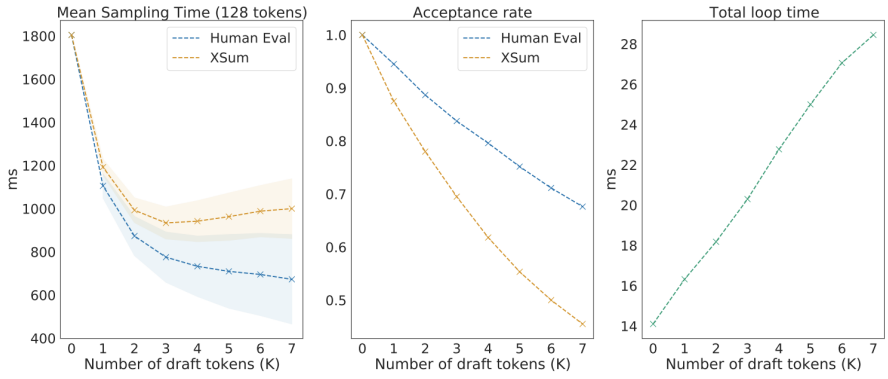


Figure 1 | **Left:** The average time to generate 128 tokens, with standard deviation. Note that as K increases, the overall speedup plateaus or even regresses, with XSum being optimal at $K = 3$. The variance consistently increases with K. **Middle:** The average number of tokens accepted divided by $K + 1$ – this serves as a measure of the overall efficiency of the modified rejection scheme, which decreases with the lookahead. **Right:** Average time per loop increases approximately linearly with K due to the increased number of model calls. Note that the gradient is slightly higher than the sampling speed of the draft model, due to additional overheads in nucleus decoding.

Figure 23: this figure shows the per-step latency and speculation success rate for Chinchilla (a 70B model from DeepMind) with a 4B parameter drafter (small model). For XSum (a natural language dataset), the ideal amount of speculation is about 3-4 tokens ahead, while HumanEval (a coding dataset) is more predictable and sees wins from more aggressive speculation.

How does this work for non-greedy decoding? This is a bit more complicated, but essentially boils down to a Metropolis-Hastings inspired algorithm where have $P_{\text{draft model}}(\text{chosen token})$ and $P_{\text{target model}}(\text{chosen token})$ derived from the logits, and reject the chosen token probabilistically if the ratio of these probabilities is smaller than some threshold.

These two papers[8] [9] derived this concurrently and have good examples of how this works in practice.

> **Takeaway:** Speculative sampling is yet another powerful lever for trading throughput for better per token latency. However, in the scenario where batch size is limited (e.g. small hardware footprint or large KV caches), it becomes a win-win.

7.3 Distributing Inference Over Multiple Accelerators

So far we've handwaved how we're scaling beyond a single chip. Following Section 5, let's explore the different strategies available to us and their tradeoffs. As always, we will look at prefill and generation separately.

7.3.1 Prefill

From a roofline standpoint, **prefill is almost identical to training** and almost all the same techniques and tradeoffs apply — model (Megatron) parallelism, sequence sharding (for sufficiently long context), pipelining, even FSDP are all viable! You just have to keep the KVs kicking around so you can do generation later. As in training, increasing the number of chips gives us access to more FLOPs/s (for potentially lower TTFT), but adds communication overhead (potentially reducing throughput per chip).

The general rule for sharding prefill: here's a general set of rules for prefill. We'll assume we're doing prefill on a single sequence only (no batch dimension):

1. *Model sharding:* We typically do some amount of model parallelism first, up to the point we become ICI-bound. As we saw in Section 5, this is around $F/2550$ for 1 axis (usually around 4-8 way sharding).
2. *Sequence parallelism:* Beyond this, we do sequence parallelism (like data parallelism but sharding across the sequence dimension). While sequence parallelism introduces some extra communication in attention, it is typically fairly small at longer contexts. As with training, we can overlap the communication and computation (using collective matmuls for Megatron and ring attention respectively).

> **Takeaway:** During prefill, almost any sharding that can work during training can work fine. Do model parallelism up to the ICI bound, then do sequence parallelism.

7.3.2 Generation

Generation is a more complicated beast than prefill. For one thing, it is harder to get a large batch size because we need to batch many requests together. Latency targets are lower. Together, these mean we are typically more memory-bound and more sensitive to communication overhead, which restrict our sharding strategies:

[8] https://arxiv.org/abs/2211.17192
[9] https://arxiv.org/abs/2302.01318

1. **FSDP is impossible:** since we are memory-bound in loading our parameters and KV caches from HBM to the MXU, we do not want to move them via ICI which is orders of magnitudes slower than HBM. *We want to move activations rather than weights.* This means methods similar to FSDP are usually completely unviable for generation.[10]
2. **There is no reason to do data parallelism:** pure data parallelism is unhelpful because it replicates our parameters and doesn't help us load parameters faster. You're better off spinning up multiple copies of the model instead.[11]
3. **No sequence = no sequence sharding.** Good luck sequence sharding.

This mostly leaves us with variants of model sharding for dense model generation. As with prefill, the simplest thing we we can do is simple model parallelism (with activations fully replicated, weights fully sharded over hidden dimension for the MLP) up to 4-8 ways when we become ICI bound. However, since we are often memory bandwidth bound, we can actually go beyond this limit to improve latency!

Note on ICI bounds for generation: during training we want to be compute-bound, so our rooflines look at when our ICI comms take longer than our FLOPs. However, during generation, if we're memory bandwidth bound by parameter loading, we can increase model sharding beyond this point and improve latency at a minimal throughput cost. More model sharding gives us more HBM to load our weights over, and our FLOPs don't matter.[12] Let's look at how much model parallelism we can do before it becomes the bottleneck.

$$T_{\text{HBM comms}} = \frac{2DF}{Y \cdot W_{\text{hbm}}} \qquad T_{\text{ICI comms}} = \frac{2BD}{W_{\text{ici}}}$$

$$T_{\text{ICI comms}} > T_{\text{HBM comms}} \rightarrow \frac{W_{\text{hbm}}}{W_{\text{ici}}} > \frac{F}{Y \cdot B} \rightarrow Y > F/(B \cdot \beta)$$

where $\beta = W_{\text{hbm}}/W_{\text{ici}}$. This number is usually around 8 for TPU v5e and TPU v6e. That means e.g. if F is 16,384 and B is 32, we can in theory do model parallelism up to `16384 / (32 * 8) = 64` ways without a meaningful hit in throughput. This assume we can fully shard our KV caches 64-ways which is difficult: we discuss this below.

For the attention layer, we also model shard attention W_Q and W_O over heads Megatron style. The KV weights are quite small, and replicating them is often cheaper than sharding beyond K-way sharding.

Takeaway: Our only options during generation are variants of model parallelism. We aim to move activations instead of KV caches or parameters, which are larger. When our batch size is large, we do model parallelism up to the FLOPs-ICI bound (F/α). When our batch size is smaller, we can improve latency by model sharding more (at a modest throughput cost). When we want to model shard more ways than we have KV heads, we can shard our KVs along the batch dimension as well.

7.3.3 Sharding the KV cache

We also have an additional data structure that needs to be sharded — the KV cache. Again, we almost always prefer to avoid replicating the cache, since it is the primary source of attention latency. To do this,

[10] Accidentally leaving it on after training is an easy and common way to have order of magnitude regressions

[11] By this we mean, spin up multiple servers with copies of the model at a smaller batch size. Data parallelism at the model level is strictly worse.

[12] In the sense that FLOPs time isn't bottlenecking us, so the thing we need to worry about is ICI time exceeding parameter loading time.

we first Megatron-shard the KVs along the head dimension. This is limited to K-way sharding, so for models with a small number of heads, we shard the head dimension as much as possible and then shard along the batch dimension, i.e. KV$[2, B_Z, S, K_Y, H]$. This means the KV cache is completely distributed.

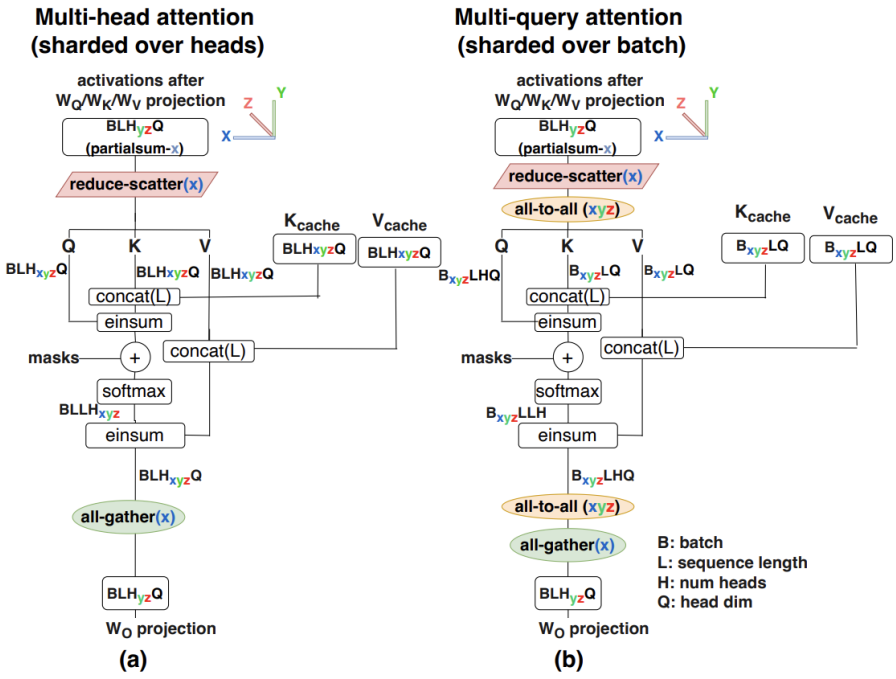


Figure 24: Comparison of the attention mechanism with (a) Multi head attention with pure model sharding and (b) Multiquery attention with batch sharding of the KV cache. Notice how we need two extra all to alls to shift the activations from model sharding to batch sharding, so they can act on the KV caches.

The cost of this is two AllToAlls every attention layer — one to shift the Q activations to the batch sharding so we can compute attention with batch sharding, and one to shift the batch sharded attention output back to pure model sharded.

Here we'll write out the full attention algorithm with model parallelism over both Y and Z. I apologize for using K for both the key tensor and the KV head dimension. Let $M = N/K$.

1. X[B, D] = ... (existing activations, unsharded from previous layer)
2. $K[B_Z, S, K_Y, H]$, $V[B_Z, S, K, H]$ = ... (existing KV cache, batch sharded)
3. $Q[B, N_{YZ}, H] = X[B, D] * W_Q[D, N_{YZ}, H]$
4. $Q[B_Z, N_Y, H] = \textbf{AllToAll}_{Z->B}(Q[B, N_{YZ}, H])$
5. $Q[B_Z, K_Y, M, H] = \textbf{Reshape}(Q[B_Z, N_Y, H])$
6. $O[B_Z, S, K_Y, M] = Q[B_Z, K_Y, M, H] \ {}_H \ K[B_Z, S, K_Y, H]$
7. $O[B_Z, S, K, M] = \textbf{Softmax}_S(O[B_Z, S, K_Y])$
8. $O[B_Z, K_Y, M, H] = O[B_Z, S, K, M] *_S V[B_Z, S, K_Y, H]$
9. $O[B, K_Y, M_Z, H] = \textbf{AllToAll}_{Z->M}(O[B_Z, K_Y, M, H])$
10. $O[B, N_{YZ}, H] = \textbf{Reshape}(O[B, K_Y, M_Z, H])$
11. $X[B, D] \{U_{YZ}\} = W_O[N_{YZ}, H, D] *_{N,H} O[B, N_{YZ}, H]$
12. $X[B, D] = \textbf{AllReduce}(X[B, D] \{ U_{YZ}\})$

This is pretty complicated but you can see generally how it works. The new comms are modestly expensive since they operate on our small activations, while in return we save a huge amount of memory bandwidth loading the KVs (which are stationary).

Sequence sharding: If the batch size is too small, or the context is long, we can sequence shard the KV cache. Again, we pay a collective cost in accumulating the attention across shards here. First we need to AllGather the Q activations, and then accumulate the KVs in a similar fashion to Flash Attention.

7.4 Designing an Effective Inference Engine

So far we've looked at how to optimize and shard the individual prefill and generate operations efficiently in isolation. To actually use them effectively, we need to design an inference engine which can feed these two operations at a point of our choosing on the latency/throughput Pareto frontier.

The simplest method is simply to run a batch of prefill, then a batch of generations:

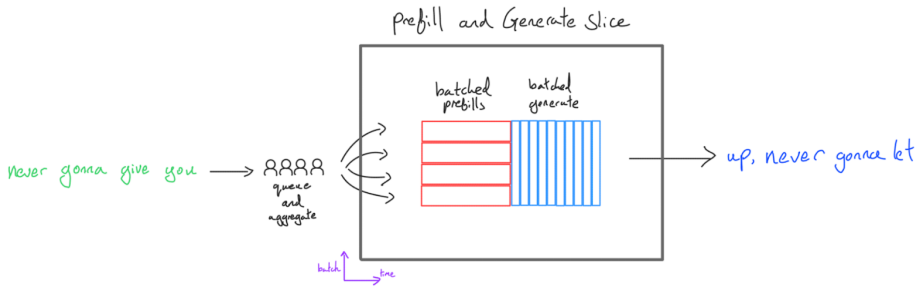


Figure 25: in the simplest setup, requests are aggregated, and the server alternates between running a batch of prefills and calling the generate function until completion for all sequences.

This is easy to implement and is the first inference setup in most codebases, but it has multiple drawbacks:

1. **Latency is terrible.** We couple the prefill and generate batch size. Time to first token (TTFT) is terrible

at big prefill batch sizes — you need to finish all prefills before any users can see any tokens. Generate throughput is terrible at small batch sizes.

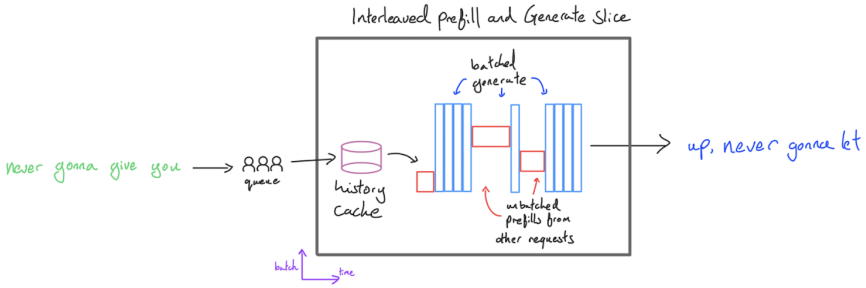
2. **We block shorter generations on longer ones.** Many sequences will finish before others, leaving empty batch slots during generation, hurting generate throughput further. The problem exacerbates as batch size and generation length increases.

3. **Prefills are padded. Prefills are padded to the longest sequence and we waste a lot of compute.** There are solutions for this, but historically XLA made it quite difficult to skip these FLOPs. Again this becomes worse the bigger the batch size and prefill sequence length.

4. **We're forced to share a sharding between prefill and generation.** Both prefill and generate live on the same slice, which means we use the same topology and shardings (unless you keep two copies of the weights) for both and is generally unhelpful for performance e.g. generate wants a lot more model sharding.

Therefore this method is only recommended for edge applications (which usually only cares about serving a single user and using hardware with less FLOPs/byte) and rapid iteration early in the lifecycle of a Transformer codebase (due to its simplicity).

A slightly better approach involves performing prefill at batch size 1 (where it is compute-bound but has reasonable latency) but batch multiple requests together during generation:

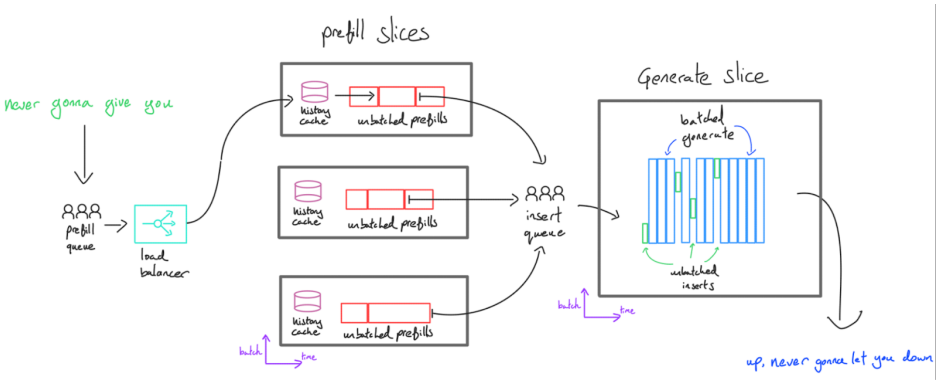


This will avoid wasted TTFT from batched prefill while keeping generation throughput high. We call this an **interleaved** configuration, since we "interleave" prefill and generation steps. This is very powerful for bulk generation applications like evaluations where throughput is the main goal. The orchestrator can be configured to prioritise prefill the moment any generation slots open up, ensuring high utilisation even for very large generation batch sizes. We can also avoid padding our prefill to the maximum length, since it isn't batched with another request.

The main disadvantage is that when the server is performing a prefill, the generation of all other requests pauses since all the compute resources will be consumed by the prefill. User A whose response is busy decoding will be blocked by user B whose prefill is occurring. This means even though TTFT has improved, the token generation will be jittery and slow on average, which is not a good user experience for many applications — other user's prefills are on the critical path of the overall latency of a request.

To get around this, we separate decode and prefill. While Transformer inference can be done on one server, it is often better from a latency standpoint to execute the two different tasks on two sets of TPUs/GPUs.

Prefill servers generate KV caches that get sent across the network to the generate servers, which batch multiple caches together and generate tokens for each of them. We call this **"disaggregated"** serving.



This provides a few advantages:

1. **Low latency at scale:** A user's request never blocks on another user's, except if there is insufficient prefill capacity. The request should be immediately prefilled, then sent to the generation server, then immediately slotted into the generation buffer. If we expect many concurrent requests to come in, we can scale the number of prefill servers independently from the number of generate servers so users are not left in the prefill queue for an extended period of time.
2. **Specialization:** Quite often, the latency-optimal parameter sharding strategy/hardware topology for prefill and generate is quite different (for instance, more model parallelism is useful for generate but not prefill). Constraining the two operations to use the same sharding hurts the performance of both, and having two sets of weights uses memory. Also, by moving prefill onto its own server, it doesn't need to hold any KV caches except the one it's currently processing. That means we have a lot more memory free for history caching (see the next section) or optimizing prefill latency.

One downside is that the KV cache now needs to be shifted across the network. This is typically acceptable but again provides a motivation for reducing KV cache size.

> **Takeaway:** for latency-sensitive, high-throughput serving, we typically have to separate prefill and generation into separate servers, with prefill operating at batch 1 and generation batching many concurrent requests together.

7.4.1 Continuous Batching

Problem (2) above motivates the concept of **continuous batching**. We optimise and compile:

- A number of prefill functions with variable context lengths and inserts it into some KV buffer, some maximum batch size and context length/number of pages.
- A generate function which takes in the KV cache, and performs the generation step for all currently active requests.

We then combine these functions with an orchestrator which queues the incoming requests, calls prefill and generate depending on the available generate slots, handles history caching (see next section) and streams the tokens out.

7.4.2 Prefix Caching

Since prefill is expensive and compute-bound (giving us less headroom), one of the best ways to reduce its cost is to do less of it. Because LLMs are autoregressive, the queries ["I", "like", "dogs"] and ["I", "like", "cats"] produce KV caches that are identical in the first two tokens. What this means is that, in principle, if we compute the "I like dogs" cache first and then the "I like cats" cache, we only need to do 1 / 3 of the compute. We can save most of the work by reusing the cache. This is particularly powerful in a few specific cases:

1. **Chatbots:** most chatbot conversations involve a back-and-forth dialog that strictly appends to itself. This means if we can save the KV caches from each dialog turn, we can skip computation for all but the newest tokens.
2. **Few-shot prompting:** if we have any kind of few-shot prompt, this can be saved and reused for free. System instructions often have this form as well.

The only reason this is hard to do is memory constraints. As we've seen, KV caches are big (often many GB), and for caching to be useful we need to keep them around until a follow-up query arrives. Typically, any unused HBM on the prefill servers can be used for a local caching system. Furthermore, accelerators usually have a lot of memory on their CPU hosts (e.g. a 8xTPUv5e server has 128GiB of HBM, but around 450GiB of Host DRAM). This memory is much slower than HBM — too slow to do generation steps usually — but is fast enough for a cache read. In practice:

- Because the KV cache is local to the set of TPUs that handled the initial request, we need some form of affinity routing to ensure follow-up queries arrive at the same replica. This can cause issues with load balancing.
- A smaller KV cache is helpful (again) — it enables us to save more KV caches in the same amount of space, and reduce read times.
- The KV cache and their lookups can be stored quite naturally in a tree or trie. Evictions can happen on an LRU basis.

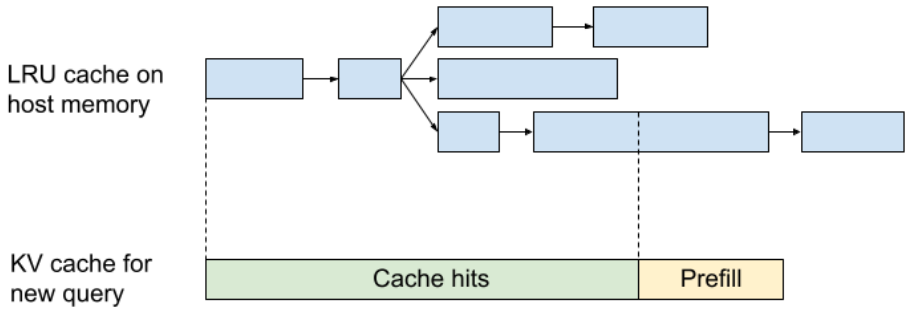


Figure 26: KV prefix cache implemented as an LRU tree. Source: Character.ai blog.

7.4.3 Let's look at an implementation: JetStream

Google has open-sourced a library that implements this logic called JetStream[13]. The server has a set of "prefill engines" and "generate engines", usually on different TPU slices, which are orchestrated by a single

[13] https://github.com/google/JetStream

controller. Prefill happens in the "prefill thread", while generation happens in the "generate thread". We also have a "transfer thread" that orchestrates copying the KV caches from the prefill to generate slices.

The Engine interface is a generic interface that any LLM must provide. The key methods are:

- **prefill:** takes a set of input tokens and generates a KV cache.
- **insert:** takes a KV cache and inserts it into the batch of KV caches that generate is generating from.
- **generate:** takes a set of batched KV caches and generates one token per batch entry, appending a single token's KV cache to the decode state for each token.

We also have a PyTorch version of JetStream available [14].

7.5 Worked Problems

I'm going to invent a new model based on LLaMA-2 13B for this section. Here are the details:

hyperparam	value
n_{layers} (L)	64
d_{model} (D)	4,096
d_{ff} (F)	16,384
n_{heads} (N)	32
$n_{\text{kv\_heads}}$ (K)	8
d_{qkv} (H)	256
$n_{\text{embeddings}}$ (V)	32,128

Exercise 7.1

How many parameters does the above model have? How large are its KV caches per token? *You can assume we share the input and output projection matrices.*

Exercise 7.2

Let's say we want to serve this model on a TPUv5e 4x4 slice and can fully shard our KV cache over this topology. What's the largest batch size we can fit, assuming we use int8 for everything. What if we dropped the number of KV heads to 1?

Exercise 7.3

Let's pretend we're totally HBM bandwidth bound. How long does it take to load all the parameters into the MXU from HBM? *This is a good lower bound on the per-step latency.*

[14] `https://github.com/google/jetstream-pytorch`

Exercise 7.4

Let's say we want to serve this model on a TPUv5e 4x4 slice. How would we shard it? *Hint: maybe answer these questions first:*

1. What's the upper bound on tensor parallelism for this model over ICI?
2. How can we shard the KV caches?

For this sharding, what is the rough per-step latency for generation?

Exercise 7.5

Let's pretend the above model is actually an MoE. An MoE model is effectively a dense model with E copies of the FFW block. Each token passes through k of the FFW blocks and these `k` are averaged to produce the output. Let's use `E=16` and `k=2` with the above settings.

1. How many parameters does it have?
2. What batch size is needed to become FLOPs bound?
3. How large are its KV caches per token (assume no local attention)?
4. How many FLOPs are involved in a forward pass with T tokens?

Exercise 7.6

With MoEs, we can do "expert sharding", where we split our experts across one axis of our mesh. In our standard notation, our first FFW weight has shape [E, D, F] and we shard it as $[E_Z, D_X, F_Y]$ where X is only used during training as our FSDP dimension. Let's say we want to do inference on a TPU v5e:

1. What's the HBM weight loading time for the above model on a TPU v5e 8x16 slice with Y=8, Z=16? How much free HBM is available per TPU?
2. What is the smallest slice we could fit our model on?

Exercise 7.7

Here we'll work through the math of what the ESTI paper[15] calls 2D weight-stationary sharding. We describe this briefly in Appendix B, but try doing this problem first to see if you can work out the math. The basic idea of 2D weight stationary sharding is to shard our weights along both the D and F axes so that each chunk is roughly square. This reduces the comms load and allows us to scale slightly farther.

Here's the algorithm for 2D weight stationary:

1. $In[B, D_X] = \textbf{AllGather}_{YZ}(In[B, D_{XYZ}])$
2. $Tmp[B, F_{YZ}] \{U_X\} = In[B, D_X] *_D W_{in}[D_X, F_{YZ}]$
3. $Tmp[B, F_{YZ}] = \textbf{AllReduce}_X(Tmp[B, F_{YZ}] \{U_X\})$
4. $Out[B, D_X] \{U_{YZ}\} = Tmp[B, F_{YZ}] *_F W_{out}[F_{YZ}, D_X]$
5. $Out[B, D_{XYZ}] = \textbf{ReduceScatter}_{YZ}(Out[B, D_X] \{U_{YZ}\})$

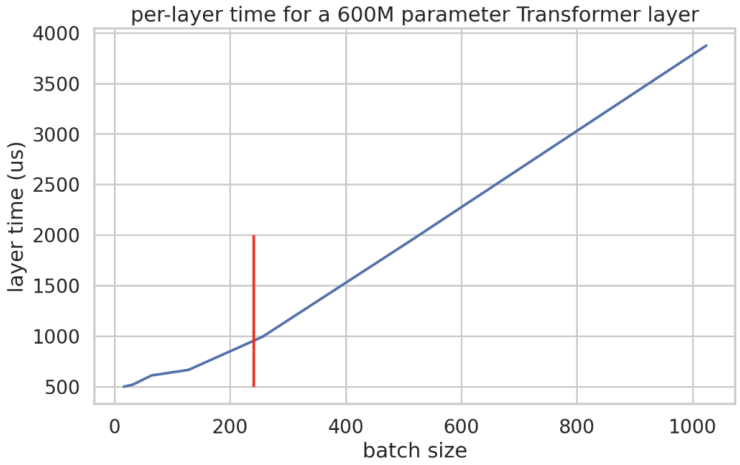
[15] https://arxiv.org/pdf/2211.05102

Your goal is to work out T_{math} and T_{comms} for this algorithm and find when it will outperform traditional 3D model sharding?

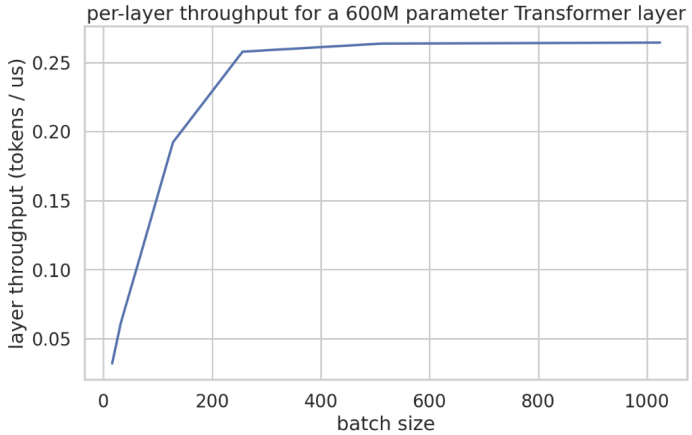
7.6 Appendix

7.6.1 Appendix A: How real is the batch size > 240 rule?

The simple rule we provided above, that our batch size must be greater than 240 tokens to be compute-bound, is roughly true but ignores some ability of the TPU to prefetch the weights while other operations are not using all available HBM, like when doing inter-device communication.
Here's an empirical plot of layer time (in microseconds) for a small Transformer with d_{model} 8192, d_{ff} 32768, and only 2 matmuls per layer. This comes from a colab that is linked in the online version of this book. You'll see that step time increases very slowly up until around batch 240, and then increases linearly.



Here's the actual throughput in tokens / us. This makes the argument fairly clearly. Since our layer is about 600M parameters sharded 4 ways here, we'd expect a latency of roughly 365us at minimum.

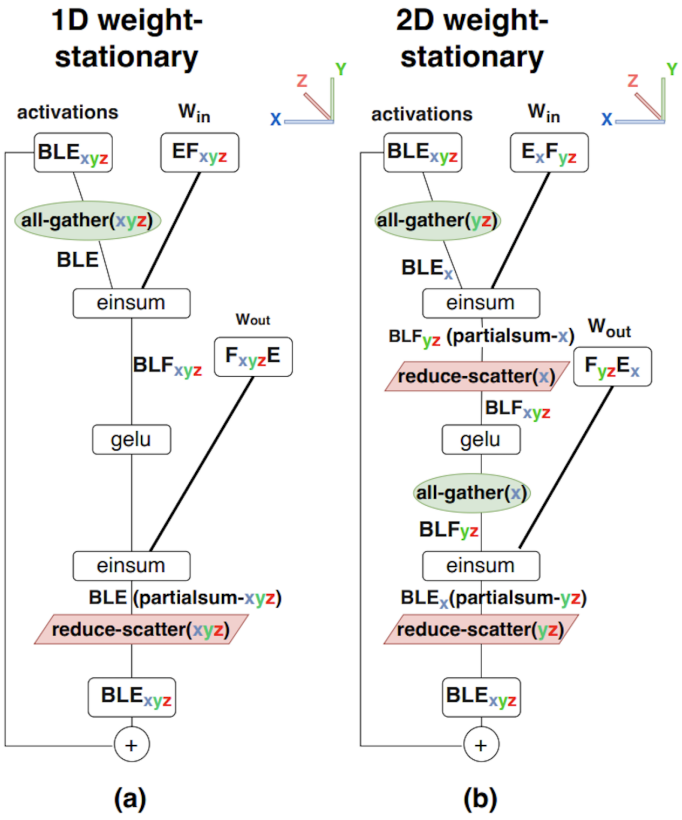


So at least in this model, we do in fact see throughput increase until about BS 240 per data parallel shard.

7.6.2 Appendix B: 2D Weight Stationary sharding

As the topology grows, if we have access to higher dimensional meshes (like that of TPUs) it is possible to refine this further with "**2D Weight Sharding**". By introducing a second sharding axis. We call this "**2D Weight Stationary**", and was described in more detail in the Efficiently Scaling Transformer Inference paper[16].

Because we're only sharding the hidden F dimension in Megatron, it can become significantly smaller than E (the d_{model} dimension) once the number of chips grows large with 1D sharding. This means at larger batch sizes, it can be more economical to perform a portion of the collectives over the hidden dimension after the first layer of the MLP is applied.



This figure shows:

1. 1D weight-stationary sharding, a.k.a. Pure Megatron sharding, where activations are fully replicated after AllGather, and weights are fully sharded over the hidden F dimension.
2. 2D weight stationary sharding, where weights are sharded over both the hidden F and reduction E dimension, and activations are sharded over the E dimension. We perform an AllGather on the (yz) axis before the first layer, then ReduceScatter on the (x) axis.

For the attention layer, Megatron style sharding is also relatively simple for smaller numbers of chips. However, Megatron happens over the n_{heads} dimension, which puts a limit on the amount of sharding that is possible. Modifying the 2D sharding with for (instead of sharding the hidden, we shard the n_{heads} dimension), we gain the ability to scale further.

[16] https://arxiv.org/abs/2211.05102

7.6.3 Appendix C: Latency bound communications

As a recap, in Section 3 we derived the amount of time it takes to perform an AllGather into a tensor of size B on each TPU, over X chips on a 1D ring links of full duplex bandwidth of W_{ICI} and latency T_{min}.

$$T_{total} = \max\left(\frac{T_{min} \cdot |X|}{2}, \frac{B}{W_{ICI}}\right)$$

For large B, the wall clock stays relatively constant because as you add more chips to the system, you simultaneously scale the amount of data movement necessary to perform the operation and the total bandwidth available.

Because of the relatively low amounts of data being moved during latency optimized inference, collectives on activations are often bound by the latency term (especially for small batch sizes). One can visualise the latency quite easily, by counting the number of hops we need to complete before it is completed.

On TPUs, if the tensor size-dependent part of communication is less than 1 microsecond per hop (a hop is communication between two adjacent devices) we can be bottlenecked by the fixed overhead of actually dispatching the collective. With `4.50e10` unidirectional ICI bandwidth, ICI communication becomes latency bound when: $(\text{bytes}/n_{\text{shards}})/4.5e10 < 1e-6$. For 8-way Megatron sharding, this is when `buffer_size < 360kB`. **This actually is not that small during inference:** with `BS=16` and `D=8192` in int8, our activations will use `16*8192=131kB`, so we're already latency bound.

> **Takeaway:** our comms become latency bound when total bytes $< W_{ICI} \times 1e-6$. For instance, with model parallelism over Y, we become bound in int8 when $Y > BD/45,000$.

There's a parallel to be drawn here with the compute roofline — we are incurring the fixed cost of some small operations (latency for comms, memory bandwidth for matmuls).

8 Serving LLaMA 3-70B on TPUs

Let's take a close look at how we'd serve LLAMA 3-70B models on TPU v5e. How expensive are different models to serve at roofline? How large are their KV caches? What batch sizes should we use? How are the parameters and activations sharded during inference? Let's work through some back-of-the-envelope estimates for latency and throughput in production.

8.1 What's the LLaMA Serving Story?

Let's remind ourselves what LLAMA 3-70B looks like (see Section 6 for reference):

hyperparam	value
n_{layers} (L)	80
d_{model} (D)	8,192
f_{ff} (F)	28,672
n_{heads} (N)	64
$n_{\text{kv heads}}$ (K)	8
d_{qkv} (H)	128
$n_{\text{embeddings}}$ (V)	128,256

Let's start with a simple question: **what hardware should we serve on?** The answer is basically, whichever is cheapest in FLOPs / dollar.[1] For this reason, we typically want to serve on TPU v5e, our current dedicated inference chip (cost comes from Google Cloud pricing[2] as of February 2025):

TPU type	bfloat16 FLOPs/s	Google Cloud USD / hour	FLOPs / $
H100	9.9e14	$10.8	3.3e17
v5p	4.59e14	$4.20	3.9e17
v5e	1.97e14	$1.20	**5.8e17**

Each TPU v5e has 16GB of HBM which will require us to shard our model fairly aggressively. Let's start by thinking about some basic quantities that might matter for us:

Question: How large are LLaMA 3-70B's KV caches per token? *You can assume we store them in int8. This determines how large our batch size can be on a given topology.*

LLaMA 3-70B has 8 KV heads, so the size per token is `2 * K * H * L = 2 * 8 * 128 * 80 = 160kB`.

Note just how big this is! If we have a sequence length of 32k tokens (as is common), this uses `162e3 * 32,768 = 5.3GB / sequence`. For BS=240, this is 1.3TB! Since TPU v5e only have 16GB a piece, we would need about `(70e9 + 1.3e12) / 16e9 = 86` TPU v5e chips to even fit this much memory. Also note how

[1] This isn't always true, sometimes more HBM or ICI bandwidth is critical rather than FLOPs, but this is a good heuristic.

[2] `https://cloud.google.com/tpu/pricing`

large this is compared to the 70GB of model parameters.

Question: Let's say we want to serve L3 70B at batch size 32 and 8192 sequence length with everything (params and KVs) in int8. How much total memory will this use? What's the smallest slice we could serve this on?

Since our KVs are `160e3` bytes in int8, our total KV memory is `160e3 * 8192 * 32 = 41.9e9` bytes. Our parameters are `70e9` bytes, since we have 1 byte per parameter. Thus, our total memory usage is `41.9e9 + 70e9 = 112GB`.

The smallest slice we could use would have `112e9 / 16e9 = 7` TPUs, or (rounding to an even size), TPU v5e `4x2`. This will be a tight fit and we might not be able to quite fit this accounting for other overhead, so we might need a `4x4` at minimum (or to drop the batch size).
Question: At this batch size and quantization on a TPU v5e `4x2`, roughly what latency would we expect per decode step? What throughput (tokens / sec / chip). What about a `4x4`? *Assume we perform our FLOPs in bfloat16 and everything is fully sharded.*
We can invoke the formula from the previous section that

$$\text{Theoretical Step Time (General)} = \underbrace{\frac{\text{Batch Size} \times \text{KV Cache Size}}{\text{Total Memory Bandwidth}}}_{\text{Attention (always bandwidth-bound)}} + \underbrace{\max\left(\frac{2 \times \text{Batch Size} \times \text{Parameter Count}}{\text{Total FLOPs/s}}, \frac{\text{Parameter Size}}{\text{Total Memory Bandwidth}}\right)}_{\text{MLP (can be compute-bound)}}$$

Here our critical batch size will be about 120 since our parameters are in int8 but our FLOPs are in bfloat16. We could also manually calculate the RHS maximum, but that's basically a calculation we've already done several times. **So we're well into the memory-bound regime for both our matmul and our FLOPs.**

Strictly looking at memory bandwidth then, our step time is basically `(KV size + param size) / (8 * HBM bandwidth) = 112e9 / (8 * 8.1e11) = 17ms`. **So theoretically our step time is about 17ms.** Our throughput would be `32 / .017 = 1882 tokens / sec`, or `1882 / 8 = 235 tokens / sec / chip`.

There's one caveat here which is to check if we might be ICI bound on our matmuls. We could dedicate 2 axes to it here, so we're ICI bound in theory when $Y > 2 * F/2550 = 2 * 28672/2550 = 22$, so we're golden!

If we were to run on a `4x4`, we'd still be fine ICI-wise, so our latency would drop to `17 / 2 = 8.5ms`, but our throughput per-chip would remain the same.

8.1.1 Thinking about throughput

Let's spend a little time thinking purely about throughput. When we optimize for throughput, we want to be compute bound, meaning we come close to utilizing all the TPU MXU capacity. Typically that means we want the batch size to be as large as possible, so we are doing as much work as possible.

Question: On TPU v5e, using bfloat16 weights and activations, how large do our batch sizes need to be for us to be compute-bound in our matmuls? What if we do int8 weights but perform our FLOPs in bfloat16? What about int8 weights with int8 FLOPs?

As discussed in Section 7, for any bfloat16 matmul for which $B \ll D, F$ we have

$$T_{\text{math}} > T_{\text{comms}} \leftrightarrow \frac{2BDF}{2DF} \geq \frac{\text{TPU bfloat16 FLOPs/s}}{\text{HBM bandwidth}} = 240$$

When our weights are in int8, we lose a factor of 2 in the denominator, so we have $2BDF / DF = 2B > 240$, or equally $B > 120$, half the critical batch size from before. That's really helpful for us! When we do int8 weights and int8 FLOPs, we have to use the int8 value for TPU FLOPs/s, which goes from `1.97e14` for bfloat16 to `3.94e14`, nearly double. That means we're back where we started at about $B > 240$.

The case of int8 weights and bfloat16 FLOPs is quite common, since quantizing parameters losslessly is often easier than doing low-precision arithmetic.

Question: What is the smallest TPU v5e topology we could serve LLaMA 3-70B on using bfloat16, int8, and int4 (both KVs and parameters) with 8k context? *You can think of KV caches as negligibly small for this one.*

This is easy! If we're OK with a tiny batch size then the only limit is fitting parameter memory in HBM, i.e. it is just `ceil(num_params * sizeof(dtype) / HBM per TPU`, or `ceil(70e9 * sizeof(dtype) / 16e9)` rounded to the nearest reasonable topology (some multiple of 2):

dtype	param size	KV size / token (bytes)	min TPU v5es	actual min slice	remaining HBM for KV caches	num KV caches @ 8k
bf16	140GB	324kB	8.75	4x4 = 16 chips	116	43
int8	70GB	162kB	4.38	4x2 = 8 chips	68	52
int4	45GB	81kB	2.81	2x2 = 4 chips	19	67

That's pretty cool! It tells us we could fit LLaMA 70B on a TPU v5e 2x2 if we wanted to. Except you'll notice the number of KV caches is very small. That's our batch size! That means we'll be getting terrible FLOPs utilization. We'd be very happy to use a larger topology in order to push our batch size up to 240.

Question: Assume we use the largest batch size that fits on these topologies, what latency we could expect for each generate step?

This is also easy, since we're picking our batch size to fill up all our HBM! This is just a question of how long it takes to load a full TPU v5e's worth of bytes into the MXU. This is just `v5e HBM / v5e HBM memory bandwidth = 16GB / 8.2e11 = 19ms`, so this is **19ms / step**. Assuming our generations have a median length of 512 tokens, that is about 9s for each decode. Note that we could get marginally better latency with a smaller batch size, for instance if we only looked at model parameters in int4 our minimum latency is about 10ms / step, since HBM is no longer full.

Takeaway: we can always lower bound decode latency by asking how long it takes to load all the model's parameters from HBM into the MXU. When our KV caches are small, you can think about each layer as just loading the weights chunk-by-chunk and then discarding them. Unless we're using large batch sizes or lots of inter-device comms, this is often a reasonable bound (within 1.5x). When our batch size is bigger, we need to model the KV cache loading as well, since that dominates the parameters.

Likewise, in the FLOPs-bound regime (e.g. training or big-batch inference), we can use the Total FLOPs$/(N \cdot C) = 2 \cdot$ param count $\cdot B/(N \cdot C)$ lower bound, which assumes no communication.

Question: For each of these, what throughput per chip does this give us (in terms of queries / chip)? *You can assume our median decode length is 512 tokens.*

This is an important question because it's exactly correlated with cost / token.

With our assumption about median decode length, our throughput is just $B/$(per-step latency$\cdot$median steps$\cdot N) \cong 43/(0.019 * 512 * N)$. This gives us roughly $(4.42/N)$ QPS, so plugging in N we get:

dtype	QPS / chip
bf16	0.27
int8	0.66
int4	1.72

Note that this is rather optimistic since it totally ignores the working memory of the forward pass (memory allocated to activations and attention). This is not ridiculous with Flash Attention, but it is also not realistic. The real numbers are likely maybe 1/2 of this. For absolutely maximum throughput we would probably want to more than double the number of chips and increase the batch size significantly as well.

Question: How would our peak throughput change if we doubled our topology for each of the above examples?

If we used a 4x8 slice in bfloat16, we would have 186GB remaining for KV caches, which would let us up our batch size to 161. Then since our step time would remaining the same, we would have a throughput of `16.54 / num_chips`, or

dtype	QPS / chip
bf16 (on 4x8)	0.51
int8 (on 4x4)	1.03
int4 (on 2x4)	2.06

A further increase would give an even bigger win! The big takeaway is that **the smallest topology is not the most performant topology in all cases**, if we're limited by KV cache size.

Question: Now let's dig into the question of sharding. Let's say we wanted to serve in bfloat16 on a TPU v5e 4x8. What sharding would we use for our model on a TPU v5e 4x8 during generation? Can we avoid being communication bound?

As discussed in the previous section, we only really have one option for sharding during generation: model parallelism. How much can we do before we become communication bound? As we've discussed in the previous section, our models become communication bound roughly when

$$Y > \frac{F \cdot n_{\text{axes}}}{2550}$$

For LLaMA 3-70B we have `F = 28,672`, so if we do 2 axes of model sharding this gives us roughly $Y = 28672 \cdot 2/2550 = 22$, so in general we could scale up to about 16 chips without being communication bound, which lets us use a `4x4` but not a `4x8`. Generally, since we do not perfectly overlap computation, even this estimate is overly optimistic.

> **Takeaway:** we cannot actually serve on a 4x8 with pure model parallelism. The best we can do here is a 4x2 or *maybe* a 4x4.

However, as we've discussed, when our batch size is small we can often do more model parallelism without significantly hurting throughput, since our model is memory-bandwidth-bound and not FLOPs bound. We said before that this value is roughly $Y = F/(8 \cdot B)$, so if we did batch size 64, we could in theory go up to `Y = 28,672 / (8 * 64) = 56` way model parallelism before we become ICI-bound. To sanity check this, we can look at $T_{\text{ici comms}}$, $T_{\text{hbm comms}}$, and T_{math} for a single matmul. We clearly have:

$$T_{\text{ici comms}} = \frac{2BD}{W_{\text{ici}}} \qquad T_{\text{hbm comms}} = \frac{2DF}{Y \cdot W_{\text{hbm}}} \qquad T_{\text{math}} = \frac{2BDF}{Y \cdot C}$$

For a `4x8`, this would give us $T_{\text{ici comms}}$ = `(2 * 64 * 8192) / 9e10 = 11us`, $T_{\text{hbm comms}}$ = `(2 * 8192 * 28,672) / (32 * 8.1e11) = 18us`, and T_{math} = `(2 * 64 * 8192 * 28,672) / (32 * 1.97e14) = 4us`, so in theory we're still HBM bandwidth bound, which is great! *Note that scaling up from a `4x4` to a `4x8` probably isn't helpful from a throughput standpoint, but it'll reduce our latency!

If we look at the int8 and int4 configs, we *can* do those with pure model parallelism. So we've hit a point at which quantization actually gives us a meaningful advantage beyond faster FLOPs: it lets us use a larger batch size before we become comms-bound. **So the end of this story is that we can't achieve peak throughput on a 4x8, but for the int8 and int4 configs we could do pure model parallelism**.

> **Takeaway:** the maximum amount of useful model parallelism depends on d_{ff} and the number of axes over which you're sharding your model. The maximum value usually ranges between 8 and 32 depending on the model size. You can scale beyond this limit to improve latency at some throughput cost.

8.1.2 What About Prefill?

We've mostly ignored prefill here because it's much simpler. Let's put a couple of concepts together and think about the end-to-end picture.

Question: Assume we achieve a 40% FLOPs utilization during prefill. How long will a prefill of length 8192 take on 16 TPU v5e chips?

At 8k tokens, we are solidly compute bound, so we just need to reason about FLOPs. We know our model has `70e9` parameters so each forward pass uses `2 * 70e9 * B` FLOPs. Assuming 40% MFU (FLOPs utilization), this gives us a runtime of about `2 * 70e9 * 8192 / (16 * 1.97e14 * 0.4) = 0.91s`. Compared to the numbers we've been looking at before, that's actually quite a lot!

Question: Assume we have a median prefill length of 8192 tokens and a median decode length of 4096 tokens. Say we have a generate batch size of 32. On average how many sequences finish decoding per step? On average how many tokens are evicted from our KV cache each step?

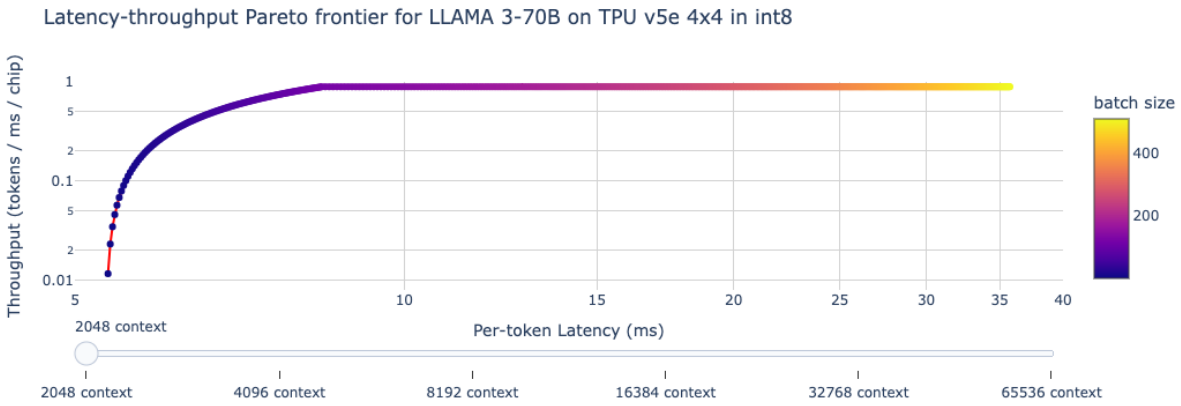
This is kind of straightforward. Since we have a median decode length of 4096 tokens, a sequence will finish roughly every 1 / 4096 tokens. Given a batch size of 32, this means we have `32 / 4096` sequences evicted per step. Since our KV cache length is roughly `8192 + 4096`, this is `32 * (8192 + 4096) / 4096 = 96` tokens evicted per step. The general formula is $B * (P + G)/G$ where P and G are the prefill and generate lengths.

Question: Assume we do disaggregated serving with a median prefill length of 8192 and a median decode length of 512. Assume the prefill and generate latencies calculated above in bfloat16. What ratio of prefill:generate servers will you need to keep both fully saturated.

This is kind of a fun question. Let P be the number of prefill servers and G be the number of generate servers. So generally speaking, this is a pipeline problem where we feed sequences in at a rate of `P / prefill_latency` and consume them at a rate of `B * G / (generate_latency * median_decode_length)`. We had calculated `910ms` per prefill step and `19ms` per decode step at batch size 43 (let's call that 32). Therefore we need `P / 0.91 = 32 * G / (0.019 * 512)` or `P = 3G`, i.e. we need about 3 times more prefill servers than generation servers!

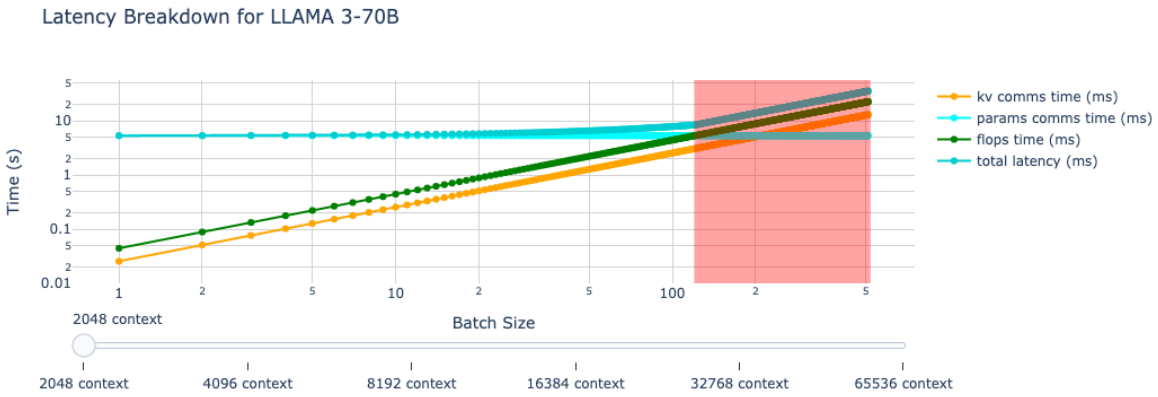
8.2 Visualizing the Latency Throughput Tradeoff

Sticking with LLAMA 70B for a second, we can calculate the theoretical peak latency and throughput at different batch sizes during generation. As we showed in the previous section for PaLM models, this gives us a Pareto frontier for throughput/latency. We assume 8-way model parallelism. We'll use a TPU v5e 4x4 topology here. The online version of the textbook provides an interactive diagram to explore this. A screenshot is provided:



- **See how dramatic the tradeoff is between cost and latency.** At the cost of doubling per-token latency, we can achieve a roughly 100x reduction in per-token cost. Also, our latency can range anywhere from 5.5ms with low batch size to 20 ms with very large batches.
- Note how at 2k context the throughput effectively plateaus at around 1 token / ms / chip when it hits the BS 120 roofline (120 here because we do int8 weights but bf16 FLOPs). As the sequence length increases, however, we can no longer fit this batch size in memory, so we never hit the point of full saturation.
- Note how much higher the latency is at large batch sizes for the same throughput, since KV loading becomes dominant (instead of parameter loading).

We can understand this better by breaking down the sources of cost and latency into param loading time, KV loading time, and FLOPs time. The red sector is the region in which we expect to be compute-bound in our MLP blocks. The online version of the textbook provides an interactive diagram to explore this. A screenshot is provided:



This tells quite a story. You can see that initially, parameter loading represents the vast majority of the latency, until the batch size becomes large enough that FLOPs and KV loading become more significant. Notably, at all sequence lengths greater than 2048, we spend more time on KV cache loading than we do on FLOPs! **So while we can improve our hardware utilization by increasing batch size, at long context lengths KV loading always dominates the total step time.**

Takeaway: for LLaMA 3-70B, we are strongly KV cache memory bandwidth-bound (and HBM-bound) in almost all of these configurations, highlighting just how important reducing KV cache size is for generation throughput. Also note just how dramatic the latency/throughput tradeoff remains here.

Here's the code for computing these rooflines:

```
import numpy as np

num_chips = 16  # we fix 16 as the amount of total model parallelism we do
param_size = 70e9  # int8 means 1 byte per param
sequence_length = 8192  # can vary this

hbm_bandwidth = 8.20E+11  # v5e
flops = 1.97E+14  # v5e

param_size = bytes_per_param * param_count

def kv_cache_size(bs):
    return 2 * bs * 128 * 8 * 80

def min_topology(bytes):
    return 2 ** np.ceil(np.log2(bytes / 16e9))

def get_max_batch_size(max_num_chips: int = 16):
  # for num_chips in topo_sizes:
  batch_sizes = np.arange(1, 1024, 4)
  kv_sizes = kv_cache_size(sequence_length * batch_sizes)
  num_chips = min_topology(kv_sizes + param_size)
  max_idx = np.where(num_chips <= max_num_chips)[0][-1]
  return max_idx

max_idx = get_max_batch_size(num_chips, sequence_length, param_size)  # get the largest batch
    size that can fit
batch_sizes = np.arange(1, 512, 1)[:max_idx]
kv_sizes = kv_cache_size(sequence_length * batch_sizes)

kv_comms_time = kv_sizes / (num_chips * hbm_bandwidth)

param_comms_time = param_size / (num_chips * hbm_bandwidth)
param_comms_time = np.asarray([param_comms_time] * batch_sizes.shape[0])

flops_time = 2 * param_count * batch_sizes / (num_chips * flops)  # roughly true in a 2ND sense

mlp_time = np.maximum(flops_time, param_comms_time)
attn_time = kv_comms_time  # always bandwidth-bound for generate

latency = 1000 * (mlp_time + attn_time)
throughput = batch_sizes / (latency * num_chips)
```

Note how we very explicitly break out latency into two sources: KV loading and param loading, and how the latency is either bound by FLOPs or comms, whichever is bigger.

8.3 Worked Problems

Here are a few worked problems. Some of these repeat things that are worked above, but might be pedagogically useful.

Exercise 8.1

How many FLOPs does each forward pass for LLAMA 3-405B use per-token? Assuming we're FLOPs bound, what is a lower bound on a single forward pass on N chips on TPU v5e? What if we're comms bound? *Ignore the fact that the model does not fit on a single chip.*

Exercise 8.2

Assume we want to serve LLAMA 3-8B with BS240 using int8 weights and int8 KV caches. How many bytes are used by (a) model parameters (b) KV caches and (c) peak working activations (roughly)? What's the smallest topology we can run this on?

Exercise 8.3

How would you serve LLAMA 3-405B on TPU v5e? Assume int8 weights and bfloat16 FLOPs. Let's say we have a firm limit of 15ms / token, what's the highest throughput configuration we could achieve? What is the theoretical minimum step time?

9 Profiling

So far this series has been entirely theoretical: back-of-the-envelope calculations based on hardware rooflines. That understanding gets you far but a lot of optimization comes down to practical details: how the XLA compiler works and how to use profiling tools like the JAX/Tensorboard Profiler to figure out what to do when it fails. We discuss this here.

9.1 A Thousand-Foot View of the TPU Software Stack

Google exposes a bunch of APIs for programming TPUs, from high level JAX code to low level Pallas or HLO. Most programmers write JAX code exclusively, which lets you write abstract NumPy-style linear algebra programs that are compiled automatically to run efficiently on TPUs.

Here's a simple example, a JAX program that multiplies two matrices together:

```
import jax
import jax.numpy as jnp

def multiply(x, y):
  return jnp.einsum('bf,fd->db', x, y)

y = jax.jit(multiply)(jnp.ones((128, 256)), jnp.ones((256, 16), dtype=jnp.bfloat16))
```

By calling `jax.jit`, we tell JAX to trace this function and emit a lower-level IR called StableHLO[1], a platform-agnostic IR for ML computation, which is in turn lowered to HLO by the XLA compiler. The compiler runs many passes to determine fusions, layouts, and other factors that result in the HLO that is observable in a JAX profile. This HLO represents all the core linear algebra operations in the JAX code (matmuls, pointwise ops, convolutions, etc) in an LLVM-style graph view. For instance, here is an abridged version of the above program as HLO[2]:

```
ENTRY %main.5 (Arg_0.1: f32[128,256], Arg_1.2: bf16[256,16]) -> f32[16,128] {
  %Arg_1.2 = bf16[256,16]{1,0} parameter(1), metadata={op_name="y"}
  %convert.3 = f32[256,16]{1,0} convert(bf16[256,16]{1,0} %Arg_1.2),
  %Arg_0.1 = f32[128,256]{1,0} parameter(0), metadata={op_name="x"}
  ROOT %dot.4 = f32[16,128]{1,0} dot(f32[256,16]{1,0} %convert.3, f32[128,256]{1,0} %Arg_0.1),
    lhs_contracting_dims={0}, rhs_contracting_dims={1},
}
```

We'll explain the syntax of HLO in just a second, but for now just note that it actually matches the JAX code above fairly well. For instance,

```
ROOT %dot.4 = f32[16,128]{1,0} dot(f32[256,16]{1,0} %convert.3, f32[128,256]{1,0} %Arg_0.1),
    lhs_contracting_dims={0}, rhs_contracting_dims={1}
```

is the actual matmul above that multiplies two f32 matrices along the 0 and 1 dimension, respectively.

To transform this HLO to code that can be executed on the TPU, the XLA compiler first lowers it to LLO (low-level optimizer) IR. LLO programs the TPU directly, scheduling copies between memories, pushing arrays onto the systolic array, etc. LLO code contains primitives that push buffers into the systolic array, pull results off, and schedule DMAs that communicate between different pieces of TPU memory. Once this has

[1] https://openxla.org/stablehlo

[2] To get this HLO, you can run `jax.jit(f).lower(*args, **kwargs).compile().as_text()`.

been lowered to LLO, it is then compiled to machine code that is loaded into the TPU IMEM and executed.

When a program is running slower than we'd like, we primarily work with the JAX level to improve performance. Doing so, however, often requires us to understand some of the semantics of HLO and how the code is actually running on the TPU. When something goes wrong at a lower level, we pull yet another escape hatch and write custom kernels in Pallas[3]. To view the HLO of a program and its runtime statistics, we use the JAX profiler.

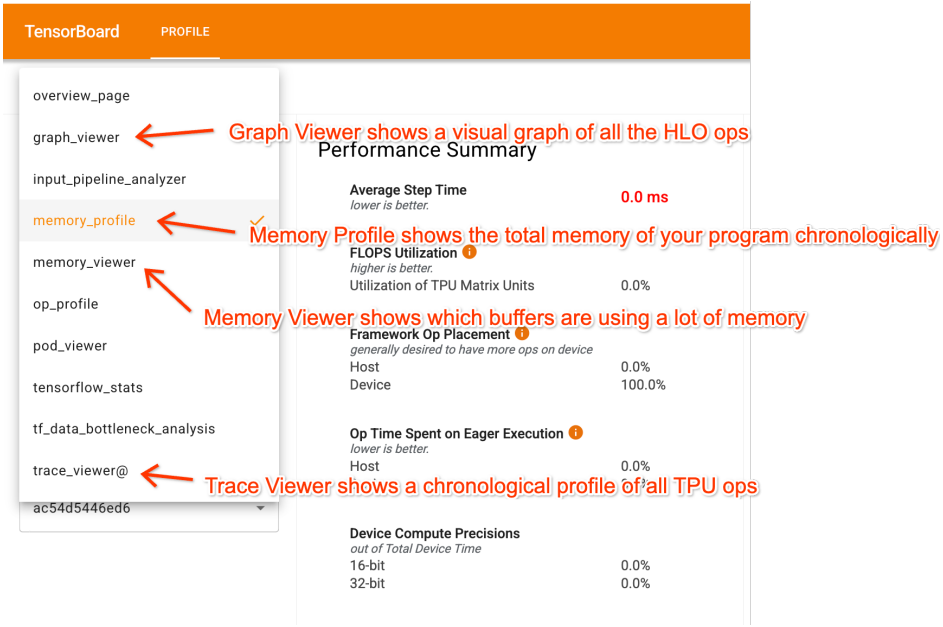
9.2 The JAX Profiler: A Multi-Purpose TPU Profiler

JAX provides a multi-purpose TPU profiler with a bunch of useful tools for understanding what's happening on the TPU when a program is run. You can use the `jax.profiler` module to trace a program as it's running and record everything from the duration of each subcomponent, the HLO of each program, memory usage, and more. For example, this code will dump a trace to a file in `/tmp/tensorboard` that can be viewed in TensorBoard.

```
import jax
with jax.profiler.trace("/tmp/tensorboard"):
  key = jax.random.key(0)
  x = jax.random.normal(key, (1024, 1024))
  y = x @ x
  y.block_until_ready()

# Now you can load TensorBoard in a Google Colab with
#
# !pip install tensorboard-plugin-profile
# %load_ext tensorboard
# %tensorboard --logdir=/tmp/tensorboard
#
# or externally with
#
# > tensorboard --logdir=/tmp/tensorboard
#
```

Here's an overview of what you can do in the profiler:

[3] https://jax.readthedocs.io/en/latest/pallas/tpu/details.html



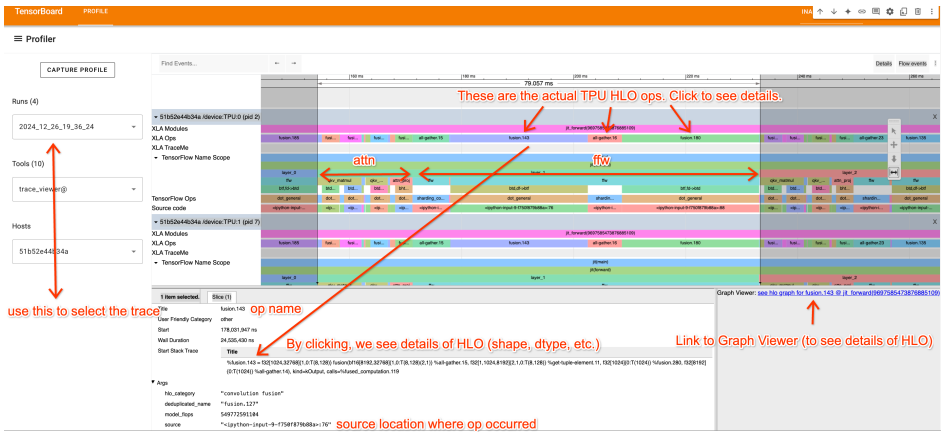
Once in TensorBoard, the profiler has a few key tabs that help you understand your program:

1. **Trace Viewer** shows a detailed timeline of what's actually happening on the TPU as a timeline.
2. **Graph Viewer** shows the HLO graph, letting you see what parts of the program feed into each other and how things are sharded.
3. **Memory Profile and Memory Viewer:** these show how much memory your program is using.

While it's slightly difficult to share profiles, the online version of this contains a Perfetto link that contains at least the Trace Viewer component for a simple Transformer. There is also a colab provided that lets you generate the full JAX/TensorBoard trace and play around with it.

9.2.1 Trace Viewer

The Trace Viewer is probably the most useful part of the profiler. The example below shows a simple Transformer with pieces annotated. Names come from labels provided in the code.



The Trace Viewer shows a chronological timeline of all the actions on each TPU core. We're only looking at TPU:0 here, since typically all TPUs execute the same instructions. A few key notes:

1. The top row (XLA Ops) shows the actual TPU operations (the names are HLO names). Everything else is an approximate trace based on `jax.named_scope`, `jax.named_call`, and the Python stack trace.
2. Noting the repeated blocks, we can isolate a single layer here. We can also see (from looking at the code/understanding how a transformer works) what parts are attention and what parts are MLPs.
3. By clicking on an XLA op, we can view where in the code it comes from (useful for understanding the trace) and see links to the Graph viewer.

Tip: you can navigate the Trace Viewer using "video game" style controls, with A/D panning left and right, and W/S zooming in and out. These controls make navigating a lot easier.

9.2.2 How to Read an XLA Op

HLO isn't actually very hard to read, and it's very helpful for understanding what a given part of the trace above corresponds to. Here's an example op called fusion.3.

```
%fusion.3 = bf16[32,32,4096]{2,1,0:T(8,128)(2,1)S(1)} fusion(bf16[32,32,8192]{2,1,0:T(8,128)
    (2,1)S(1)} %fusion.32), kind=kCustom, calls=%all-reduce-scatter.3
```

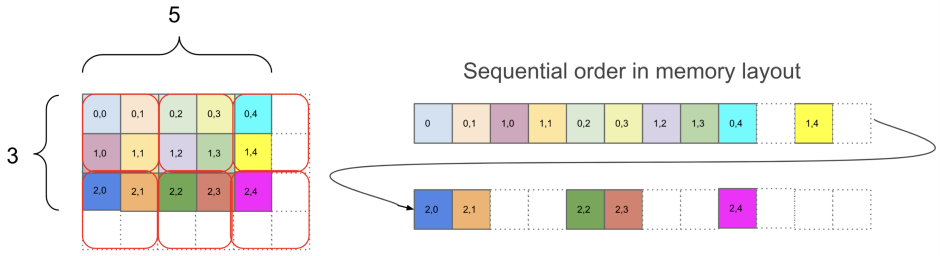
Let's break this down into its pieces.

- **Op Name:** `fusion.3`
  - A dot or fusion op is a set of operations containing at most 1 matrix multiplication and possibly a bunch of related pointwise VPU-ops.
- **Shape/layout:** `bf16[32,32,4096]`
  - This is the output shape of the op. We can see the dtype is bf16 (2 bytes per parameter) and `[32,32,4096]` is the shape.
- **Layout:** `{2,1,0:T(8,128)(2,1)}`
  - `{2,1,0:T(8,128)(2,1)}` tells us the order of the axes in memory (column major, row major, etc.) and the array padding. More below.
- **Memory location:** `S(1)`
  - `S(1)` tells us this array lives in VMEM. `S(0)` (sometimes omitted) is HBM. `S(2)` and `S(3)` are other memory spaces.
- **Arguments:** `bf16[32,32,8192]{2,1,0:T(8,128)(2,1)S(1)} %fusion.32`
  - This op has one input, a bf16 array called `fusion.32` with a particular shape. This tells us what function feeds into this one.

Let's try to understand this notation a little more. Let's take this as a simple example:

`f32[3,5]{1,0:T(2,2)}`

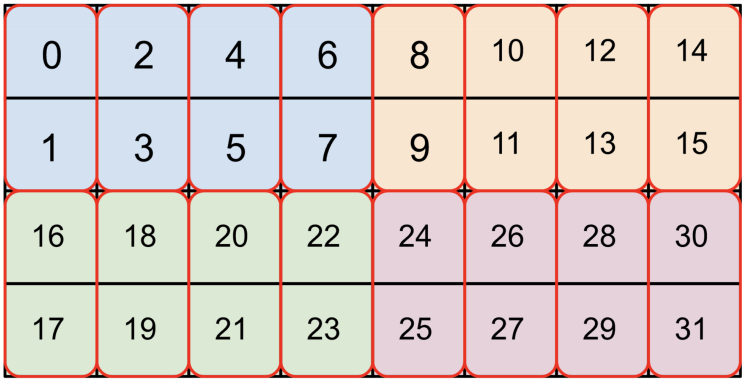
which again tells us that this Op returns a float32 array of shape `[3, 5]` with a particular tiling `{1,0:T(2,2)}`. While tilings don't matter *too* much, briefly, tilings tell us how an N-dimensional array is laid out sequentially in memory. Here's a diagram showing how this array is laid out:



F32[3,5] with tile size of 2x2

Within `,0:T(2,2)1,0:T(2,2)`, the `1,0` part tells us the ordering of array dimensions in physical memory, from most minor to most major. You can read this part from right to left and pick out the corresponding dimensions in `f32[3,5]` to figure out the physical layout of the array. In this example, the physical layout is `[3,5]`, identical to the logical shape.

After that, `T(2,2)` tells us that the array is tiled in chunks of `(2,2)` where within each chunk, the array has rows first (**row-major**), then columns, i.e. `(0,0)` is followed by `(0,1)`, then `(1,0)` and `(1,1)`. Because of the `T(2,2)` tiling, the array is padded to `[4,6]`, expanding its memory usage by about 1.6x. For the big bf16 array given above, `,1,0:T(8,128)(2,1)S(1)bf16[32,32,8192]2`, we do `T(8,128)(2,1)` which tells us the array has two levels of tiling, an outer `(8, 128)` tiling and an inner `(2,1)` tiling within that unit (used for bf16 so our loads are always multiples of 4 bytes). For example, here's `,0,T(2,4)(2,1)bf16[4,8]1` (colors are (2,4) tiles, red boxes are (2,1) tiles):

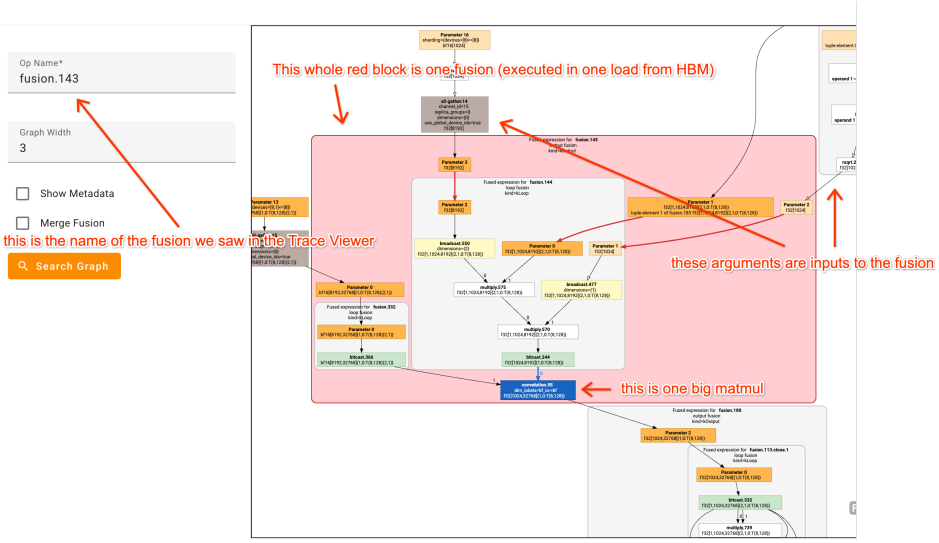


Tiling can affect how efficiently chunks of tensors can be loaded into VMEM and XLA will sometimes introduce copies that "retile" or "re-layout" a tensor inside a program, sometimes at non-trivial overhead.[4]

[4]JAX provides an experimental feature to work around this issue, by allowing XLA to compute its "preferred" layout for inputs to a program. When you "just-in-time" compile a program with `jax.jit`, you typically pass in "mock" inputs that tell JAX what shape and dtype to expect. These typically also carry tiling information that may not be optimal. Instead, you can specify the input layouts as AUTO, and `jax.jit` will return a layout that the jitted program prefers. You can then explicitly load the tensor in that layout to avoid inducing copies within the program.

9.2.3 Graph Viewer

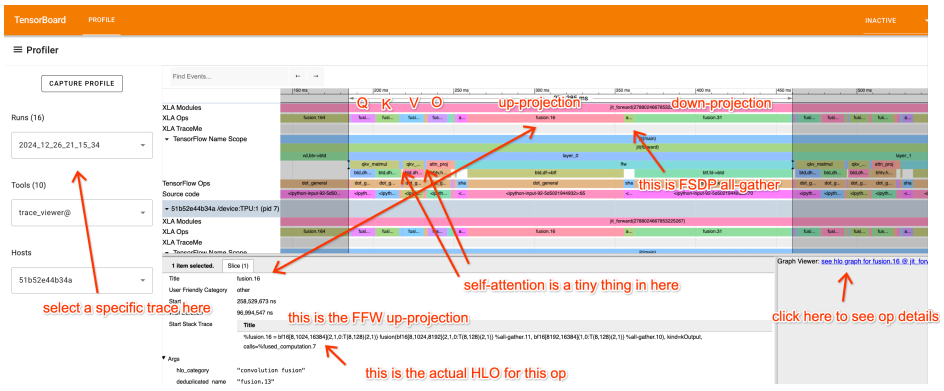
While some of the fusions above can seem complicated, the XLA Graph Viewer makes them easier to parse. For example here's the view of a fairly complicated fusion:



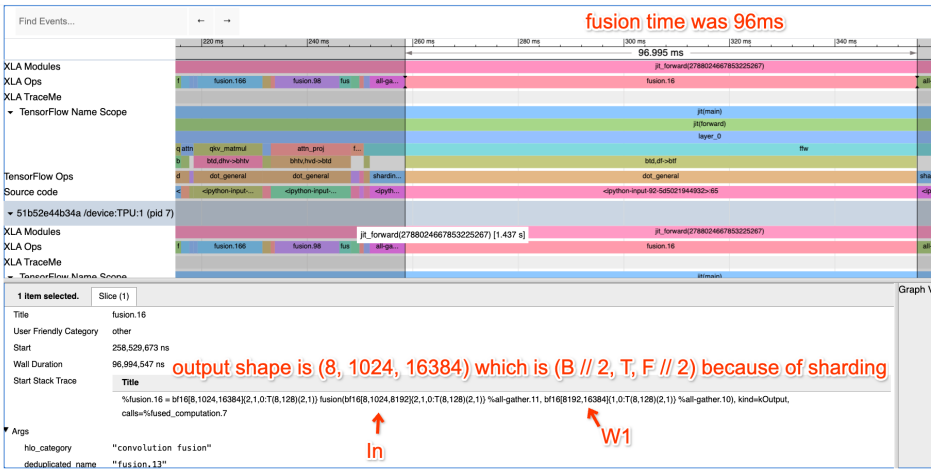
It's really helpful to stare at a bunch of HLO graphs and try to map HLO ops onto the code you're profiling. By hovering over a box you'll often see the line of code where the function was defined.

9.2.4 Looking at a Real(ish) Example Profile

This section covers an example profile (can be found in the online version of this book). I've gone to more effort than usual to annotate the trace with `jax.named_scope` calls so you can identify what's going on.



Take a look at the profile and try to really understand what each part is doing. Let's break it down a bit, starting with the FFW block:



Here we've zoomed into the FFW block. You'll see the up-projection Op is a fusion (matmul) with inputs `bf16[8, 1024, 8192]` and `bf16[8192, 16384]` and output `bf16[32, 1024, 16384]`. I know (because I wrote this code) that this is a local view of a 4-way DP, 2-way MP sharded matmul, so we're actually doing:

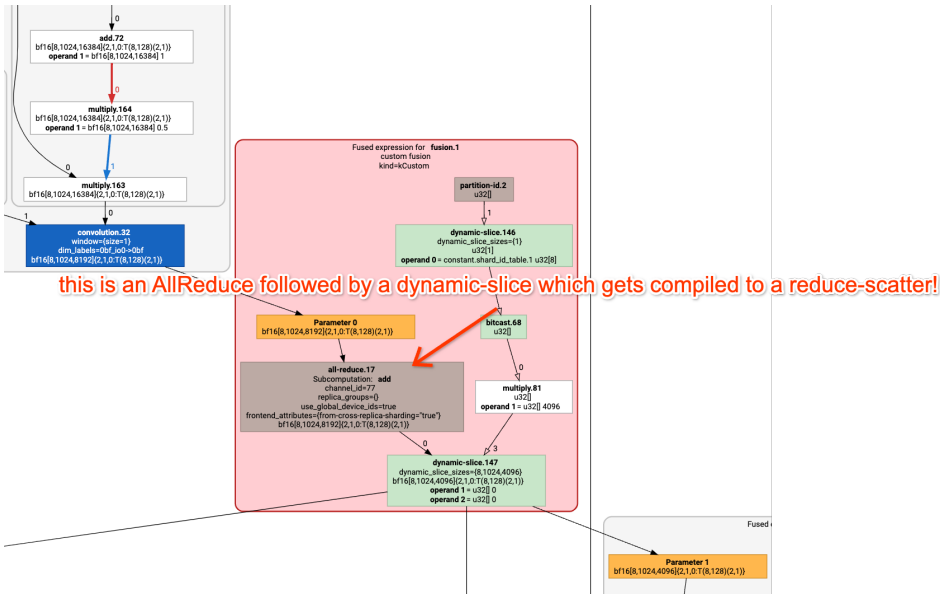
X: `bf16[32, 1024, 8192]` * $\mathbf{W}_{in}$: `bf16[8192, 32768]` -> **Tmp**: `bf16[32, 1024, 32768]`

How long do we expect this to take? First of all, our batch size per data parallel shard is `8 * 1024 = 8192`, so we should be solidly compute-bound. This is on 8 TPUv2 cores (freely available on Google Colab), so we expect it to take about `2 * 32 * 1024 * 8192 * 32768 / (23e12 * 8) = 95.6ms` which is pretty much exactly how long it takes (96ms). That's great! That means we're getting fantastic FLOPs utilization!

What about communication? You'll notice the little fusion hidden at the end of the second matmul. If we click on it, you'll see

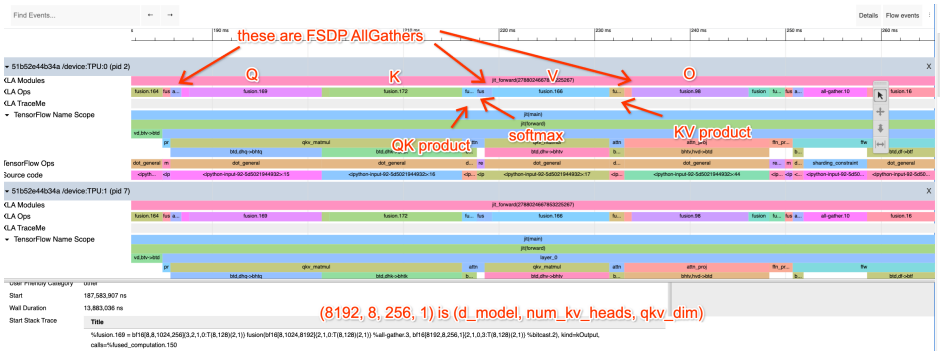
```
%fusion.1 = bf16[8,1024,4096]{2,1,0:T(8,128)(2,1)} fusion(bf16[8,1024,8192]{2,1,0:T(8,128)(2,1)
    } %fusion.31), kind=kCustom, calls=%all-reduce-scatter.1
```

which is basically a little ReduceScatter (here's the GraphViewer);



How long do we expect this to take? Well, we're doing a ReduceScatter on a TPUv2 4x2, which should require only one hop on `1.2e11` bidirectional bandwidth. The array has size `2*32*1024*8192` with the batch axis sharded 4 ways, so each shard is `2*8*1024*8192=134MB`. So this should take roughly 1.1ms. **How long does it actually take?** 1.13ms reported in the profile. So we're really close to the roofline!

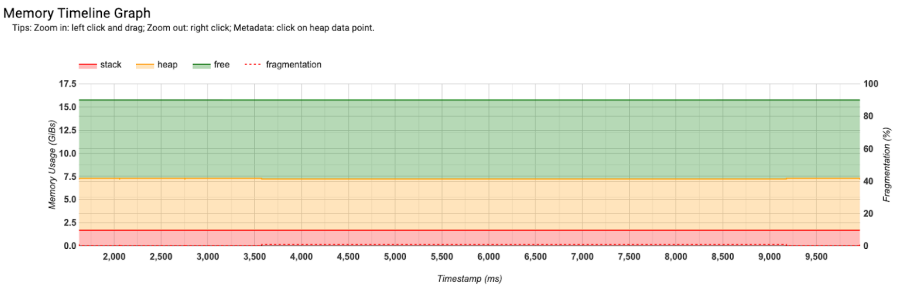
Let's look at attention too! Here's a profile of the attention component:



I've clicked on the Q projection op, which uses a matrix W_Q of shape [d_{model} = 8192, n_{heads} = 32, d_{qkv} = 256]. We're megatron sharding along the head dimension. Try to do the same exercise of calculating how long these should take.

9.2.5 Memory Profile

The Memory Profile makes it easy to see the program memory as a function of time. This is helpful for debugging OOMs. You can see here about 7.5GB allocated to model parameters and about 10GB free. So we can fit a lot more into memory.



9.3 Worked Problems

Please view the online version of this book for these questions. A computer is necessary to work on them.

10 Programming TPUs in JAX

How to use JAX to program TPUs efficiently! Much of this section is taken from `https://jax.readthedocs.io/en/latest/jep/14273-shard-map.html`

10.1 How Does Parallelism Work in JAX?

JAX supports two schools of thought for multi-device programming:

1. **Compiler, take the wheel!** Let the compiler automatically partition arrays and decide what communication to add to facilitate a given program. This lets you write a program on a single device and automatically run it on hundreds without changing anything.
2. **Just let me write what I mean, damnit!** While compilers are nice, they sometimes do the wrong thing and add communication you don't intend. Sometimes we want to be extremely explicit about what we're doing.

Correspondingly, JAX provides two APIs for each of these schools: **jit** (`jax.jit`) and **shard_map** (`jax.experimental.shard_map.shard_map`).

1. `jax.jit` lets you specify the sharding of the inputs and outputs to a program (via `in_shardings` and `out_shardings`) and infers the rest using the GSPMD[1] compiler. While it isn't perfect, it usually does a decent job at automatically scaling your program to any number of chips.
2. `jax.experimental.shard_map.shard_map` is the more explicit counterpart. You get a device-local view of the program and have to write any communication you want explicitly. Have a sharded array and want the whole thing on each device? Add a `jax.lax.all_gather`. Want to sum an array across your devices? Add a `jax.lax.psum` (an AllReduce). Programming is harder but far less likely to do something you don't want.

10.1.1 `jax.jit`: the automatic parallelism solution

jax.jit plays two roles inside JAX. As the name suggests, it "just-in-time" compiles a function from Python into bytecode (via XLA/HLO/LLO) so it runs faster. But if the input is sharded or the user specifies an `in_sharding` or `out_sharding`, it also lets XLA distribute the computation across multiple devices and add communication as needed. For example, here's how you could write a sharded matmul using `jax.jit`:

```
import jax
import jax.numpy as jnp
import jax.sharding as shd

P = shd.PartitionSpec

# Running on a an TPU v5e 2x2. This assigns names to the two physical axes of the hardware.
mesh = jax.make_mesh(axis_shapes=(2, 2), axis_names=('X', 'Y'))

# This tells JAX to use this mesh for all operations, so you can just specify the PartitionSpec
    P.
shd.set_mesh(mesh)

# We create a matrix W and input activations In sharded across our devices.
```

[1] `https://arxiv.org/abs/2105.04663`

```
In = jnp.zeros((8, 2048), dtype=jnp.bfloat16, out_sharding=P('X', 'Y'))
W = jnp.zeros((2048, 8192), dtype=jnp.bfloat16, out_sharding=P('Y', None))

def matmul_square(In, W):
  return jnp.einsum('bd,df->bf', jnp.square(In), W)

# We can explicitly compile the sharded matmul function here. This adds all the
# necessary comms (e.g. an AllReduce after the matmul).
jit_matmul = jax.jit(matmul_square, out_shardings=P('X', None)).lower(In, W).compile()

out = jit_matmul(In, W)
```

This will run automatically with any sharding and partition the computation across our devices. **But what's actually happening at the hardware level?**

1. First we create In and W sharded across our devices[2]. W is sharded 2 way along the contracting dimension, while In is sharded 4-ways (along both the contracting and output dimensions). This corresponds to a sharding $W[D_X, F]$ and $In[B_X, D_Y]$, aka a kind of model and data parallelism.
2. If we were running this locally (i.e. on one device), `matmul_square` would simply square the input and perform a simple matmul. But because we specify the `out_shardings` as `P('X', None)`, our output will be sharded along the batch but replicated across the model dimension and will require an AllReduce to compute.

Using our notation from previous sections, this will likely do something like

1. $Out[B_X, F] \{ U_Y \} = In[B_X, D_Y] *_D W[D_Y, F]$
2. $Out[B_X, F] = \text{AllReduce}(Out[B_X, F] \{ U_Y \})$

`jax.jit` will add this for us automatically! We can actually print the HLO with `jit_matmul.as_text()` and see the following HLO (abbreviated dramatically):

```
# This fusion is the actual matmul of the sharded inputs and matrix
%fusion = bf16[4,8192]{1,0:T(4,128)(2,1)S(1)} fusion(bf16[4,1024]{1,0:T(4,128)(2,1)} %param,
    bf16[8192,1024]{1,0:T(8,128)(2,1)S(1)} %copy-done)

# We reduce the partially summed results across devices
ROOT %AllReduce = bf16[4,8192]{1,0:T(4,128)(2,1)} AllReduce(bf16[4,8192]{1,0:T(4,128)(2,1)S(1)}
     %fusion)
```

We can see the matmul (the fusion) and the AllReduce above. Pay particular attention to the shapes. `bf16[4, 1024]` is a local view of the activations, since our `batch_size=8` is split across 2 devices and our `d_model=2048` is likewise split 2 ways.

This is pretty magical! No matter how complicated our program is, GSPMD and jit will attempt to find shardings for all the intermediate activations and add communication as needed. With that said, GSPMD has its flaws. It can make mistakes. Sometimes you'll look at a profile and notice something has gone wrong. A giant AllGather takes up 80% of the profile, where it doesn't need to. When this happens, we can try to correct the compiler by explicitly annotating intermediate tensors with `jax.lax.with_sharding_constraint`. For

[2]Notice how we did this. This is one way to create an array with a particular sharding (i.e. by adding the device argument to the creation function). Another one is to create an array normally with `jnp.array(...)` and then do e.g. `jax.device_put(..., P('x', 'y'))`. Yet another is to write a function which creates the array you want, and jit-compile it with `out_shardings` being what you want.

instance, with two matmuls I can force the intermediate activations to be sharded along the `y` dimension (not that this is a good idea) with the following:

```
import jax
import jax.numpy as jnp

def matmul(x, Win, Wout):
  hidden = jnp.einsum('bd,df->bf', x, Win)
  hidden = jax.lax.with_sharding_constraint(hidden, P('x', 'y'))
  return jnp.einsum('bf,df->bd', hidden, Wout)
```

This makes up like 60% of JAX parallel programming in the jit world, since it's our only way of intervening with the compiler. It's worth playing around with `with_sharding_constraint` in a Colab and getting a sense for how it works. When we write LLMs using `jax.jit`, 90% of what we do to control shardings is changing the input and output shardings (via `in_shardings` and `out_shardings`) and annotating intermediate tensors with `with_sharding_constraint` to ensure the correct comms are happening. For more `jax.jit` examples, this is a great doc to read[3].

10.1.2 `shard_map`: explicit parallelism control over a program

While GSPMD is the "compiler take the wheel" mode, jax shard_map puts everything in your hands. You specify the sharding of the inputs, like in `jax.jit`, but then you write all communication explicitly. Whereas `jax.jit` leaves you with a global cross-device view of the program, `shard_map` gives you a local per-device view.

Here's an example. Try to reason about what this function does:[4]

```
import jax
import jax.numpy as jnp
import jax.sharding as shd

from jax.experimental.shard_map import shard_map as shmap

P = shd.PartitionSpec
shd.set_mesh(jax.make_mesh(axis_shapes=(2, 4), axis_names=('x','y')))

x = jnp.arange(0, 512, dtype=jnp.int32, device=P(('x', 'y')))

# This function will operate on 1/8th of the array.
def slice_and_average(x):
  assert x.shape == (512 // 8,)
  return jax.lax.pmean(x[:4], axis_name=('x', 'y'))

out = shmap(slice_and_average, shd.get_abstract_mesh(), in_specs=P(('x', 'y')), out_specs=P(
    None,))(x)
assert out.shape == (4,)
```

What does this do? `slice_and_average` is run on each TPU with 1/8th of the array, from which we slice the first 4 elements and average them across the full mesh. This means we're effectively doing `mean(x[:4], x[64:68], x[128:132], ...)`. This is pretty cool, because that's not an easy operation to express in JAX otherwise.

[3] https://jax.readthedocs.io/en/latest/notebooks/Distributed_arrays_and_automatic_parallelization.html

[4] If you want to play with this yourself in a colab by emulating a mesh, you can do so using the following cell `import os; os.environ["XLA_FLAGS"] = '--xla_force_host_platform_device_count=8'`

Why do this instead of `jax.jit`? If we'd used `jax.jit`, `slice_and_average` would have seen a global view of the array (the full `[512,]` array). We'd have had to slice out this non-uniform slice and then perform an average which XLA would have had to interpret correctly. XLA might have added the wrong communication or gotten confused. Here we see the local view and write only the communication we need.

Example [Collective Matmul]: To take a more realistic example, say we to implement model parallelism where the activations are initially model sharded, i.e. $A[B_X, D_Y] * W[D, F_Y] \to Out[B_X, F_Y]$. Naively, we would do this by AllGathering A first followed by a local matrix multiplication:

1. $A[B_X, D] = \textbf{AllGather}_Y(A[B_X, D_Y])$
2. $Out[B_X, F_Y] = A[B_X, D] *_D W[D, F_Y]$

Sadly, this is bad because it doesn't allow us to overlap the communication with the computation. Overlapping them can be done with a "collective matmul", as described in Wang et al. 2023[5]. The algorithm is basically as follows:

- For each Y shard, perform a matmul of the local chunk of A with the local chunk of W, producing a result of shape [B / X, F / Y]. Simultaneously, permute A so you get the next chunk locally, perform the matmul, and sum the result.

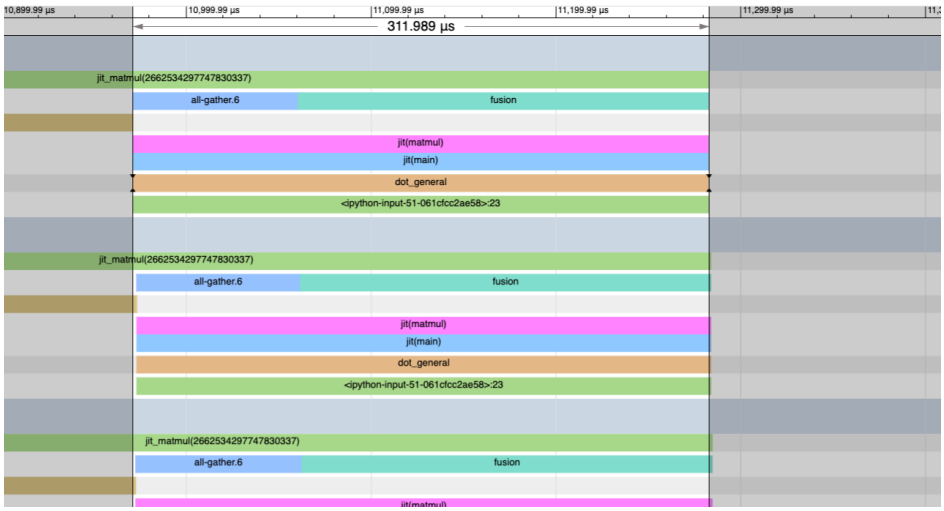
We can implement that quite easily with shard_map:

```
import functools

import jax
import jax.numpy as jnp
import jax.sharding as shd
import numpy as np

from jax.experimental.shard_map import shard_map

mesh = jax.make_mesh(axis_shapes=(2, 4), axis_names=('X', 'Y'))
def P(*args):
  return shd.NamedSharding(mesh, shd.PartitionSpec(*args))

B, D, F = 1024, 2048, 8192
A = jnp.arange(np.prod((B, D))).reshape((B, D))
W = jnp.arange(np.prod((D, F))).reshape((D, F))

A = jax.device_put(A, P('X', 'Y'))
W = jax.device_put(W, P(None, 'Y'))

@functools.partial(jax.jit, out_shardings=P('X', 'Y'))
def matmul(lhs, rhs):
  return lhs @ rhs

def collective_matmul_allgather_lhs_contracting(lhs, rhs):
  # lhs is the looped operand; rhs is the local operand
  axis_size = jax.lax.psum(1, axis_name='Y')  # axis_size = 4 for this example
  idx = jax.lax.axis_index('Y')

  chunk_size = lhs.shape[1]
  assert rhs.shape[0] % chunk_size == 0
```

[5] https://dl.acm.org/doi/pdf/10.1145/3567955.3567959

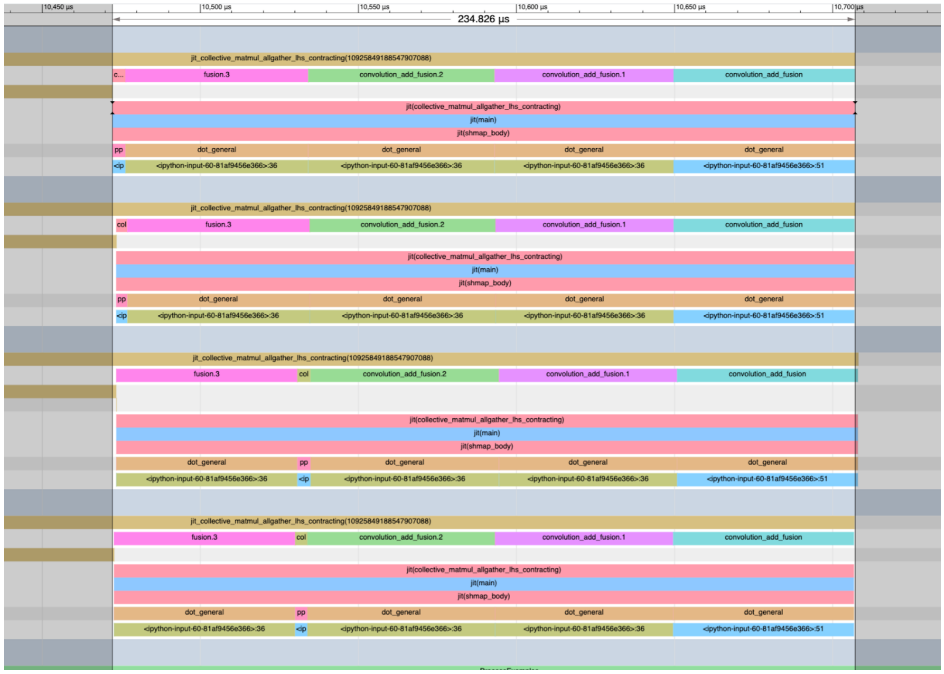
```
  def f(i, carrys):
    accum, lhs = carrys
    rhs_chunk = jax.lax.dynamic_slice_in_dim(rhs, (idx + i) % axis_size * chunk_size,
    chunk_size)
    # Matmul for a chunk
    update = lhs @ rhs_chunk
    # Circular shift to the left
    lhs = jax.lax.ppermute(
        lhs,
        axis_name='Y',
        perm=[(j, (j - 1) % axis_size) for j in range(axis_size)]
    )
    return accum + update, lhs

  accum = jnp.zeros((lhs.shape[0], rhs.shape[1]), dtype=lhs.dtype)
  accum, lhs = jax.lax.fori_loop(0, axis_size - 1, f, (accum, lhs), unroll=True)

  # Compute the last chunk after the final permute to leave lhs in the state we found it
  i = axis_size - 1
  rhs_chunk = jax.lax.dynamic_slice_in_dim(rhs, (idx + i) % axis_size * chunk_size, chunk_size)
  update = lhs @ rhs_chunk
  return accum + update

jit_sharded_f = jax.jit(shard_map(
  collective_matmul_allgather_lhs_contracting, mesh,
  in_specs=(shd.PartitionSpec('X', 'Y'), shd.PartitionSpec(None, 'Y')), out_specs=shd.
    PartitionSpec('X', 'Y')))

shmapped_out = jit_sharded_f(A, W)
expected_out = matmul(A, W)

np.testing.assert_array_equal(shmapped_out, expected_out)
```

This is pretty neat! We can benchmark this and see that it's also a lot faster! Here's[6] the profile with the default jit matmul which takes 311us with a big blocking AllGather at the beginning:



[6] https://imgur.com/a/e9I6SrM

And here's[7] the version above that takes 244 us. You can see the profile doesn't have the AllGather. It's all useful work! Our FLOPs utilization is also a lot higher.



It's also worth noting that the matmul time with no sharding on the contracting dimension is 224us[8], so we're remarkably close to the unsharded baseline here. This is a good example of the kind of performance engineering you might end up doing to improve TPU utilization. For more `shard_map` examples, this note is great[9].

Now here are a couple of useful worked problems to try and implement using `jax.jit` or `shard_map`!

10.2 Worked Problems

Here are some random JAX-related problems. I'll add some more later. For all of these, you'll need some number of TPUs in a Colab. You can use a public Colab with TPUv2-8. From now on, we'll assume you have N devices available.

Exercise 10.1

As an exercise, let's start by implementing an AllReduce collective matmul, i.e. $A[B_X, D_Y] *_D W[D_Y, F] \rightarrow Out[B_X, F]$. Note that the output isn't replicated. The naive algorithm is discussed above, basically just a local matmul followed by an AllReduce. Try to make a comms overlapped "collective" version of this operation. *Hint: tile over the output dimension and feel free to use `jax.lax.psum` (aka AllReduce). Note: due to the way XLA handles this, it may not actually be faster than the baseline.*

[7] https://imgur.com/a/21iy0Sv

[8] https://imgur.com/a/i3gNKfq

[9] https://jax.readthedocs.io/en/latest/notebooks/shard_map.html#example-1-all-gather-on-one-side

Exercise 10.2

The complement to the AllReduce collective matmul above is a ReduceScatter collective matmul, as in Tmp[B_X, F_Y] *$_F$ W2[F_Y, D] -> Out[B_X, D_Y]. This occurs in the down-projection matrix in a Transformer. Implement a collective, overlapped version of this in JAX. Be careful about passing only the minimal amount of data you need. *Hint: try permuting the result as you accumulate it.*

Exercise 10.3

Put these two together into an end-to-end Transformer block that performs In[B_X, D_Y] *$_D$ W_{in}[D, F_Y] *$_F$ W_{out}[F_Y, D] -> Out[B_X, D_Y] with overlapped communication.[10] How much faster is this than a `jax.jit` implementation?

Exercise 10.4

All of the collective matmuls implemented above are unidirectional: they only permute in one direction. Rewrite the collective AllReduce matmul and the collective ReduceScatter matmuls to use bidirectional communication. How much faster are these?

[10] As before, we can't do $W_{\text{in}} \cdot W_{\text{out}}$ first because of a non-linearity we've omitted here.

11 Conclusions

Thank you for reading this set of essays and congratulations on making it all the way to the end. Before we conclude, a few acknowledgments:

11.1 Acknowledgments

This document represents a significant collective investment from many people at Google DeepMind, who we'd like to briefly acknowledge!

- James Bradbury, Reiner Pope, and Blake Hechtman originally derived many of the ideas in this manuscript, and were early to understanding the systems view of the Transformer.
- Sholto Douglas wrote the first version of this doc and is responsible for kicking off the project. He is more than anyone responsible for the overall narrative of this doc.
- Jacob Austin led the work of transforming this first version from rough notes into a more polished and comprehensive artifact. He did much of the work of editing, formatting, and releasing this document, and coordinated contributions from other authors.
- Most of the figures and animations were made by Anselm Levskaya and Charlie Chen.
- Charlie Chen wrote the inference section and drew many of the inference figures.
- Roy Frostig helped with publication, editing, and many other steps of the journey.

We'd also like to thank many others gave critical feedback throughout the process, in particular Zak Stone, Nikhil Sethi, Caitlin Stanton, Alex Dimitriev, Sridhar Lakshmanamurthy, Albert Magyar, Diwakar Gupta, Jeff Dean, Corry Wang, Matt Johnson, Peter Hawkins, and many others. Thanks to Ruiqi Gao for help with the HTML formatting.

Thank you all!

11.2 Further Reading

There is a bunch of related writing, including the following:

- **TPU Deep Dive**: A wonderful in-depth look at the TPU architecture in the spirit of this book.
- **Making Deep Learning Go Brrrr From First Principles**: a more GPU and PyTorch-focused tutorial on LLM rooflines and performance engineering.
- **Writing TPU Kernels with Pallas**: increasingly, TPU programming involves writing custom kernels in Pallas. This series discusses how to write kernels and many lower level TPU details that aren't mentioned here.
- **How to Optimize a CUDA Matmul Kernel for cuBLAS-like Performance: a Worklog**: while GPU and CUDA specific, this is an excellent blog post showing how to optimize a matmul kernel in CUDA. This might be a good deep dive into how TPUs and GPUs are different.
- **Distributed arrays and automatic parallelization**: this is a really nice guide to parallelism APIs in JAX and is a good way to learn how to actually implement some of the ideas we've discussed here.

- **Rafi Witten's High Performance LLMs 2024 Class**: our former colleague Rafi gave a great course on TPU performance engineering and the slides are all on GitHub. This covers a bunch of things in more depth than we do here.
- **[2211.05102] Efficiently Scaling Transformer Inference**: a detailed paper on the mathematics of Transformer inference. This is the inspiration for a lot of this document.
- **Huggingface Ultra-Scale Playbook**: something of a GPU analog to this book, this talks more at depth about how PyTorch implements parallelism techniques and memory-saving techniques during training.
- **Transformer Inference Arithmetic**: a blog with many of the same ideas as this book and some excellent illustrations.
- **Stanford CS336 Slides and Videos**: a fantastic Stanford course covering many details of LLM training and serving, with some useful exercises. Assignments 1 and 2 are particularly relevant.

There remains a lot of room for comprehensive writing in this area, so we hope this manuscript encourages more of it! We also believe that this is a fruitful area to study and research. In many cases, it can be done even without having many hardware accelerators on hand.

11.3 Feedback

Please leave comments or questions so that we can improve this further. You can reach our corresponding author, Jacob Austin, at jaaustin [at] google [dot] com, or suggest edits by posting issues, pull requests, or discussions on GitHub.

12 Solutions

Solution 1.1

1. Because we're storing our parameters in int8, we have 1 byte per parameter, so we have $BD + DF$ bytes loaded from HBM and BF written back.
2. This is the same as in bfloat16, but in theory int8 OPs/s should be faster. So this is still $2BDF$ FLOPs.
3. Arithmetic intensity is $2BDF/(BD + DF + BF)$. If we make the same assumption as above about $B \ll D$ and $B \ll F$, we get an arithmetic intensity of $2B$, meaning our rule becomes $B > \frac{\text{HBM int8 arithmetic intensity}}{2}$. Using the numbers given, this int8 intensity is $3.94e14/8.1e11 = 486$, so the rule is $B > 486/2 = 243$. Note that this is basically unchanged!
4. $T_{\text{math}} = 2BDF/3.94e14$ and $T_{\text{comms}} = (BD + DF + BF)/8.1e11$, so a reasonable lower bound is $\max(T_{\text{math}}, T_{\text{comms}})$ and an upper bound is $T_{\text{math}} + T_{\text{comms}}$.

Solution 1.2

Again assuming B is small, we have 2BDF bfloat16 FLOPs but only DF weights (instead of 2DF in bfloat16). This means we become compute-bound when $2B > 240$ or $B > 120$. This is a lot lower, meaning if we can do int8 weight quantization (which is fairly easy to do) but still do bfloat16 FLOPs, we get a meaningful win in efficiency (although int8 OPs would be better).

Solution 1.4

Let's start by looking at the total FLOPs and comms.

1. Total FLOPs: the FLOPs is basically the same, since we're doing the same number of $BD \times DF$ matmuls (this is discussed more in section 4). So this is just $2BDF$.
2. Total comms: we have a lot more comms here: $BD + BDF + BF$.
3. Therefore, our arithmetic intensity is now actually $2BDF/(BD + BDF + BF)$. Since BDF dominates the denominator, this is roughly 2. So instead of it depending on the batch size, this is essentially constant. This is bad because it means we'll basically always be comms bound no matter what.

Solution 1.5

From the spec sheet, we see that the reported bfloat16 FLOPs value is `1.979e15` FLOPs/s with an asterisk noting "with sparsity". The true value is half this without sparsity, meaning close to `1e15` FLOPs/s. The memory bandwidth is 3.35TB/s, or `3.35e12` bytes / second. Thus B_{crit} is `1e15 / 3.35e12 = 298`, rather similar to the TPU.

Solution 2.1

We're loading `sizeof(bf16) * 200e9 = 400e9` bytes on 32 chips, meaning 12.5e9 bytes / chip, each with an HBM bandwidth of 1.23e12. So the load takes around 10ms.

That's pretty cool, because *that's a reasonable lower bound on the latency of sampling* from the model. Each sampling step needs to load all parameters from HBM, so it cannot take less than 10 ms. In practice, at small batch sizes, this is close to being achievable.

Solution 2.2

For TPU v5e, each pod is `16x16` and each host is a `4x2` slice, so we have `16*16 / 8 = 32` hosts. For TPU v5e, each TPU has only one core, so we have 256 TensorCores. The total FLOPs/s is `16*16*2e14 = 5.1e16` in bfloat16. Each chip has 16GB of HBM, so that's `256 * 16 = 4TB` of memory.
For a full TPU v5p pod, we have `16x20x28` chips and each host is `2x2x1`, so we have `16*20*28 / 2*2 = 2,240` hosts. For TPU v5p, each TPU has two TensorCores, so we have `8960 * 2 = 17,920` cores. The total FLOPs/s is `8960 * 4.5e14 = 4e18` in bfloat16. Each chip has 96GB of HBM, so that's `8960 * 96 = 860TB` of memory.

Solution 2.3

We have to perform $2BDF$ floating point operations, and each chip can perform `9.2e14` floating point operations per second. This then requires $2BDF/9.2e14$ seconds to perform. We have to load $2DF + 2BD$ bytes from DRAM, and write $2BF$ bytes back to it. We are bottlenecked by PCIe transfer speeds, so we need $2 \cdot (BD + DF + BF)/1.5e10$ seconds to transfer data to and from the TPU. Since we want computation to take longer than weight loading, assuming we can overlap all weight loading with computation, we want $2BDF/9.2e14 > 2 \cdot (BD + DF + BF)/1.5e10$. We can simplify this using our assumptions that $B \ll D$, and $F = 4D$, to get

$$\frac{8BD^2}{9.2e14} > \frac{8D^2}{1.5e10}$$

or

$$B > \frac{9.2e14}{1.5e10} \simeq 61,000$$

Solution 2.4

1. The number of floating point operations we need to perform is $2 \cdot 4096 \cdot 16384 \cdot B = 1.3e8 \cdot B$. So $T_{\text{math}} = (1.3e8 \cdot B)/3.94e14$ seconds. We need to load $16384 \cdot 4096 + 4096 \cdot B$ bytes from HBM to VMEM, and write back $16384 \cdot B$ bytes from VMEM to HBM. This means $T_{\text{comms}} = (6.7e7 + 2e4 \cdot B)/8.1e11$ seconds. Assuming as much overlap of communication and computation as possible, the whole multiplication will take approximately

$$\max\{T_{\text{math}}, T_{\text{comms}}\} = \max\left\{\frac{6.7e7 + 2e4 \cdot B}{8.1e11}, \frac{1.3e8 \cdot B}{3.94e14}\right\}$$

   We'll be FLOPs-bound when $\frac{6.7e7+2e4 \cdot B}{8.1e11} < \frac{1.3e8 \cdot B}{3.94e14}$, or equivalently, $B > 271$. This is slightly larger than the 240 number we derive below because we factor in the full impact of D and F.

2. If instead we are loading from VMEM, let's consider VMEM bandwidth to the MXU as 22 times the HBM $\leftrightarrow$ VMEM bandwidth. This turns our data loading denominator from 8.1e11 to 1.78e13, and we get $B > 11$. Note that in practice, we cannot dedicate all of our VMEM bandwidth to loading W, so in practice it will be closer to 20.

Solution 2.5

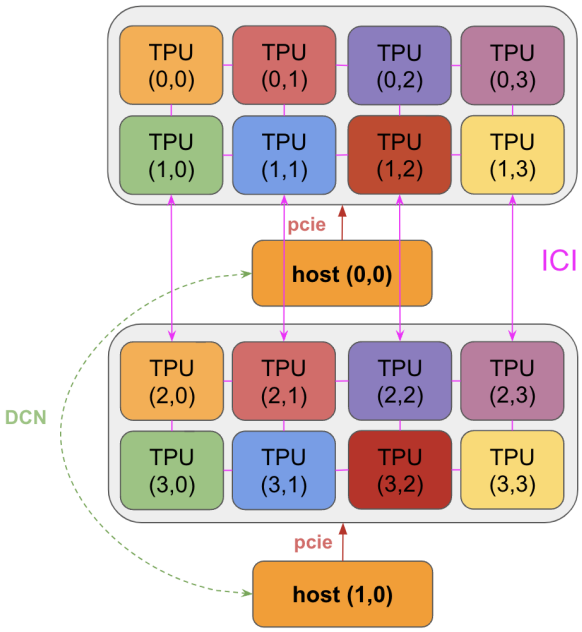
In a TPUv5e we have 2D connectivity. Because we have only a `4x4` slice (with no axes of size 16), we have no wraparound connections. Thus there are two ports from which our target chip can receive data, and likewise two ports from which our source chip can send data. The amount of data we have to transfer is `2 * 8 * 128 * 8192 = 1.7e7` bytes. We can transfer from both ports simultaneously (i.e. send half the array right and half down), so we get `2 * 4.5e10 = 9e10` bytes transferred per second, which means it'll take about `1.7e7 / 9e10 = 188us` to transfer the whole array through (assuming we're bandwidth bound). In a `4x4` slice, we have six hops between chips $(0, 0)$ and $(3, 3)$, since there are no wraparound links for axes with fewer than 16 chips. Since the latency of each hop is about $1\mu s$, the first byte will arrive in about`6us` and the total transfer will take `188us`.

Solution 2.6

Let's start by outlining the operations we have to perform. Our array is about 16GB. From the table above, a TPU v5e host has a 4x2 topology, so a 4x4 has 2 hosts, Thus, since our array is evenly sharded, each host effectively contains a chunk of 1/2 of the array, or 8GB. We need to copy these chunks all to `TPU{0,0}`, which gives us two options:

1. We can copy over DCN and then load the entire unsharded array over PCIe into HBM.
2. We can load our sharded arrays onto their corresponding TPUs, then perform a gather over ICI, then perform the matmul on `TPU{0,0}`.

It should be clear that option (2) is better. DCN is slow compared to ICI and we'd much prefer to load a big array over many PCIe links rather than just a few (the 8 on host 0). Here's a diagram of part of the system. As described above, note that TPUs are connected to their neighbors by ICI (even across hosts), all TPUs are connected to their host CPU (via PCIe), and hosts are connected by DCN.



Now let's work through how long each piece will take:

1. **PCIe load:** we're loading chunks of 16GB / 2 = 8GB over 8 PCIe links, each of which has `1.5e10` bytes/second bandwidth. Thus this will take about 66ms.

2. **ICI copy:** each TPU now has 16GB / 16 = 1GB of our array. Our ICI bandwidth is `10e10` bytes/second per link *bidirectional*, and you'll notice from the above diagram that only 2 of the 4 ICI links on the TPU v5e are in use in this topology. Since `TPU{0,0}` needs to receive a total of 15GB along 2 axes at `4.5e10` bytes/s/link, we can lower bound the time by `15e9 / (4.5e10 * 2) = 167ms`. In practice this probably isn't achievable because the load is very uneven, but it's probably within a factor of 2. As you'll see in Section 2, performing a full AllGather would also take roughly `16e9 / (4.5e10 * 2)`, so this is close to optimal.

3. **HBM → MXU load:** to perform our final matmul, we need to load these 16e9 bytes plus the `bf16[8, 128 * 1024]` array (another 2MB, so negligible) over HBM bandwidth into the MXU, which will take `16e9 / 8.1e11 = 19ms`.

4. **FLOPs:** we're performing a total of $2 \cdot 8 \cdot 128 \cdot 1024 \cdot 128 \cdot 1024 = 2.7e11$ FLOPs and since we can perform `1.97e14` bf16 FLOPs/s, we get 1.3ms.

An upper bound for the total time is the sum of all of these times, but since the TPU can typically overlap these operations, we can think of this as a pipelining problem that's bottlenecked by the slowest piece. Assuming that's true, then the answer is about 150-200ms.

Solution 3.1

Our array is only sharded along X, which has size 4, so effectively each shard has size $[I/4, J, K, ...] = \text{sizeof}(A)/4$. Since our array is replicated across Y and Z, the total size is $Y \cdot Z \cdot \text{sizeof}(A)$, so the ratio of total size to single chip size is $Y \cdot Z \cdot \text{sizeof}(A)/\text{sizeof}(A) = 16$.

Solution 3.2

We have a wraparound link on all axes because we have a full `4x4x4` cube, so we have 9e10 bidirectional bandwidth to work with.

1. Because we're just gathering over one axis and the other is sharded, we're effectively gathering $2BD/Y$ bytes over 1 axis. Since our ICI bandwidth for TPU v4p is 9e10 bytes/second bidirectional, this will take $2BD/(9e10 \cdot Y) = 2 \cdot 1024 \cdot 4096/(9e10 \cdot 4) = 23\mu s$.

2. We have twice the bandwidth as before but we're AllGathering the full array, so `T = 2BD / (2 * W) = 2*1024*4096 / (2 * 9e10) = 46us`. This is far from the latency bound of 4us (1us per hop), so we're fine.

3. The cost of an AllReduce is twice that of an AllGather. Each shard has size $2BD/(X * Y)$, so the cost is about $4BD/(X * Y * W)$, or roughly `4 * 1024 * 4096 / (16 * 9e10) = 11.6us`.

Solution 3.3

Our array in bfloat16 uses only 256 bytes total, and only 64 per device. Since we have an axis of size 4 on a TPU v4p, we have a wraparound link, so we can send the array in both directions. With `4.5e10` of unidirectional bandwidth, each hop would take roughly `64 / 4.5e10 ~ 0`, so we're definitely latency bound. Counting the number of hops, we can do the full gather in only 2 hops, so roughly 2us a good estimate.

Solution 3.4

Let's start with our baseline (*Strategy 1*). As we've shown, the cost of the AllGather is $2DF/W_{\text{ici}}$. Once we have the fully replicated arrays, the total compute time is $2BDF/C$ (where C is our accelerator FLOPs/s, since each TPU does the same FLOPs). So we have

$$T_{\text{total (Strategy 1)}} = \max\left(\frac{2BDF}{C}, \frac{2DF}{W_{\text{ici}}}\right)$$

By comparison, the new strategy (Strategy 2) does an AllReduce over $2BF$ bytes, which has cost $4BF/W_{\text{ici}}$ but does $1/X$ fewer FLOPs (since the computation is sharded). This means we do $2 \cdot B \cdot D \cdot F/X$ FLOPs and the resulting AllReduce communicates $2 \cdot 2 \cdot B \cdot F$ bytes in bfloat16. Thus, our total time for *Strategy 2* (no AllGather, just an AllReduce later on) is roughly

$$T_{\text{total (Strategy 2)}} = \max\left(\frac{2BDF}{X \cdot C}, \frac{4BF}{W_{\text{ici}}}\right)$$

The question is: *which of these is bigger?* Strategy (2) is compute bound when $D/(X \cdot C) > 2/W_{\text{ici}}$, or when $D/2X > C/W_{\text{ici}} \approx 2550 \rightarrow X < D/(2 * 2550)$. We might reasonably expect $D \approx 8k$, so this would mean roughly $X < 2$ which is unlikely – hence we're basically always comms bound with Strategy 2. With the baseline (Strategy 1), we're comms bound when $B < C/W_{\text{ici}} = 2550$ which is often but not always true.

So if $B < 2550$, we're comms-bound in both cases and we have

$$T_{\text{comms for Strategy 2}} < T_{\text{comms for Strategy 1}} \Leftrightarrow \frac{4BF}{W_{\text{ici}}} < \frac{2DF}{W_{\text{ici}}}$$

which is true when $D > 2B$ where $2B < 5100$. This is often true, so Strategy 2 can sometimes be better if our batch is small. When our batch is large ($B > 2550$), we have

$$T_{\text{comms for Strategy 2}} < T_{\text{math for Strategy 1}} \Leftrightarrow \frac{4BF}{W_{\text{ici}}} < \frac{2BDF}{C}$$

This is true when $2/W_{\text{ici}} < D/C$, or when $D > 2 * 2550 = 5100$, which is usually true for large models. So this alternative strategy is typically better for large models, unless D is small.

Why don't we always do this? Well, in practice we may do this sometimes, but it's typically rare to have the contracting dimension of one of the inputs to a matmul sharded along a axis that the other input isn't sharded over. For instance, if we're doing FSDP (explained in Section 5), we'll shard our parameters over the data dimension but our activations will *also be sharded along data*. So in this sense this doesn't show up much.

Solution 3.9

1. First compute the outer products, storing the result in $O[N, K] : o_{kj} = \sum_i a_{ki} b_{ij}$. Note that the repeated index is not the one being contracted, as we are doing an outer product. Here the sum ranges across the set of i values stored on the particular device we are using. So, for example, if we have a contracting axis of size 16, and 4 devices, then on device 0, i would range from {0, 1, 2, 3}; on device 1, i would range from {4, 5, 6, 7}; on device 2, i would range from {8, 9, 10, 11}; and on device 3, i would range from {12, 13, 14, 15}. Then AllReduce the partial-sums of $O[N, K]$ which live on each device, to form the full $O[N, K]$.

2. Instead of doing an AllReduce in step 2, we could get away with a cheaper ReduceScatter, along either axis: $[N, K]\{U_X\} \rightarrow [N_X, K]$ or $[N, K]\{U_X\} \rightarrow [N, K_X]$.

3. As described in the main text above, the cost of doing an AllGather (when we are throughput-bound) is the same as that of a ReduceScatter; it is simply given by the size of the full matrix we are processing. So in the gather-then-matmul algorithm, this scales as NM (since we are AllGather-ing A); in the matmul-then-reduce-scatter algorithm, this scales as NK (since we are reduce-scattering O). So the communication cost ratio of the two algorithms is M/K.

Solution 3.10

1. The process is simple: in each step of the algorithm, each device will send a single-shard "strip" of the matrix (totalling $\frac{N}{D} \times N$ elements in size) to its nearest neighbor. This occurs $D-1$ times, since each shard needs to be communicated to all of the devices except the one it starts out on. So in total, $\frac{N^2(D-1)}{D}$ scalars are transferred by each device, i.e. flow across a single ICI link. The final answer is $N^2(1-\frac{1}{D})$, or simply N^2 when $D >> 1$.

2. The key difference between an AllToAll and an AllGather, from the perspective of communications, is that in an AllToAll, the entirety of the shard that lives on a particular device does not need to be communicated to every other device. Imagine the shard stored on a particular device (call it device 0) is $[A, B, C, D]$ (here A,B,C,D are matrices and we are imagining a ring with 4 devices for illustration). Now the matrix A does not need to be communicated anywhere, the matrix B needs to end up on device 1; matrix C ends up on device 2; and matrix D ends up on device 3. So in the first step of the algorithm, we send B, C, and D to device 1; in the next step, device 1 sends C and D onwards to device 2; in the final step, device 2 sends just D on to device 3. The total number of parameters transferred in this case is (size of A/B/C/D) $*$ $(3+2+1)$. The size of A/B/C/D is (in the general case now) $\frac{N^2}{D^2}$, and again in the general case the $(3+2+1)$ term becomes $((D-1)+(D-2)+...+1)$, or $\frac{(D)(D-1)}{2}$. So the total number of bytes transferred across a single ICI link is $\frac{N^2(D-1)}{D \times 2}$.

3. The factor is simply $\frac{1}{2}$, i.e. an AllToAll is half as costly as an all-gather/ReduceScatter on a unidirectional ring topology. Looking over the derivations above, this ultimately came from the fact that in the all-gather case, we are transferring the same sized block each of $(D-1)$ times, i.e. we're doing the sum tiny block size $*$ $(D+D+D+...+D)$, whereas in the AllToAll case, we're doing the sum tiny block size $*$ $(D+D-1+D-2+...+1)$. The factor of two thus essentially comes from the fact that $1+2+...+n = n(n+1)/2$.

4. The total number of scalars that any one link has to carry now reduces by a factor of 2, since in a bidirectional ring, each "sharded strip" can be sent two ways simultaneously.

5. In this case, we win a factor of 4 compared to the unidirectional case. This is easiest to see by considering the fate of each of the size-(N2/D2) blocks in a single sharded strip, say the one which originates on device 0. Instead of (as in the unidirectional case) sending one of these blocks a distance of D-1, another block a distance D - 2, etc. all the way to 1, we now divide the strip into blocks which move right or left, moving a maximum distance of ceil(D/2). So the corresponding sum now becomes $D/2 + D/2 - 1 + D/2 - 2 + ... = D/2 \cdot (D/2+1)/2$, or $D^2/8$ in the limit of large D. Compare this to $D^2/2$ in the unidirectional case, and we see that we've won a factor of 4.

6. In a unidirectional ring, we saw that the AllToAll time was already twice as fast as the all-gather time; this comes from the fact that we don't need to send our full strip to every single device. Then, when

we added bidirectionality, we saw that it was a 4x win for AllToAll, and only a 2x win for all-gathers. Putting these ratios together, we get our sought after factor of 4.

Solution 4.1

1. The total parameters is roughly $L \cdot (3DF + 4DNH + D) + 2DV$. For the given numbers, this is $64 \cdot (3 \cdot 4e3 \cdot 16e3 + 4 \cdot 4e3 \cdot 4e3 + 4e3) + 2 \cdot 4e3 \cdot 32e3 = 16e9$, or 16B parameters.
2. The ratio of attention parameters to total parameters in general is $4DNH/(4DNH + 3DF) = 4D^2/(4D^2 + 12D^2) = 1/4$. This gives us roughly 1/4 of parameters are used in attention.
3. Per token, our KV caches are $2 \cdot L \cdot N \cdot H = 2 \cdot 64 \cdot 4096$ in int8, which is `512kB / token`.

Solution 4.2

The total "theoretical" FLOPs of the operation is $2 \cdot B \cdot D \cdot F$. However, because the computation isn't sharded across the Z dimension, we're actually doing Z extra FLOPs, meaning $2 \cdot B \cdot D \cdot F \cdot Z$ total FLOPs. Since the computation is sharded across the other dimensions, the total per-device is roughly $2 \cdot B \cdot D \cdot F/(X \cdot Y)$.

Solution 4.3

Following the rule above, we have I and J as contracting dimensions and K, L, M, N, and O as non-contracting dimensions. We have no "batching dimensions", so this is just $2 \cdot I \cdot J \cdot K \cdot L \cdot M \cdot N \cdot O$, the sum of all the axes. If we had a shared axis, it would only be counted once.

Solution 4.4

Self-attention requires loading the Q, K, and V activations, then computing $\text{softmax}(Q \cdot K) \cdot V$, then writing the result back to HBM. This will be done with Flash Attention so there are some caveats to this math, but basically in bf16 self-attention performs

$$\begin{aligned} \text{Q[B,T,N,H]} \rightarrow_{\text{reshape}} \text{Q[B, T, K, G, H]} \cdot \text{K[B, S, K, H]} &\rightarrow \text{O[B, T, S, K, G]} \\ U &= \text{softmax}_S(\text{O[B, T, S, K, G]}) \\ \text{U[B, T, S, K, G]} \cdot \text{V[B, S, K, H]} &\rightarrow \text{X[B, T, K, G, H]} \end{aligned}$$

So our total bytes is $2 * \text{sizeof}(Q) + 2 * \text{sizeof(K or V)} = 4BTNH + 4BSKH = 4BHK * (TG + S)$, total FLOPs is $4BTSNH + O(BTSN)$ and the arithmetic intensity is $4BTSKGH/(4BHK * (TG + S))$. So basically, during prefill we have $S = T$ so we have an arithmetic intensity of $4BT^2KGH/4BHKT \cdot (G + 1) = TG/(G + 1) = O(T)$. During generation, $T = 1$ so we have $4BSKGH/(4BHK \cdot (G + S)) = SG/(G + S) \rightarrow G$ assuming S is very large. Depending on how you interpret the question, during prefill or training self-attention is compute bound at S=240 assuming no sequence sharding. During generation, we are never compute bound because G is small. Nonetheless, however, you can see that increasing G leads to us being closer to compute bound.

Solution 4.5

This is purely a question of when $4BTDNH = 12BT^2NH$. Simplifying we get $2D = T$, so e.g. for $D = 4096$, this is 8192. This tells us that for most reasonable context lengths, matmul FLOPs are greater.

Solution 4.7

From the spec sheet [1], we find 3,026 TFLOPs/s of FP8 performance with sparsity, or typically half this (`1.513e15` FLOPs/s) without sparsity. 2.79M H800 hours means `2.79e6 * 1.513e15 * 60 * 60 = 1.52e25` total FLOPs. Given the activated parameter count of 37B, this training run should have used about `6 * 37e9 * 14.8e12 = 3.3e24` FLOPs. That means the FLOPs utilization is about `3.3e24 / 1.52e25 = 21.7%`.

Solution 4.8

Because we have E copies of each expert, in int8, we need to load $E \cdot D \cdot F$ bytes. Because each token activates k experts, we have $2 \cdot k \cdot B \cdot D \cdot F$ FLOPs. To be compute-bound with bfloat16 FLOPs, we need an arithmetic intensity over 240 which happens when $(2 \cdot k \cdot BDF)/EDF > 240$ or $k \cdot B/E > 120$.

Therefore, we need $B > 120 \cdot E/k$ to be compute bound. For DeepSeek, this gives us $B > 120 \cdot 256/8 = 3840$. This is a remarkably large batch size at generation time.

Solution 5.1

- FFW parameters: $3LDF$ = `8.5e9`
- Attention parameters: $4DNHL$ = `4.2e9`
- Vocabulary parameters: $2VD$ = `0.3e9`
- Total: `8.5e9 + 4.2e9 + 0.39e9 = 13.1e9`, as expected!

Solution 5.2

The total memory used for the parameters (bf16) and the two optimizer states (fp32, the first and second moment accumulators) is `(2 + 4 + 4) * 13e9 ~ 130GB`. The activations after the first two matmuls are shaped BF and after the last one BD (per the Transformer diagram above), so the total memory for bf16 is $2 \cdot L \cdot (BD + 2 * BF) = 2LB \cdot (D + 2F)$ or `2 * 40 * 16e6 * 5,120 * (1 + 2 * 2.7) ~ 4.2e13 = 42TB`, since `B=16e16`. All other activations are more or less negligible.

Solution 5.3

First, let's write down some numbers. With 32k sequence length and a 3M batch size, we have a sequence batch size of 96. On a TPU v5p 16x16x16 slice, we have `393TB` of HBM.

1. We can't use pure data parallelism, because it replicates the parameters and optimizer states on each chip, which are already around 130GB (from Q2) which is more HBM than we have per-chip (96GB).
2. Let's start by looking purely at memory. Replacing BS=16M with 3M in Q2, we get `~07.86e12` total checkpoint activations, and with the `1.3e11` optimizer state this brings us to almost exactly `8e12 = 8 TB`. The TPUv5p slice has `393TB` of HBM in total, so we are safely under the HBM limit. Next let's look at whether we'll be comms or compute-bound. With 4096 chips and 3 axes of parallelism, we can do a minimum batch size of `850 * 4096 = 3.48M` tokens. That's slightly above our 3M batch size. So we're actually comms-bound, which is sad. So the general answer is **no, we cannot do FSDP alone**.

[1] `https://lenovopress.lenovo.com/lp1814.pdf`

3. Now we know our primary concern is being comms-bound, so let's plug in some numbers. First of all, from the discriminant above, we know our per-chip batch size with mixed FSDP + model parallelism needs to be above $2 \cdot 2550^2 / F = 940$ here, which is actually slightly worse than pure FSDP. Obviously that's sort of an artifact of some of the approximations we made, but this suggests mixed FSDP + model parallelism isn't actually much better. Partly this is because F is so small we can't do a full axis worth of model parallelism. One way around this is to do small subrings of 4 chips of tensor parallelism and dedicate the remaining bandwidth of the first axis to FSDP. We won't do the math out but it's good to check that we probably can do this without being comms-bound.

Solution 6.2

No solution is provided.

Solution 7.1

Parameter count:

- MLP parameter count: $L * D * F * 3$
- Attention parameter count: $L * 2 * D * H * (N + K)$
- Vocabulary parameter: $D * V$ (since we share these matrices)

Our total parameter count is thus $L * D * (3F + 2H * (N + K)) + D * V$. Plugging in the numbers above, we have `64 * 4096 * (3*16384 + 2 * 256 * (32 + 8) + 4096 * 32128) = 18.4e9`. Thus, this model has about 18.4 billion parameters.

Solution 7.6

No solution is provided.

Solution 7.7

Let's work out T_{math} and T_{comms}. All our FLOPs are fully sharded so as before we have $T_{\text{math}} = 4BDF/(N \cdot C)$ but our comms are now

$$T_{\text{2D comms}} = \frac{2BD}{2X \cdot W_{\text{ici}}} + \frac{4BF}{YZ \cdot W_{\text{ici}}} + \frac{2BD}{2X \cdot W_{\text{ici}}} = \frac{2BD}{X \cdot W_{\text{ici}}} + \frac{4BF}{YZ \cdot W_{\text{ici}}}$$

where we note that the AllReduce is twice as expensive and we scale our comms by the number of axes over which each operation is performed. Assuming we have freedom to choose our topology and assuming $F = 4D$ (as in LLaMA-2), we claim (by some basic calculus) that the optimal values for X, Y, and Z are $X = \sqrt{N/8}, YZ = \sqrt{8N}$ so the total communication is

$$T_{\text{2D comms}} = \frac{2B}{W_{\text{ici}}}\left(\frac{D}{X} + \frac{8D}{YZ}\right) = \frac{\sqrt{128}BD}{\sqrt{N} \cdot W_{\text{ici}}} \approx \frac{11.3BD}{\sqrt{N} \cdot W_{\text{ici}}}$$

Firstly, copying from above, normal 1D model parallelism would have $T_{\text{model parallel comms}} = 4BD/(3 \cdot W_{\text{ici}})$, so when are the new comms smaller? We have

$$T_{\text{model parallel comms}} > T_{\text{2D comms}} \iff \frac{4BD}{3 \cdot W_{\text{ici}}} > \frac{\sqrt{128}BD}{\sqrt{N} \cdot W_{\text{ici}}}$$
$$\iff N > 128 \cdot \left(\frac{3}{4}\right)^2 = 81$$

For a general F, we claim this condition is

$$N > 32 \cdot \left(\frac{F}{D}\right) \cdot \left(\frac{3}{4}\right)^2$$

So that tells us if we have more than 81 chips, we're better off using this new scheme. Now this is a slightly weird result because we've historically found ourselves ICI bound at around 20 way tensor parallelism. But here, even if we're communication-bound, our total communication continues to decrease with the number of total chips! What this tells us is that we can continuous to increase our chips, increase our batch size, do more parameter scaling, and see reduced latency.