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*

Google DeepMind

February 4, 2025

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[2]. Also check this list out for more useful concurrent and future reading.

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

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.[3] **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.[4] If we under-

[3]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, for the forseeable future, doing cutting edge research will be inextricably tied to an understanding of how to efficiently scale models to large hardware topologies.

[4]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

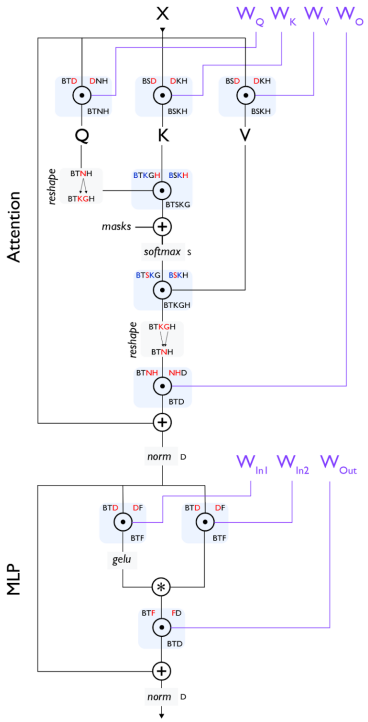
stand our hardware well enough to anticipate where these bottlenecks will arise, we can design or reconfigure our models to avoid them.[5]

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.

in memory. The interplay of per-chip computation, memory bandwidth, and total memory is critical to the scaling story.

[5]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.

High-Level Outline



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.

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 work, both as individual chips and—of critical importance—as an intercon-

nected 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[6] 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.

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

[6]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.

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. Section 12 is a new section that dives into GPUs as well.

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 Blake Hechtman who derived many of the ideas in this doc.

Chapter 1

All About Rooflines

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:

$$T_{\text{math}} = \frac{\text{Computation FLOPs}}{\text{Accelerator FLOPs/s}} \tag{1.1}$$

For instance, an NVIDIA H100 can perform about 9.89e14 bfloat16[1] FLOPs/s while a TPU v6e can perform 9.1e14 FLOPs/s.[2] 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.[3]

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

[1] bf16 is short for bfloat16, a 16-bit floating point format often used in ML.

[2] H100s and B200s can usually only achieve around 80-85% of the claimed peak FLOPs, while TPUs can get closer to 95% in normal use.

[3] Note that these chips are priced differently, and this comparison does not normalize to cost.

to as "HBM bandwidth"[4] On an H100, this is about 3.35TB/s and on TPU v6e this is about 1.6TB/s.

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:

$$T_{\text{comms}} = \frac{\text{Communication Bytes}}{\text{Network/Memory Bandwidth Bytes/s}} \tag{1.2}$$

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.

$$T_{\text{lower}} = \max(T_{\text{math}}, T_{\text{comms}}) \tag{1.3}$$

$$T_{\text{upper}} = T_{\text{math}} + T_{\text{comms}} \tag{1.4}$$

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 $T_{\text{comms}} >$

[4]NVIDIA also calls this "memory bandwidth."

T_{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.

$$\text{Arithmetic Intensity} = \frac{\text{Computation FLOPs}}{\text{Communication Bytes}} \tag{1.5}$$

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**, since the TPU can perform `1.97e14` FLOPs/s and load `8.2e11` bytes/s from HBM.[5] That means if an algorithm has a lower arithmetic intensity than 240 FLOPs/byte, it will be bound by byte loading and thus

[5]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.

we won't make good use of our hardware.[6] 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

$$\begin{aligned}\text{Intensity(dot product)} &= \frac{\text{Total FLOPs}}{\text{Total Bytes}} = \frac{N + N - 1}{2N + 2N + 2} \\ &= \frac{2N - 1}{4N + 2} \to \frac{1}{2}\end{aligned} \tag{1.6}$$

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.[7]

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:

[6] 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.

[7] 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 to be compute-bound on most hardware.

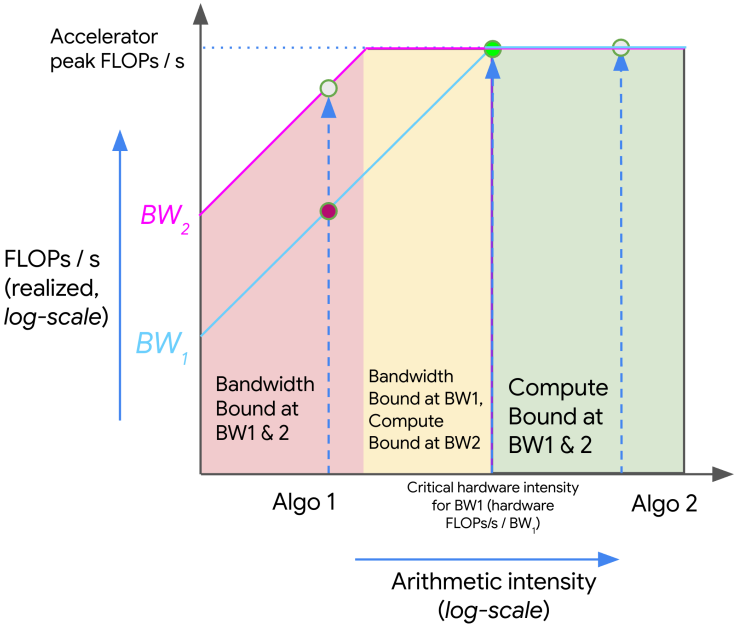


Figure 1.1: An example roofline plot showing two algorithms with different arithmetic intensities (Algo 1 and Algo 2) and their corresponding theoretical peak throughput under different bandwidths (BW1 and BW2). In the red area, an algorithm is bandwidth bound at both bandwidths and is wasting some fraction 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).

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.[8][9] Thus:

$$\text{Intensity(matmul)} = \frac{2BDF}{2BD + 2DF + 2BF} = \frac{BDF}{BD + DF + BF} \tag{1.7}$$

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

$$\frac{BDF}{BD + DF + BF} \approxeq \frac{BDF}{DF} = B \tag{1.8}$$

$$\text{Intensity(matmul)} > \text{Intensity(TPU)} \implies B > \frac{1.97e14}{8.20e11} = 240 \tag{1.9}$$

This is a reasonable assumption for Transformer matmuls since we typically have a local (per-replica) batch size $B < 1024$ tokens (*not sequences*) but D and $F > 8000$. Thus we generally become

[8]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.

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

compute-bound when our per-replica[10] batch size is greater than 240 tokens, a very simple rule!

Key Takeaway

For a bfloat16 matmul to be compute-bound on most TPUs, we need our per-replica token batch size to be greater than 240.[a]

[a]Note that this is *not* the batch size in the usual sense, where it means the batch size in sequences. It turns out most rooflines depend purely on the number of tokens, whether they belong to the same or different sequences. For instance if you have a batch size of 512 sequences of 4096 tokens on 128 GPUs, you have a total batch size of `512 * 4096 = 2M` tokens, and a per-device batch size of 16k tokens.

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. When we decompose a big matmul into smaller matmuls, the tile sizes also matter.[11] We'll discuss the lower-level GPU and TPU details in the next section.

[10]We say per-replica because, if we do some kind of model sharding to increase the number of chips used in the matmul, we scale both our available compute and memory bandwidth by the same amount. Thus the critical batch size is true per independent copy of the model weights.

[11]When we do a large matrix multiplication, we need to break it down into smaller tiles which fit into VMEM/SMEM/TMEM, the higher-bandwidth on-chip memory. This causes us to load chunks multiple times, so it's no longer quite true that we only load $O(N^2)$ bytes. Consider an $(m, k) \cdot (k, n)$ matmul with tile sizes bm, bk, bm. Let $tm = m/bm$, etc. Then the total FLOPs is $2 \cdot tm \cdot tn \cdot tk \cdot bm \cdot bn \cdot bk$ and the total bytes are $2 \cdot tm \cdot tn \cdot (tk \cdot (bm \cdot bk + bk \cdot bn) + 2 \cdot bm \cdot bn)$. Ignoring the last term, we have an intensity of $bm \cdot bn/(bm + bn)$, which is similar to the above.

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.[12]

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

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.

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

Therefore we become compute-bound (now with respect to the inter-chip network) when

$$\text{Intensity(matmul (2-chips))} > \text{Intensity(TPU w.r.t. inter-chip network)}$$

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

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.

A Few Problems to Work

Question 1 [int8 matmul]: Say we want to do the matmul $X[B, D] \cdot_D Y[D, F] \to Z[B, F]$ in int8 precision (1 byte per parameter) instead of bfloat16.[13]

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` (about 2x bfloat16).

Question 2 [int8 + 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

[13]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.

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".*

Question 3: Taking the setup from Question 2, make a roofline plot of peak FLOPs/s vs. B for $F = D = 4096$ and $F = D = 1024$. *Use the exact number of bytes loaded, not an approximation.*

Question 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?

Problem 5 [Memory Rooflines for GPUs]: Using the spec sheet provided by NVIDIA for the H100, 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.*

Chapter 2

How to Think About TPUs

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:

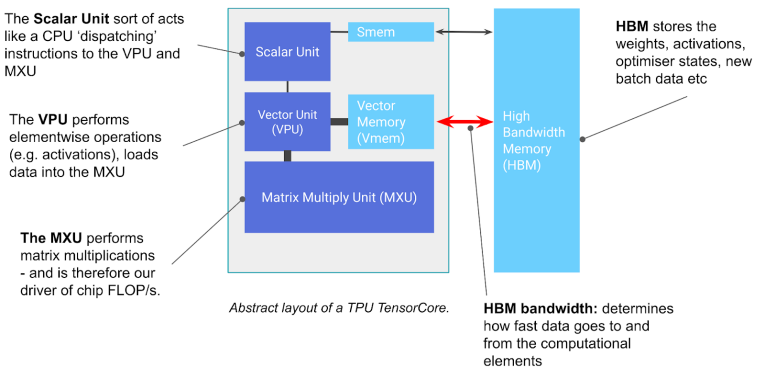


Figure 2.1: The basic components of a TPU chip. The TensorCore is the gray left-hand box, containing the matrix-multiply unit (MXU), vector unit (VPU), and vector memory (VMEM).

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 chip 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 A 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.

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

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

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 A, 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:

[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.

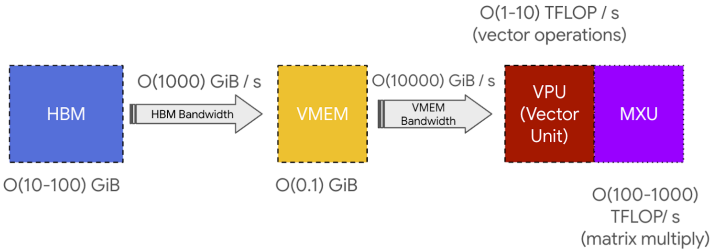
A matmul would look nearly identical except it would load into the MXU instead of the VPU/Vector unit, and the loads and stores would occur in a different order, since the same weight chunk is used for multiple chunks of activations. You can see chunks of data streaming into VMEM, then into the VREGs (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.

Key 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 $\leftrightarrow$ VMEM and VMEM $\leftrightarrow$ systolic array bandwidths set fundamental limits on what computations TPUs can do efficiently.

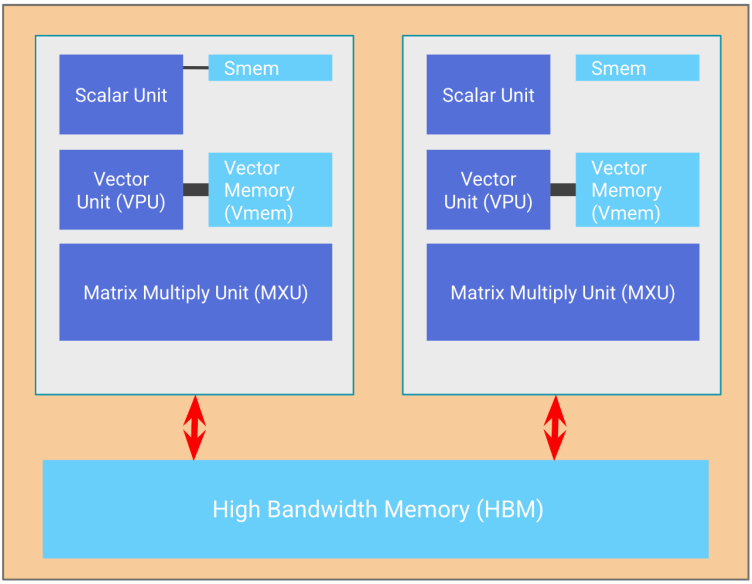
VMEM and arithmetic intensity: VMEM is much smaller than HBM but it has a much higher bandwidth to the MXU. As we saw in Section 1, this means if an algorithm can fit all its inputs/outputs in VMEM, it's much less likely to hit communication bottlenecks. This is particularly helpful when a computation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

[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



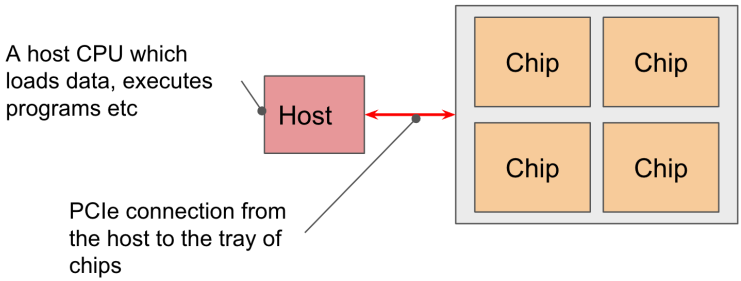
A TPU chip typically (but not always) consists of two TPU cores which share memory and can be thought of as one large accelerator with twice the FLOPs (known as a "mega-core" configuration). This has been true since TPU v4. Older TPU chips have separate memory and are regarded as two separate accelerators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

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.



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]

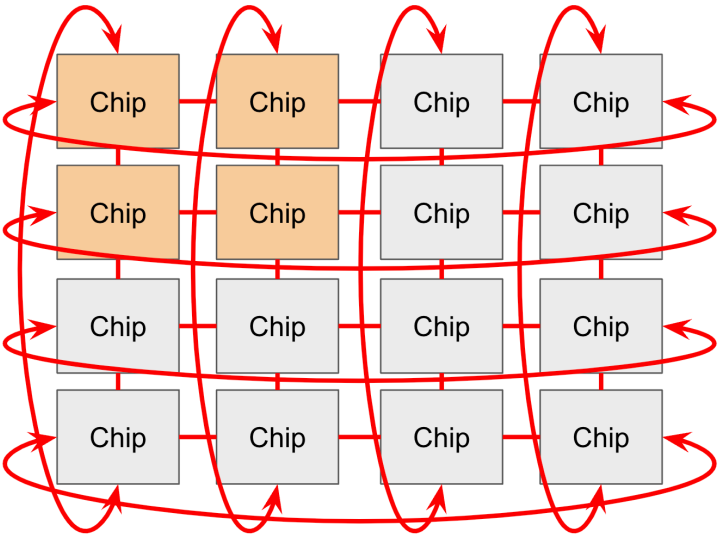
[4]On Cloud TPU VMs, each tray is exposed as part of a separate VM, so there are once again 4 cores visible.



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.

TPU Networking

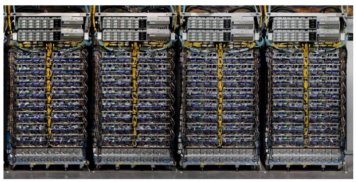
Chips are connected to each other through the ICI network in a Pod. In older generations (TPU v2 and TPU v3), inference chips (e.g., TPU v5e), and Trilium (TPU v6e), ICI ("inter-chip interconnects") connects the 4 nearest neighbors (with edge links to form a 2D torus). TPU v4 and TPU v5p are connected 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.

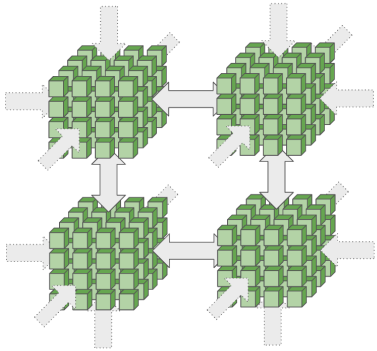
TPU pods (connected by ICI) can get really big: the maximum pod size (called a **superpod**) is `16x16x16` for TPU v4 and `16x20x28` for TPU v5p. These large pods are composed of reconfigurable cubes of `4x4x4` chips connected by optical wraparound links[5] that we can reconfigure to connect very large topologies.

[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.

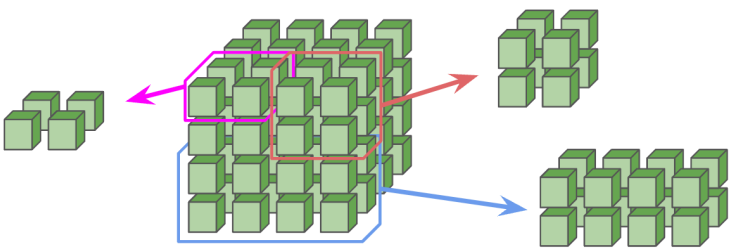


Each rack corresponds to 4x4x4 TPUv4s

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]

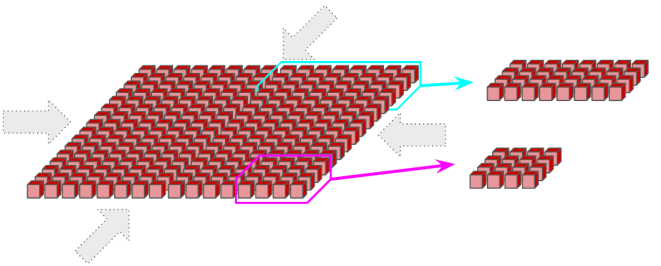


Smaller slices can also be requested.

TPU v5e and Trillium pods consist of a single `16x16` 2D torus

[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.

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.



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 72 for B200 NVL72) 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 small number of hops. On the other hand, TPUs are dramatically cheaper (since NVLink switches are expensive), 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. Read more here.

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

- `2.5e12` bytes/s (2.5 TB/s) of HBM bandwidth per chip.
- `9e10` bytes/s (90 GB/s) of ICI bandwidth per axis, with 3

axes per chip.[7]

- `6.25e9` bytes/s (6.25 GB/s) of DCN (egress) bandwidth per TPU (via 1-2 NICs on each host).[8]

This means that when we split models across multiple chips, we need to be careful to avoid bottlenecking 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.

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.
  - ICI bandwidth: Between a TPU chip and its nearest 4 or 6 neighbors.

[7]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.

[8]TPU v6e has 12.5e9 bytes/s and v5e has 3.125e9 bytes/s.

  - 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 both 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**.

TPU Specs

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	32x32	4x2	32GB	9.0e11	1.4e14	1.4e14
TPU v4p	16x16x16	2x2x1	32GB	1.2e12	2.75e14	2.75e14
TPU v5p	16x20x28	2x2x1	96GB	2.8e12	4.59e14	9.18e14
TPU v5e	16x16	4x2	16GB	8.1e11	1.97e14	3.94e14
TPU v6e	16x16	4x2	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). 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]

PCIe bandwidth is typically around `1.6e10` bytes / second per TPU (`3.2e10` for TPU v6e), while DCN bandwidth is typically around `6.25e9` bytes / second per TPU (`12.5e9` for TPU v6e and `3.125e9` for TPU v5e).

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 Part 3.

Question 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.*

Question 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.

[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.

Question 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.

Question 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 TPU v5e 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?

Question 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?

Question 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.*

Appendix

Appendix A: More on TPU internals

Here we'll dive more deeply into the internal operations of a TPU. Unless otherwise noted, we'll provide specs for a TPU v5p.

VPU

The VPU is the TPU's vector arithmetic core. The VPU consists of a two dimensional SIMD vector machine (the **VPU**) that performs elementwise arithmetic 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.

VREGs: Each TPU v5p core has 64 32-bit VREGs (32 in TPU v4), giving us a total of about `64 * 8 * 128 * 4 = 256kB` of VREG memory per core (or 2x this for the whole chip since we have two cores). A TPU v5p can load 3 registers from VMEM each cycle, and write 1 register to VMEM each cycle.

VPU: The VPU is a 2D vector arithmetic unit of shape `(8, 128)` where the 128 dimension is referred to as lane axis and the dimension of 8 is referred to as the sublane axis. Each (lane, sublane) pair on v5 contains 4 standard floating-point ALUs which are independent of each other. 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.

Pop Quiz [Calculating VPU throughput]: Using the above information, calculate how many vector FLOPs/s a TPU

v5p can perform. A TPU v5p has a clock speed of about 1.75GHz.

Reductions: Generally, communication or reduction across the sublane dimension is easier than across the lane dimension. For instance, the VPU supports an intra-lane shuffle operation that can roll along the axis of size 8 in about a cycle. This can be used to perform efficient reductions along the sublane dimension (just shuffle by 4, 2, and 1 and do 3 pairs of elementwise sums).

Cross-lane reductions are much harder and involve a separate hardware unit called the XLU or "cross lane unit", which is slow and fairly expensive.

Comparison to GPUs: For those familiar with NVIDIA GPUs, each ALU in the VPU is analogous to a CUDA core, and a single VPU lane is analogous to a "Warp Scheduler", i.e. the set of usually 32 CUDA Cores that perform SIMD arithmetic. Reductions within the lane are pretty easy, but if we need to cross lanes, we need to transit at least VMEM/XLU/SMEM which is much slower. See the GPU section for more details.

Scalar Core

The scalar core is the control unit of the TPU. It fetches and dispatches all instructions and executes transfers from HBM into VMEM, and can be programmed to do scalar metadata work. Because the scalar core is single-threaded, 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 scalar core controls a VPU (consisting of 4096 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.

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 per-

form one `bfloat16[8,128] @ bf16[128x128] -> f32[8,128]`[10] 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.

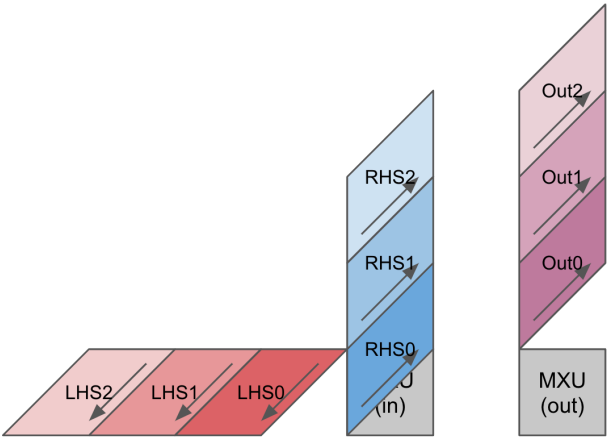
- Weights (**W**, the `128x128` input) are passed down from above (called the RHS) while inputs (**X**, the `8x128` input) are passed in from the left (called the LHS).

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.

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:

[10] 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.



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.

Here's a bad animation of a bf16[2, 3] x bf16[3, 3] matrix multiplication, which you could imagine as a matmul of a 2x3 weight matrix with an input activation of batch 1 and size 3. This is rotated compared to the previous slides and inputs flow out to the right instead of down, but you can roughly see the structure.

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. Here's a good animation for this.[11]

Trillium (TPU v6e) has a `256x256` systolic array, which means it can perform 4x more FLOPs / cycle. This also means the di-

[11] Available at: https://www.youtube.com/watch?v=sJltBQ4MOHA

mensions of your tensors needs to be twice as large to utilize the MXU fully.

This blog post has another excellent animation of a systolic array multiplication for a fixed weight matrix.[12]

[12] Available at: https://fleetwood.dev/posts/domain-specific-architectures#google-tpu

Chapter 3

Sharded Matrices and How to Multiply Them

Partitioning Notation and Collective Operations

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

Here's an example 2D array $\mathbf{A}$ sharded across 4 TPUs:

[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.

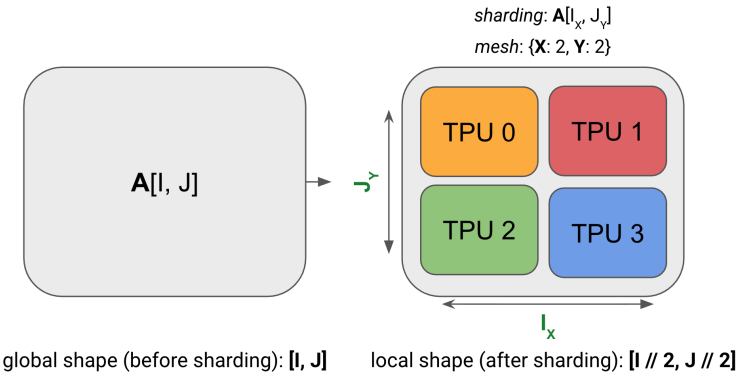


Figure 3.1: An example array of shape $\mathbf{A}[\mathrm{I}, \mathrm{J}]$ gets sharded across 4 devices. Both dimensions are evenly sharded across 2 devices with a sharding $\mathbf{A}[\mathrm{I}_X, \mathrm{J}_Y]$. Each TPU holds 1/4 of the total memory.

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

A unified notation for sharding

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

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

- **Mesh:** the device mesh above `Mesh(devices=((0, 1), (2,`

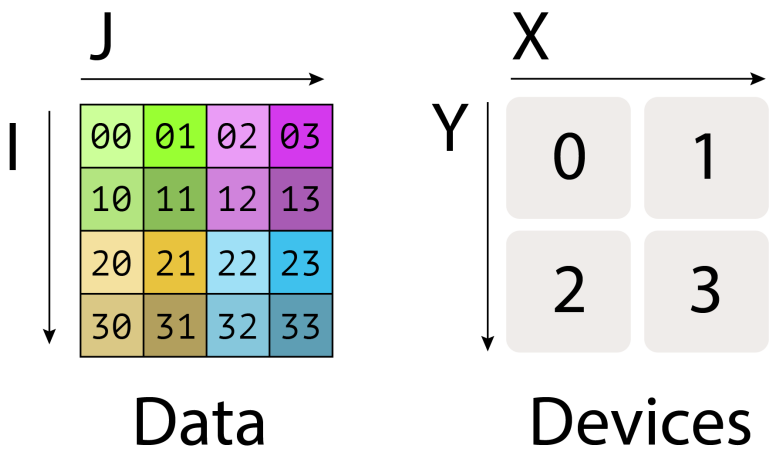
`3))`, `axis_names=('X', 'Y'))`, which tells us we have 4 TPUs in a 2x2 grid, with axis names X and Y.

- **Sharding:** $A[I_X, J_Y]$, which tells us to shard the first axis, I, along the mesh axis X, and the second axis, J, along the mesh axis Y. This sharding tells us that each shard holds $1/(|X| \cdot |Y|)$ of the array.

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

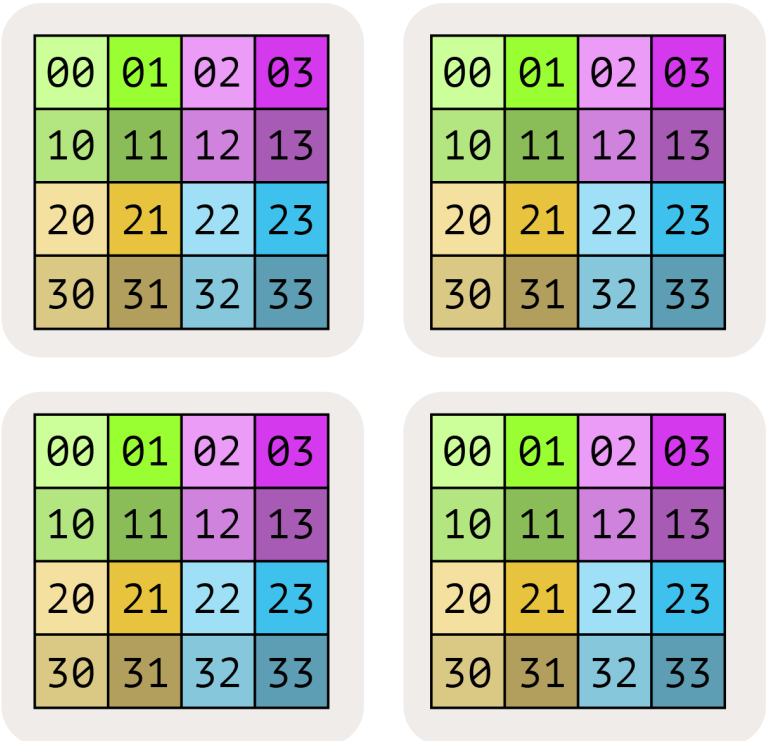
Pop Quiz [2D sharding across 1 axis]: Consider an array `fp32[1024, 4096]` with sharding $A[I_{XY}, J]$ and mesh `{'X': 8, 'Y': 2}`. How much data is held by each device? How much time would it take to load this array from HBM on H100s (assuming `3.4e12` memory bandwidth per chip)?

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

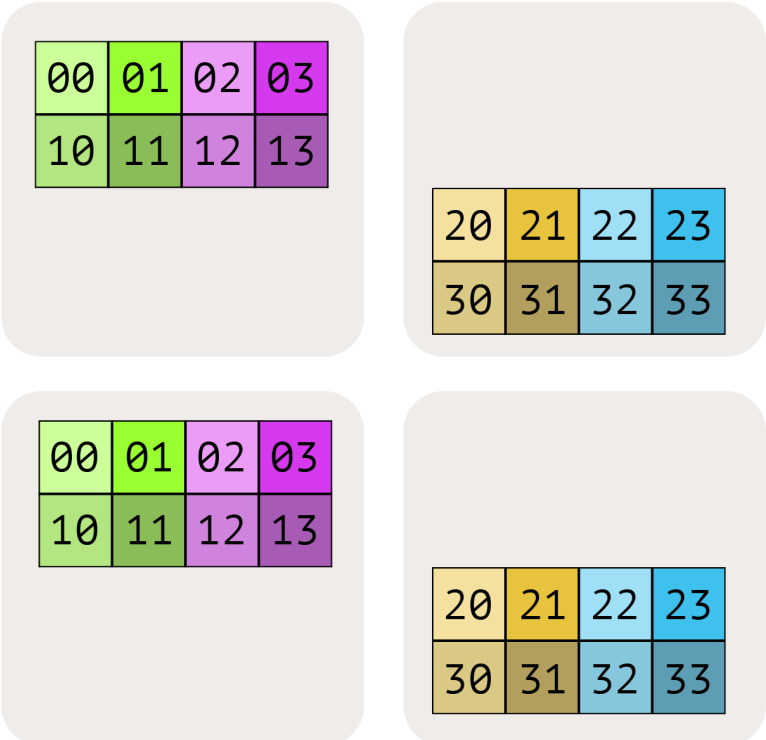


We write the *fully-replicated* form of the matrix simply as

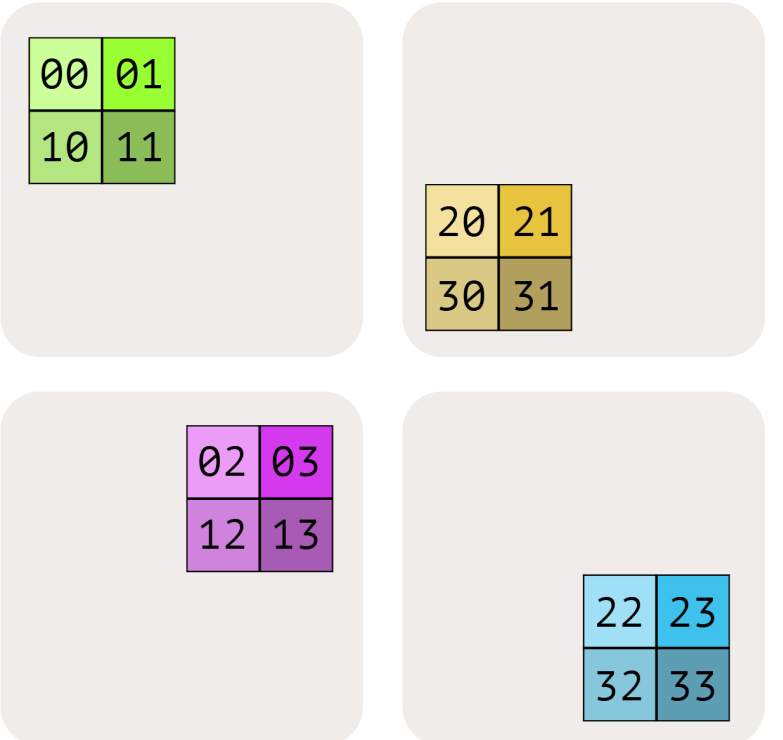
$A[I, J]$ with no sharding assignment. This means that *each* device contains a full copy of the entire matrix.



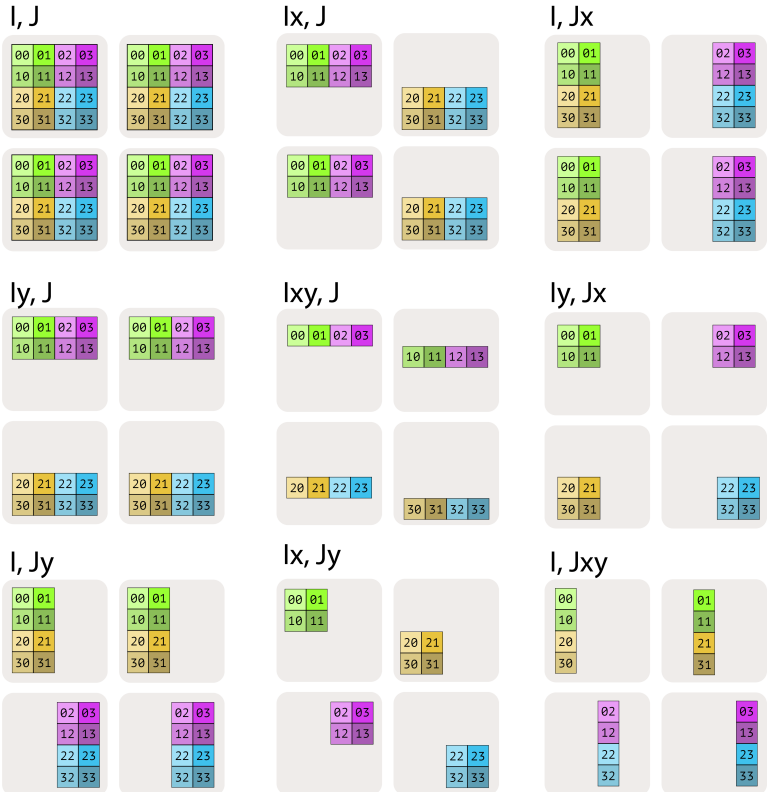
We can indicate that one of these dimensions has been partitioned across a mesh axis with a subscript mesh axis. For instance $A[I_X, J]$ would mean that the **I** logical axis has been partitioned across the **X** mesh dimension, but that the **J** dimension is *not* partitioned, and the blocks remain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



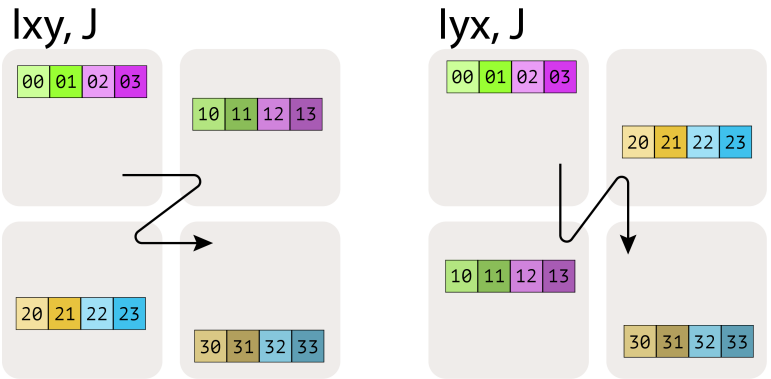
$A[I_X, J_Y]$ means that the **I** logical axis has been partitioned across the **X** mesh axis, and that the **J** dimension has been partitioned across the **Y** mesh axis.



We illustrate the other possibilities in the figure below:



Here $A[I_{XY}, J]$ means that we treat the $\mathbf{X}$ and $\mathbf{Y}$ mesh axes as a larger flattened dimension and partition the $\mathbf{I}$ 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?

How do we describe this in code?

So far we've avoided talking about code, but now is a good chance for a sneak peek. JAX uses a named sharding syntax that very closely matches the abstract syntax we describe above. We'll talk more about this in Section 10, but here's a quick preview. You can play with this in a Google Colab 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

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

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

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

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

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 $\mathbf{A}$ is `[2, 1024]` and for $\mathbf{B}$ is `[2048, 4096]`. JAX/XLA will automatically add communication across these arrays as necessary to perform the final multiplication.

Computation With Sharded Arrays

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

Obviously, this depends on the computation involved.

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

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

You can think of this in terms of "block matrix multiplication". To understand this, it can be helpful to recall the concept of a "block matrix", or a nested matrix of matrices:

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} = \left(\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} a_{02} & a_{03} \\ a_{12} & a_{13} \end{bmatrix} \right) = \begin{pmatrix} \mathbf{A_{00}} & \mathbf{A_{01}} \\ \mathbf{A_{10}} & \mathbf{A_{11}} \end{pmatrix} \tag{3.1}$$

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:

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix} = \begin{pmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{pmatrix} \tag{3.2}$$

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

1. **Case 1:** neither input is sharded along the contracting dimension. *We can multiply local shards without any communication.*

2. **Case 2:** one input has a sharded contracting dimension. *We typically "AllGather" the sharded input along the contracting dimension.*

3. **Case 3:** both inputs are sharded along the contracting dimension. *We can multiply the local shards, then "AllReduce" the result.*

4. **Case 4:** both inputs have a non-contracting dimension sharded along the same axis. We cannot proceed without AllGathering one of the two inputs first.

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

Case 1: neither multiplicand has a sharded contracting dimension

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

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

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

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

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

Case 2: one multiplicand has a sharded contracting dimension

Let's consider what to do when one input $\mathbf{A}$ is sharded along the contracting $\mathbf{J}$ dimension and $\mathbf{B}$ is fully replicated:

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

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

$$\mathbf{AllGather}_X[I, J_X] \to \mathbf{A}[I, J]$$

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

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

Key Takeaway

When multiplying matrices where one of the matrices is sharded along the contracting dimension, we generally All-Gather it first so the contraction is no longer sharded, then do a local matmul.

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

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

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

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

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

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

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

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

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

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

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

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

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

[2]A GPU AllGather can also work like this, where you create a ring out of the GPUs in a node and pass the chunks around in that (arbitrary) order.

[3]The factor of 2 in the numerator comes from the fact that we're using the bidirectional bandwidth. We send V/X in each direction, or $2V/X$ total.

[4]technically, $\lfloor X/2 \rfloor$

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.

Key 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.

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

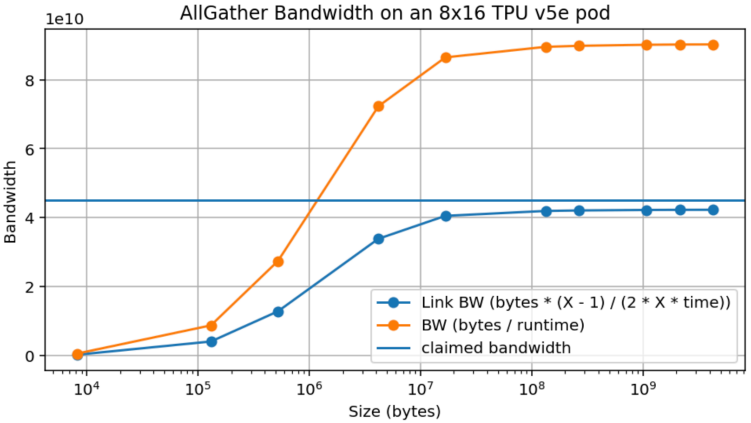


Figure 3.2: Empirical bandwidth and estimated link bandwidth for TPU v5e during an AllGather. BW in orange is the actual bytes per second AllGathered, while the blue curve shows the empirical unidirectional link bandwidth calculated according to the known cost of the collective.

Note that we not only achieve about 95% of the peak claimed bandwidth (`4.5e10`) but also that we achieve this peak at about 10MB, which when 16-way sharded gives us about 500kB per device (*aside*: this is much better than GPUs).

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

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

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

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

Pop Quiz 2 [AllGather time]: Using the numbers from Part 2, how long does it take to perform the $\text{AllGather}_{\text{Y}}([\text{E}_{\text{Y}}, \text{F}]) \rightarrow [\text{E},$

F] on a TPUv5e with a 2D mesh `{'X': 8, 'Y': 4}`, $E = 2048$, $F = 8192$ in bfloat16? What about with $E = 256, F = 256$?

For part (1), we can use the formula above. Since we're performing the AllGather over one axis, we have $T_{\text{comms}} = 34e6/9e10 = 377\text{us}$. 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 $T_{\text{comms}} = 3 * 8.4e6/4.5e10 = 560\mu s$. **Here's an actual profile** from this Colab, which 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.

Key Takeaway

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

Case 3: both multiplicands have sharded contracting dimensions

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

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

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 $\{\ \mathbf{U_X}\ \}$ 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 $\square$ 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 $\mathbf{A} \bullet \mathbf{B}$ has at that entry. We still need to perform that sum over **P**, which we sharded over mesh axis **X**, to obtain the full $\mathbf{A} \bullet \mathbf{B}$. 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:

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

AllReduce removes partial sums, resulting in *each* device along the axis having the same fully-summed value. AllReduce is the

second of several key communications we'll discuss in this section, the first being the AllGather, and the others being ReduceScatter and AllToAll. An AllReduce takes an array with an unreduced (partially summed) axis and performs the sum by passing those shards around the unreduced axis and accumulating the result. The signature is

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

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

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

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

What about a ReduceScatter? Just as the AllReduce removes a subscript ($F_Y \to F$ above), a ReduceScatter sums an unreduced/partially summed array and then scatters (shards) a different logical axis along the same mesh axis. $[F]\{U_Y\} \to [F_Y]$.

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

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

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

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

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

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

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

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

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

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

or

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

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

A Deeper Dive into TPU Communication Primitives

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

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

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

Our final communication primitive: the AllToAll

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

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

AllToAlls are typically required to rearrange sharded layouts between different regions of a sharded computation that don't have

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

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

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

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

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

More about the ReduceScatter

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

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

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

[5]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.

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.

How AllGather and ReduceScatter are derivatives of eachother This stems from the fact that broadcasts and reductions are transposes of eachother as linear operators, and AllGather and ReduceScatter are outer products (also known as Kronecker products) of broadcast and reduce, respectively. Concretely, if we have a vector $x \in \mathbb{R}^n$, any number of devices $p \in \mathbb{N}$, and we let $u = (1, \dots, 1) \in \mathbb{R}^p$, we can define broadcast and reduce in the following way, which should match your intuitive understanding of them:

$$\begin{aligned} \text{broadcast} &: \mathbb{R}^n \to \mathbb{R}^{pn} \\ \text{broadcast} &= u \otimes \mathbf{I}_n \\ \text{reduce} &: \mathbb{R}^{pn} \to \mathbb{R}^n \\ \text{reduce} &= u^T \otimes \mathbf{I}_n \end{aligned}$$

Let's see how this looks in an example, where $n = 1$, $p = 2$. If $x = (7)$, we have

$$\text{broadcast}(x) = \left(\begin{pmatrix}1\\1\end{pmatrix} \otimes (1)\right) x = \begin{pmatrix}1\\1\end{pmatrix} x = \begin{pmatrix}7\\7\end{pmatrix} \in \mathbb{R}^{pn}$$

. This matches what we'd expect, broadcasting a vector in $\mathbb{R}^n$ to $\mathbb{R}^{pn}$. Now letting $y = (8, 9)$, we have

$$\text{reduce}(y) = \left(\begin{pmatrix}1 & 1\end{pmatrix} \otimes (1)\right) y = \begin{pmatrix}1 & 1\end{pmatrix} \begin{pmatrix}8\\9\end{pmatrix} = (17)\,.$$

This again matches what we'd expect, reducing a vector in $\mathbb{R}^{pn}$ to a vector in $\mathbb{R}^n$. Since $(A \otimes B)^T = A^T \otimes B^T$ for any two matrices A and B, we see that $\text{reduce} = \text{broadcast}^T$. We recover AllGather and ReduceScatter as the following outer products:

$$\begin{aligned}
\text{AllGather} &: \mathbb{R}^{pn} \to \mathbb{R}^{p^2 n} \\
\text{AllGather} &= \text{broadcast} \otimes \mathbf{I}_p \\
\text{ReduceScatter} &= \mathbb{R}^{p^2 n} \to \mathbb{R}^{pn} \\
\text{ReduceScatter} &= \text{reduce} \otimes \mathbf{I}_p
\end{aligned}$$

Here we think of $\mathbb{R}^{p^2 n}$ as $\mathbb{R}^{p \times pn}$, so one $\mathbb{R}^{pn}$ vector for each of our p devices. We suggest playing around with small examples, say $n = 2$, $p = 3$, to see what these operators look like as matrices. Using the same transposition property, we once more obtain $\text{AllGather}^T = \text{ReduceScatter}$, and of course $\text{ReduceScatter}^T = \text{AllGather}$. This transposition will arise during backpropagation, since if we have $y = Ax$ for some linear operator A, such as AllGather or ReduceScatter, then during backpropagation we will have the derivative of the loss with respect to y, $\frac{\partial L}{\partial y}$, and we obtain $\frac{\partial L}{\partial x}$ as $\frac{\partial L}{\partial x} = A^T \frac{\partial L}{\partial y}$. This shows how the derivative of AllGather will be ReduceScatter, and viceversa.

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.

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

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

What Have We Learned?

- The sharding of an array is specified by a **Mesh** that names the physical, hardware axes of our TPU mesh and a **Sharding** that assigns mesh axis names to the logical axes of the array.
  - For example, $\mathbf{A}[\mathrm{I}_{\mathrm{XY}}, \mathrm{J}]$ describes an abstract array **A** with its first dimension sharded along two mesh axes X and Y. Combined with Mesh(mesh_shape=(4, 8), axis_names=('X', 'Y')) or the abbreviated Mesh({'X': 4, 'Y': 8}), this tells us our array is sharded 32 ways along the first dimension.
- **Arithmetic with sharded arrays works exactly like with unsharded arrays unless you perform a contraction along a sharded axis**. In that case, we have to introduce some communication. We consider four cases:
  1. *Neither array is sharded along the contracting dimension*: no communication is needed.
  2. *One array is sharded along the contracting dimension* (or the contracting dimensions are sharded along different axes): we AllGather one of the inputs before performing the operation.
  3. *Both arrays are identically sharded along the contracting dimension:* we multiply the shards locally then perform an AllReduce or ReduceScatter.

4. *Both arrays are sharded along the same mesh axis along a non-contracting dimension:* we AllGather one of the inputs first.

- TPUs use roughly **4 core communication primitives**:

1. AllGather: $[A_X, B] \to [A, B]$
2. ReduceScatter: $[A, B]\{U_X\} \to [A, B_X]$
3. AllToAll: $[A, B_X] \to [A_X, B]$
4. AllReduce: $[A_X, B]\{U_Y\} \to [A_X, B]$ (technically not a primitive since it combines a ReduceScatter + AllGather)

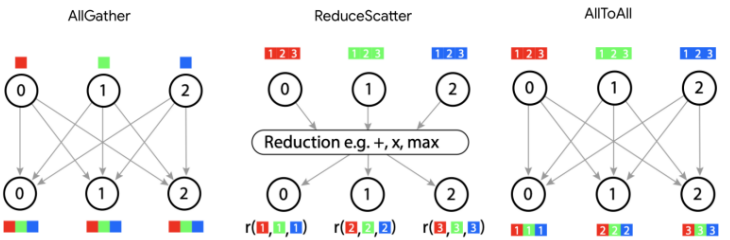


Figure 3.3: All collective communication primitives used in TPU sharding.

- 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:

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

$$\longrightarrow \frac{\text{Data volume}}{\text{bandwidth (bidirectional)}} \tag{3.3}$$

- 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:

Table 3.1: Summary of collective communication operations

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

Some Problems to Work

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

Question 1 [replicated sharding]: An array is sharded $A[I_X, J, K, ...]$ (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?

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

4096 in bfloat16? How about $\text{AllGather}_{XY}([B_X, D_Y])$? How about $\text{AllReduce}_Z([B_X, D_Y]\{U_Z\})$?

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

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

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

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

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

Question 7: A typical Transformer block has two matrices $B[D, F]$ and $C[F, D]$ where $F \gg D$. With a batch size B, the whole block is $C \cdot B \cdot x$ with $x[B, D]$. Let's pick $D = 8192$, $F = 32768$, and $B = 128$ and assume everything is in bfloat16. Assume we're running on a TPUv5e 2x2 slice but assume each TPU only has

300MB of free memory. How should **B, C, and the output be sharded to stay below the memory limit while minimizing overall time? How much time is spent on comms and FLOPs?**

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

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

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

Answer the following:

1. Explicitly write out this algorithm for matrices $A[N, M]$ and $B[M, K]$, using indices to show exactly what computation is done on what device. Assume A is sharded as $A[I, J_X]$ across ND devices, and you want your output to be replicated across all devices.

2. Now suppose you are ok with the final result not being replicated on each device, but instead sharded (across either the N or K dimension). How would the algorithm above change?

3. Looking purely at the communication cost of the strategy above (in part (b), not (a)), how does this communication

cost compare to the communication cost of the algorithm in which we first AllGather A and then do the matmul?

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

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

2. Now let's think about an AllToAll, still in the single-directional ICI case. How is the algorithm different in this case than the all-gather case? Calculate the number of scalars that are transferred across a single ICI link in this algorithm.

3. You should have found that the ratio between your answers to part (a) and part (b) is a nice number. Explain where this factor comes from in simple terms.

4. Now let's add bidirectional communication. How does this affect the total time needed in the all-gather case?

5. How does adding bidirectional communication affect the total time needed in the AllToAll case?

6. Now simply explain the ratio between AllGather time and AllToAll time in a bidirectional ring.

Chapter 4

All the Transformer Math You Need to Know

Counting Dots

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

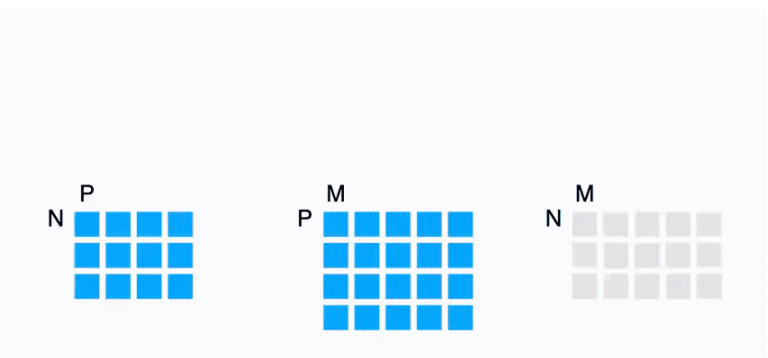
array	shape
x	[P]
y	[P]
A	[N P]
B	[P M]

- 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 a matrix-vector product for each of the M columns of B, for $2NPM$ FLOPs total.
- In general, if we have two higher dimensional arrays C and D, where some dimensions are CONTRACTING and some are BATCHING. (e.g. $C[GHIJKL], D[GHMNKL]$) then the

FLOPs cost of this contraction is two times the product of all of the C and D dimensions where the batch and contraction dimensions are only counted once, (e.g. $2GHIJMNKL$). Note that a dimension is only batching if it occurs in both multiplicands. (Note also that the factor of 2 won't apply if there are no contracting dimensions and this is just an elementwise product.)

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

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



Forward and reverse FLOPs

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

If we imagine $\mathbf{B}$ is just one matrix in a larger network and $\mathbf{A}$ are our input activations with $\mathbf{C} = \mathbf{A}\ \mathbf{B}$, the derivative of the loss $\mathbf{L}$ with respect to $\mathbf{B}$ 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 $\mathbf{A}$ 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 $\mathbf{dL/dC}$ 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 dL/dC is used to compute dL/dB above).

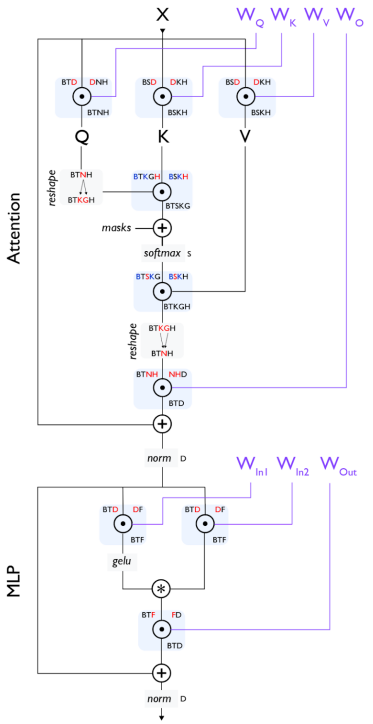
Adding these up, we see that **during training, we have a total of 6NPM FLOPs**, compared to 2NPM during inference: 2NPM in the forward pass, 4NPM in the backward pass. Since PM is the number of parameters in the matrix, this is the simplest form of the famous 6∗num parameters∗num tokens approximation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

4.1 Transformer Accounting

Transformers are the future. Well, they're the present at least. Maybe a few years ago, they were one of many architectures. But

today, it's worth knowing pretty much every detail of the architecture. We won't reintroduce the architecture but this blog and the original Transformer paper may be helpful references.

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



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

Note [**gating einsum**]: The diagram above uses a "gating einsums" Shazeer [2020] 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) Shazeer [2019] K=1 and for Grouped MQA (GMQA) Ainslie et al. [2023] K merely has to divide N.

4.2 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.

4.2.1 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
	$\approx 18BTDF$	$3DF$

4.2.2 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
$Q[B,T,K,G,H] \cdot K[B,S,K,H]$	$6BTSKGH = 6BTSNH$
$\text{softmax}_S \;\; L[B,T,S,K,G]$	$O(BTSKG) = O(BTSN)$
$S[B,T,S,K,G] \cdot V[B,S,K,H]$	$6BTSKGH = 6BTSNH$
	$\approx 12BTSNH = 12BT^2NH$

4.2.3 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

4.2.4 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

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

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.)

4.2.5 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 $N = K$:

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

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

4.3 Miscellaneous Math

4.3.1 Sparsity and Mixture-of-Experts

We'd be remiss not to briefly discuss Mixture of Experts (MoE) models Shazeer et al. [2017], 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.

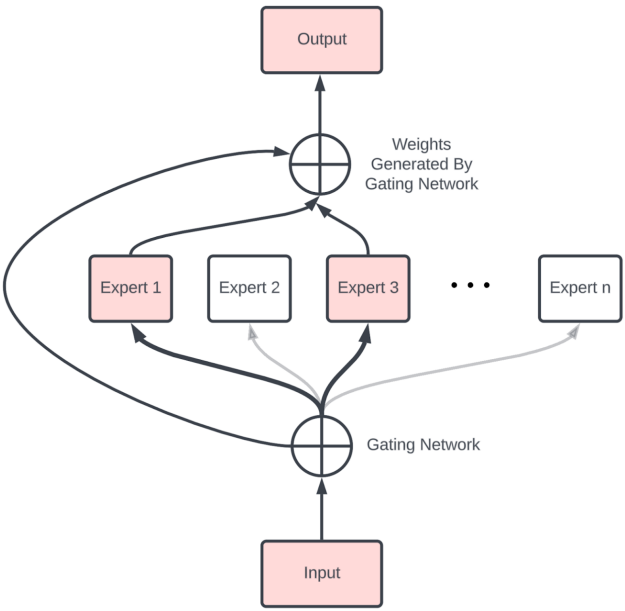


Figure 4.2: An example MoE layer with n experts. The gating expert routes each token to k of them, and the output of those k MLPs get summed. Our parameter count is n times the size of each expert, but only k are used for each token. Source.

Compared to a dense model, an MoE introduces new comms, primarily two AllToAlls (one before and one after the MoE block) that route tokens to the correct expert and bring them back to their

home device.[1] However as we saw in the previous section, the cost of each AllToAll is only 1/4 that of a comparable AllGather along a single axis (for a bidirectional ring).

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

$$f(x) = \exp(g(x)) \tag{4.1}$$

$$\frac{df}{dx} = \exp(g(x)) \cdot \frac{dg}{dx} \tag{4.2}$$

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 $8ND$.

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

- **Big matmuls only:** another simple policy is to only save the outputs of large matmuls. This lets us avoid recomputing any large matmuls during the backward pass, but still makes us recompute other activation functions and parts of attention. This reduces 20 per layer to closer to 7 per layer.

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

4.3.3 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 $K \ll N$.

4.4 What Should You Take Away from this Section?

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

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

- The parameter count of the MLP block dominates the total parameter count and the MLP block also dominates the FLOPs budget as long as the sequence length $T < 8D$.
- The total FLOPs budget during training is well approximated by 6·num_params·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.

4.5 A Few Problems to Work

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

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

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

Question 4: What is the arithmetic intensity of self-attention (ignoring the Q/K/V/O projections)? *Give the answer as a function of the Q and KV lengths T and S.* At what context length

is attention compute-bound? Given the HBM bandwidth of our TPUs, plot the effective relative cost of attention to the FFW block as the context length grows.

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

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

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

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

4.6 Appendix

4.6.1 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

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

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

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

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

where

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

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

Algorithm 1 FLASHATTENTION

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

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

Figure 4.3: Flash Attention algorithm

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.

Chapter 5

How to Parallelize a Transformer for Training

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

In this section, we'll discuss four common parallelism schemes: (pure) **data parallelism, fully-sharded data parallelism** (FSDP / ZeRO sharding), **tensor parallelism** (also known as model parallelism), and (briefly) **pipeline parallelism**. For each, we'll show what communication cost we incur and at what point

that cost starts to bottleneck our compute cost.[1] For this section, you can focus solely on inter-chip communication costs, since as long as we have a large enough single-chip batch size, the transfer of data from HBM to MXU is already overlapped with computation.

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

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

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

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]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.

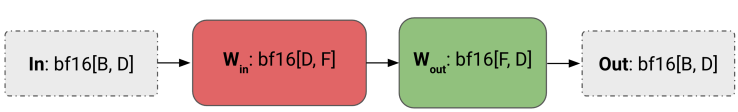


Figure 5.1: A simplified Transformer layer. We treat each FFW block as a stack of two matrices $\mathbf{W}_{\text{in}}$: `bf16[D, F]` (up-projection) and $\mathbf{W}_{\text{out}}$: `bf16[F, D]` (down-projection) with an input **In**: `bf16[B, D]`.

Here are the 4 parallelism schemes we will discuss. Each scheme can be thought of as uniquely defined by a sharding for **In**, $\mathbf{W}_{\text{in}}$, $\mathbf{W}_{\text{out}}$, 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: $\text{In}[L_Z, B, D][i] \cdot_D W_{\text{in}}[L_Z, D, F][i] \cdot_F W_{\text{out}}[L_Z, F, D][i] \to \text{Out}[L_Z, B, D][i]$

5.1.1 Data Parallelism

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

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

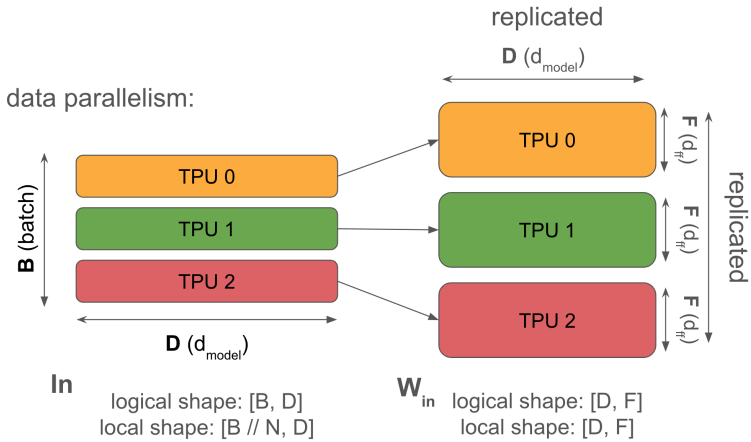


Figure 5.2: Pure data parallelism (forward pass). Our activations (left) are fully sharded along the batch dimension and our weights are fully replicated, so each TPU has an identical copy of the weights. This means the total memory of our weights is increased by a factor of N, but no communication is required on the forward-pass.

Algorithm

Pure Data Parallelism Algorithm:
Forward pass: need to compute Loss$[B_X]$

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

Backward pass: need to compute $\text{dW}_{\text{out}}[\text{F}, \text{D}]$, $\text{dW}_{\text{in}}[\text{D}, \text{F}]$

1. $\text{dOut}[B_X, \text{D}] = ...$
2. $\text{dW}_{\text{out}}[\text{F}, \text{D}]\ \{U_X\} = \text{Tmp}[B_X, \text{F}] *_B \text{dOut}[B_X, \text{D}]$

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

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 $\textbf{AllReduce}(\text{Loss}[B_X])$, we only need to compute the AllReduce on the backward pass when averaging weight gradients.

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

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

Why not do this? Pure data parallelism does nothing to reduce memory pressure from model parameters or optimizer states, which means pure data parallelism is rarely useful for interesting models at scale where our parameters + optimizer state don't fit in

a single TPU. To give a sense of scale, if we train with parameters in bf16 and optimizer state in fp32 with Adam[2], the largest model we can fit has TPU memory/10 parameters, so e.g. on a TPUv5p chip with 96GB of HBM and pure data parallelism this is about 9B parameters.

Key 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.[a]

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

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[3]. 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

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

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

the ICI bandwidth W_{ici}; specifically the AllReduce time is $2 \cdot$ 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

$$T_{\text{comms}} = \frac{2 \cdot 2 \cdot 2 \cdot D \cdot F}{W_{\text{ici}}}. \tag{5.1}$$

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

$$T_{\text{math}} = \frac{2 \cdot 2 \cdot 2 \cdot B \cdot D \cdot F}{X \cdot C} \tag{5.2}$$

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

$$\frac{B}{X} > \frac{C}{W_{\text{ici}}}. \tag{5.3}$$

The upshot is that, to remain compute-bound with data parallelism, we need the per-device batch size B/X to exceed the ICI operational intensity, C/W_{ici}. This is ultimately a consequence of the fact that the computation time scales with the per-device

batch size, while the bandwidth-bound 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!**

Key Takeaway

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

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

- **Definition:** M_X (M_Y, M_Z, etc.) is the number of hardware mesh axes that a given parallelism strategy spans.

- **Effect (bandwidth-bound):** Using M axes provides ($\approx M$ times) aggregate link bandwidth, so collective time scales $\propto 1/M_X$.

5.1.2 Fully-Sharded Data Parallelism (FSDP)

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

Fully-sharded data parallelism (often called FSDP or ZeRO-sharding Rajbhandari et al. [2019]) 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.**

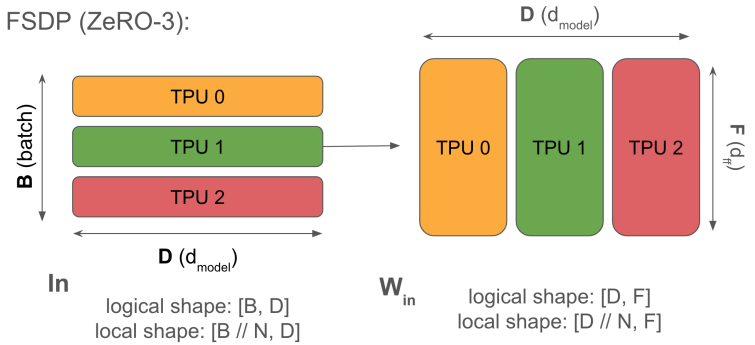


Figure 5.3: FSDP shards the contracting dimension of W_{in} and the output dimension of W_{out} along the data dimension. This reduces memory but (from Section 3) requires us to gather the weights for W before we perform the matmul. Note that the activations (left) *are not sharded along the contracting 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.

Algorithm

Fully-Sharded Data Parallelism (FSDP):
Forward pass: need to compute Loss$[B_X]$

1. $W_{\text{in}}[\text{D}, \text{F}] = \textbf{AllGather}(W_{\text{in}}[D_X, \text{F}])$ (*not on critical path, can do it during previous layer*)

2. $\text{Tmp}[B_X, \text{F}] = \text{In}[B_X, \text{D}] *_D W_{\text{in}}[\text{D}, \text{F}]$ (*can throw away W_{in}[D, F] now*)

3. $W_{\text{out}}[\text{F}, \text{D}] = \textbf{AllGather}(W_{\text{out}}[\text{F}, D_X])$ (*not on critical path, can do it during previous layer*)

4. $\text{Out}[B_X, \text{D}] = \text{Tmp}[B_X, \text{F}] *_F W_{\text{out}}[\text{F}, \text{D}]$

5. $\text{Loss}[B_X] = ...$

Backward pass: need to compute $\text{dW}_{\text{out}}[\text{F}, D_X]$, $\text{dW}_{\text{in}}[D_X, \text{F}]$

1. $\text{dOut}[B_X, \text{D}] = ...$

2. $\text{dW}_{\text{out}}[\text{F}, \text{D}]\ \{U_X\} = \text{Tmp}[B_X, \text{F}] *_B \text{dOut}[B_X, \text{D}]$

3. $\text{dW}_{\text{out}}[\text{F}, D_X] = \textbf{ReduceScatter}(\text{dW}_{\text{out}}[\text{F}, \text{D}]\ \{U_X\})$ (*not on critical path, can be done async*)

4. $W_{\text{out}}[\text{F}, \text{D}] = \textbf{AllGather}(W_{\text{out}}[\text{F}, D_X])$ (*can be done ahead of time*)

5. $\text{dTmp}[B_X, \text{F}] = \text{dOut}[B_X, \text{D}] *_D W_{\text{out}}[\text{F}, \text{D}]$ (*can throw away W_{out}[F, D] here*)

6. $\text{dW}_{\text{in}}[\text{D}, \text{F}]\ \{U_X\} = \text{dTmp}[B_X, \text{F}] *_B \text{In}[B_X, \text{D}]$

7. $\text{dW}_{\text{in}}[D_X, \text{F}] = \textbf{ReduceScatter}(\text{dW}_{\text{in}}[\text{D}, \text{F}]\ \{U_X\})$ (*not on critical path, can be done async*)

8. $W_{\text{in}}[\text{D}, \text{F}] = \textbf{AllGather}(W_{\text{in}}[D_X, \text{F}])$ (*can be done ahead of time*)

9. $\text{dIn}[B_X, \text{D}] = \text{dTmp}[B_X, \text{F}] *_F W_{\text{in}}[\text{D}, \text{F}]$ (*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[4], we can basically always do ZeRO-3 sharding, which shards the parameters, gradients, and optimizer states across a set of devices.

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

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

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

Therefore, as with pure data-parallelism, we are compute bound when $B/X > C/W_{\text{ici}}$, i.e. when the per-device batch size

[4]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.

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.

Key Takeaway

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

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

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

Key Takeaway

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

In a fully-sharded data-parallel AllReduce we move the weights across chips. We can also shard the feedforward dimension of the model and move the activations during the layer — this is called "1D model parallelism" or Megatron sharding Shoeybi et al. [2019]. This can unlock a smaller efficient batch size per pod. The figure below shows an example of a single matrix sharded in this way:

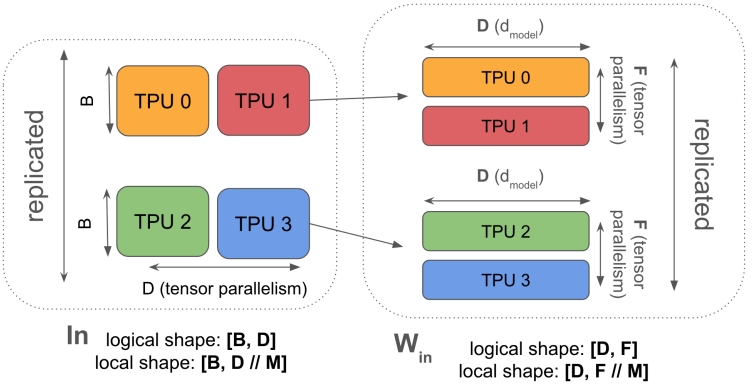


Figure 5.4: An example of basic tensor parallelism. Since we're only sharding our activations over Y (unlike in FSDP where we shard over X), we replicate our activations over X. Using our standard syntax, this is $\mathbf{A}[B, D_Y] \times \mathbf{B}[D, F_Y] \to \mathbf{C}[B, F_Y]$. Because we're only sharding over one of the contracting dimensions, we typically AllGather the activations $\mathbf{A}$ before the matmul.

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

Algorithm

Algorithm: 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 $\mathbf{In}[B, D_Y] \times W_{in}[D, F_Y] \to \mathbf{Tmp}[B, F_Y]$ and then $\mathbf{Tmp}[B, F_Y] \times W_{out}[F_Y, D] \to \mathbf{Out}[B, D_Y]$. This means

we AllGather **In** at the beginning, and ReduceScatter **Out** at the end, rather than doing an AllReduce.

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

$$T_{\text{math}} = \frac{4 \cdot B \cdot D \cdot F}{Y \cdot C} \tag{5.4}$$

$$T_{\text{comms}} = \frac{2 \cdot 2 \cdot (B \cdot D)}{W_{\text{ici}}} \tag{5.5}$$

$$\text{T} \approx \max\left(\frac{4 \cdot B \cdot D \cdot F}{Y \cdot C}, \frac{2 \cdot 2 \cdot (B \cdot D)}{W_{\text{ici}}}\right) \tag{5.6}$$

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

$$\frac{4 \cdot B \cdot D \cdot F}{Y \cdot C} > \frac{2 \cdot 2 \cdot (B \cdot D)}{W_{\text{ici}}} \tag{5.7}$$

$$\frac{F}{Y \cdot C} > \frac{1}{W_{\text{ici}}} \tag{5.8}$$

$$F > Y \cdot \frac{C}{W_{\text{ici}}} \tag{5.9}$$

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

Key Takeaway

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

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

Let's think about some examples:

- On TPUv5p with LLaMA 3-70B with $D = 8192$, $F \approx 30,000$, we can comfortably do 8-way tensor parallelism, but will be communication bound on 16 way tensor parallelism. The required F for model 8 way model sharding is 20k.

- For Gemma 7B, $F \approx 50k$, so we become communication bound with 19-way tensor parallelism. That means we could likely do 16-way and still see good performance.

5.1.4 Combining FSDP and Tensor Parallelism

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

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

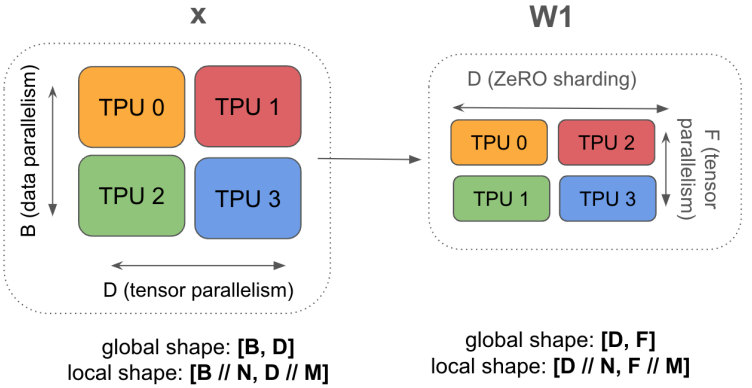


Figure 5.5: A diagram combining FSDP and tensor parallelism. Unlike the other cases, there is no duplication of model parameters.

Algorithm

Algorithm: Combining FSDP and Tensor Parallelism
Forward pass: need to compute Loss[B]

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

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

1. $dOut[B_X, D_Y] = ...$
2. $dOut[B_X, D] = \textbf{AllGather}_Y(dOut[B_X, D_Y])$ *(on critical path)*
3. $dW_{out}[F_Y, D]\ \{U.X\} = \text{Tmp}[B_X, F_Y] *_B dOut[B_X, D]$

4. $\text{dW}_{\text{out}}[\text{F}_\text{Y}, \text{D}_\text{X}] = \textbf{ReduceScatter}_\text{X}(\text{dW}_{\text{out}}[\text{F}_\text{Y}, \text{D}] \{\text{U.X}\})$
5. $\text{W}_{\text{out}}[\text{F}_\text{Y}, \text{D}] = \textbf{AllGather}_\text{X}(\text{W}_{\text{out}}[\text{F}_\text{Y}, \text{D}_\text{X}])$ *(can be done ahead of time)*
6. $\text{dTmp}[\text{B}_\text{X}, \text{F}_\text{Y}] = \text{dOut}[\text{B}_\text{X}, \text{D}] *_\text{D} \text{W}_{\text{out}}[\text{F}_\text{Y}, \text{D}]$ *(can throw away dOut[B, D] here)*
7. $\text{In}[\text{B}_\text{X}, \text{D}] = \textbf{AllGather}_\text{Y}(\text{In}[\text{B}_\text{X}, \text{D}_\text{Y}])$ *(not on critical path + this can be shared with (2) from the previous layer)*
8. $\text{dW}_{\text{in}}[\text{D}, \text{F}_\text{Y}] \{\text{U.X}\} = \text{dTmp}[\text{B}_\text{X}, \text{F}_\text{Y}] *_\text{B} \text{In}[\text{B}_\text{X}, \text{D}]$
9. $\text{dW}_{\text{in}}[\text{D}_\text{X}, \text{F}_\text{Y}] = \textbf{ReduceScatter}_\text{X}(\text{dW}_{\text{in}}[\text{D}, \text{F}_\text{Y}] \{\text{U.X}\})$
10. $\text{W}_{\text{in}}[\text{D}, \text{F}_\text{Y}] = \textbf{AllGather}_\text{X}(\text{W}_{\text{in}}[\text{D}_\text{X}, \text{F}_\text{Y}])$ *(can be done ahead of time)*
11. $\text{dIn}[\text{B}_\text{X}, \text{D}] \{\text{U.Y}\} = \text{dTmp}[\text{B}_\text{X}, \text{F}_\text{Y}] *_\text{F} \text{W}_{\text{in}}[\text{D}, \text{F}_\text{Y}]$ *(needed for previous layers)*
12. $\text{dIn}[\text{B}_\text{X}, \text{D}_\text{Y}] = \textbf{ReduceScatter}_\text{Y}(\text{dIn}[\text{B}_\text{X}, \text{D}] \{\text{U.Y}\})$ *(on critical path)*

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

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

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

Let X be the number of chips dedicated to FSDP and Y be the number of chips dedicated to tensor parallelism. Let N be the total number of chips in our slice with $N = XY$. Let M_X and M_Y be the number of mesh axes over which we do FSDP and TP respectively (these should roughly sum to 3). We'll purely model

the forward pass since it has the most communication per FLOP. Then adding up the comms in the algorithm above, we have

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

And likewise our total FLOPs time is

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

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

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

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

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

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

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

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

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

Key Takeaway

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

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

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

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

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

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

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

Further simplifying, we find that

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

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

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

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

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

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

Key Takeaway

Combining tensor parallelism with FSDP allows us to drop to a per-device batch size of $2550^2/2F$. This lets us handle a batch of as little as 100 per device, which is roughly a factor of eight smaller than we could achieve with just FSDP.

Below we plot the ratio of FLOPs to comms time for mixed FSDP + TP, comparing it both to only tensor parallelism (TP) and

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

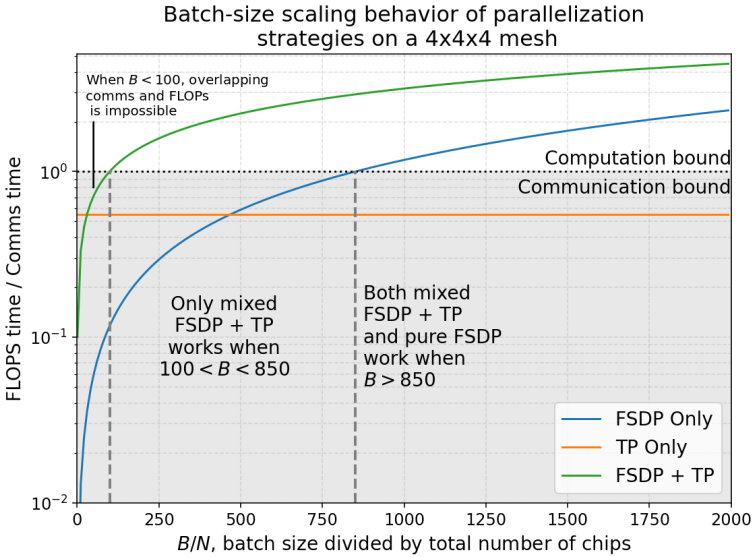


Figure 5.6: Ratio of FLOPs to comms time for optimal mixed FSDP/TP on a TPUv5p 4x4x4 slice with F=30k. As expected, tensor parallelism has a fixed ratio with batch size; ideal mixed FSDP + TP scales with $\sqrt{B}$, and FSDP scales with B. However, in intermediate batch size regimes, only FSDP + TP achieves a ratio greater than unity.

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

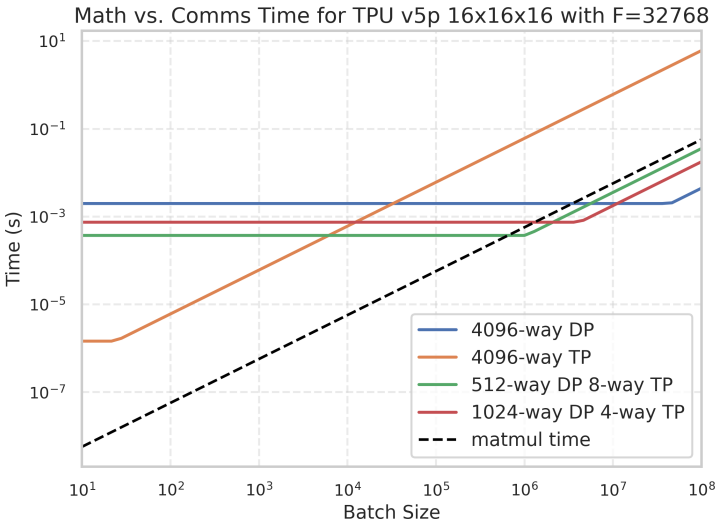


Figure 5.7: Time taken for communication with different parallelism schemes. The black dashed line is the time taken by the matrix multiplication FLOPs, so any curve above this line is comms-bound. We note that all strategies become bandwidth-bound below batch size 6e5, which is in line with our expected $4096 \times 2550^2/(2 \times 8192 \times 4) = 4e5$.

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

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

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 ($W_{\text{in}}[L_Z, D_X, F_Y]$ for pipelining with FSDP and tensor parallelism).

2. Perform the first layer on TPU 0, then copy the resulting activations to TPU 1, and repeat until you get to the last TPU.

3. Compute the loss function and its derivative $\partial L/\partial x_L$.

4. For the last pipeline stage, compute the derivatives $\partial L/\partial W_L$ and $\partial L/\partial x_{L-1}$, then copy $\partial L/\partial x_{L-1}$ to the previous pipeline stage and repeat until you reach TPU 0.

Here is some (working) Python pseudo-code. This pseudocode should run on a Cloud TPU VM. While it's not very efficient or realistic, it gives you a sense how data is being propagated across devices.

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

num_layers = len(jax.devices())

key = jax.random.PRNGKey(0)

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

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

# Assume we have num_layers == num_pipeline_stages
intermediates = [x]
for i in range(num_layers):
```

```
  x = layer_fn(x, weights[i])
  intermediates.append(x)

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

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

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

for i in range(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 micro-batching, 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 DeepSeek-AI et al. [2024] showing their "bubble-free" pipeline schedule:

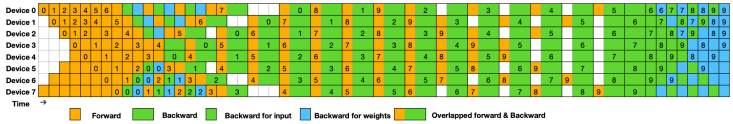


Figure 5.8: The DeepSeek v3 pipeline schedule (from their recent paper). Orange is the forward matmul, green is the dL/dx matmul, and blue is the dL/dW matmul. By prioritizing the backwards dL/dx multiplications, we can avoid "stranding" FLOPs.

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 Across Pods

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

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

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

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

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

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

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

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

- We can do Tensor Parallelism up to above $Y = M_Y \cdot F/2550 \approxeq 11 \cdot M_Y$.
- We can do FSDP so long as $B/N > 2550/M_X$. That means if we want to train with BS=2M and 3 axes of data parallelism, we'd at most be able to use ≈ 2400 chips, roughly a quarter of a TPU v5p pod.
- When we combine FSDP + Tensor Parallelism, become bandwidth-bound when we have $B/N < 2550^2/2 * 30,000 = 108$, so this lets us scale to roughly 18k chips! However, the maximum size of a TPU v5p pod is 8k chips, so beyond that we have to use DCN.

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

Key Takeaway

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

5.2 Takeaways from LLM Training on TPUs

- Increasing parallelism or reducing batch size both tend to make us more communication-bound because they reduce the amount of compute performed per device.
- Up to a reasonable context length (∼32k) we can get away with modeling a Transformer as a stack of MLP blocks and define each of several parallelism schemes by how they shard the two/three main matmuls per layer.
- During training there are 4 main parallelism schemes we consider, each of which has its own bandwidth and compute requirements (data parallelism, FSDP, tensor parallelism).

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

And here are the "formulas" for each method:

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

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

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

- Pure data parallelism is rarely useful because the model and its optimizer state use bytes = 10x parameter count. This means we can rarely fit more than a few billion parameters in memory.

- Data parallelism and FSDP become comms bound when the per-device batch size $< C/W$, 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.

- Tensor parallelism becomes comms bound when $|Y| > F/2550$. **This is around 8–16 way for most models.** This is independent of the batch size.

- Mixed FSDP + tensor parallelism allows us to drop the batch size to as low as $2550^2/2F \approx 100$. This is remarkably low.

- Data parallelism across pods requires a minimum batch size per pod of roughly 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 Some Problems to Work

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

hyperparam	value
L	40
D	5,120
F	13824
N	40
K	40
H	128
V	32,000

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

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

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

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

1. Can we use pure data parallelism? Why or why not?

2. Can we use pure FSDP? Why or why not? With pure FSDP, how much memory will be used per device (assume we do gradient checkpointing only after the 3 big FFW matrices).

3. Can we use mixed FSDP + tensor parallelism? Why or why not? If so, what should X and Y be? How much memory will be stored per device? Using only roofline FLOPs estimates and ignoring attention, how long will each training step take at 40% MFU?

5.4 Appendix

5.4.1 Appendix A: Deriving the backward pass comms

Above, we simplified the Transformer layer forward pass as $\text{Out}[B, D] = \text{In}[B, D] \times_D W_{\text{in}}[D, F] \times_F W_{\text{out}}[F, D]$. 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 $\mathbf{Y} = \mathbf{X} \times \mathbf{A}$:

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

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

1. $\text{dW}_{\text{out}}[F, D] = \text{Tmp}[B, F] *_B \text{dOut}[B, D]$

2. $\text{dTmp}[B, F] = \text{dOut}[B, D] *_D W_{\text{out}}[F, D]$

3. $\text{dW}_{\text{in}} = \text{dTmp}[B, F] *_B \text{Tmp}[B, F]$

4. $\mathrm{dIn}[B, D] = \mathrm{dTmp}[B, F] *_F W_{\mathrm{in}}[D, F]$

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.

Chapter 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 Grattafiori et al. [2024] 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.

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:

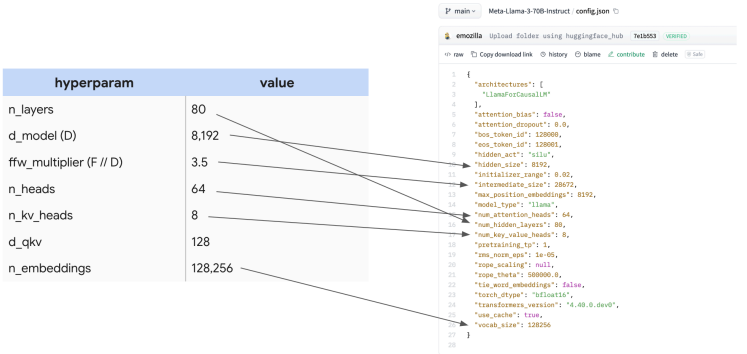


Figure 6.1: LLaMA 3 configuration file showing the architecture parameters and their mapping to the hyperparameters listed above .

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.

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 * * 128 + 2 * 8,192 * 8 n_heads * d_qkv + 2 (for k and v) * * 128) = **12e9** d_model * n_kv_heads * d_qkv]	80 * (2 * 8,192 * 64

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.

Key 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 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.*

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

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? **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.*

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

Key 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.* **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? **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?

Key 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

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.*

2. **LLaMA 405B:**

   (a) Using the LLaMA 3-405B config,[1] 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?

   (b) 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?

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

Chapter 7

All About Transformer Inference

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 $\log p(\text{next token}_i \mid \text{previous tokens})$, 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.

[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.

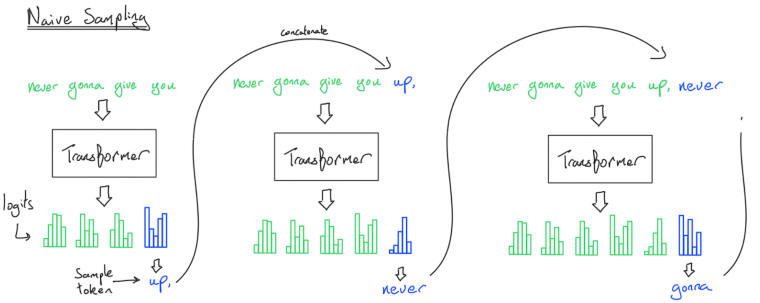


Figure 7.1: naive sampling from a Transformer. The blue logits give us a distribution over the next token that we can sample from. Note that each step re-processes the entire prefix, leading to a $\Theta(n^2)$ runtime for the algorithm.

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!

With this in mind, inference has two key parts:

- **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.

- **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:

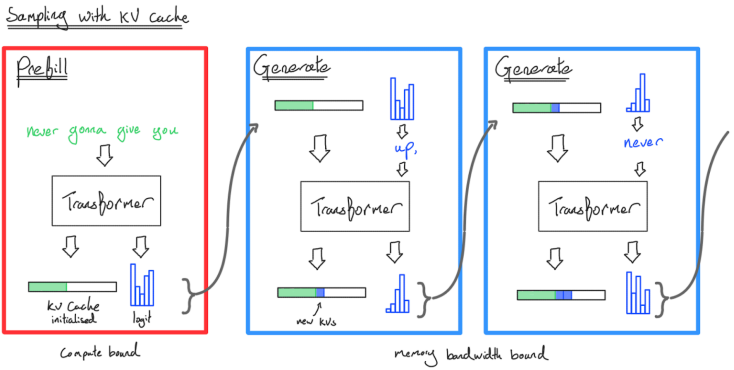


Figure 7.2: diagram of efficient Transformer sampling with a KV cache.

By sampling with a KV cache, we've reduced our time complexity to generate n tokens to $O(n)$ 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 **per chip**), 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](../transformers), the major components of a Transformer forward pass are:

[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.

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 few inner products and some softmax operations, 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 of 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. Within a single accelerator, 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](../roofline) 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 (W_{in} or W_{out}) or one of the smaller attention projections (W_Q, W_K, W_V, W_O). To do this matmul, 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:

$$T_{\text{math}} = \frac{\text{Computation FLOPs}}{\text{Accelerator FLOPs/s}} = \frac{2BDF}{\text{Accelerator FLOPs/s}}$$

$$T_{\text{comms}} = \frac{\text{Communication Bytes}}{\text{Bandwidth Bytes/s}} = \frac{2BD + 2FD + 2BF}{\text{Bandwidth Bytes/s}}$$

A TPU or GPU 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:

$$\frac{2BDF}{2BD + 2DF + 2BF} \geq \frac{\text{Accelerator FLOPs/s}}{\text{Bandwidth Bytes/s}} \underset{\text{TPU v5e}}{=} \frac{1.97E + 14}{8.20E + 11} = 240$$

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

$$\frac{2BDF}{2BD + 2DF + 2BF} \approxeq \frac{2BDF}{2DF} \geq \frac{\text{Accelerator FLOPs/s}}{\text{Bandwidth Bytes/s}}$$
$$\underset{\text{TPU v5e}}{=} \frac{1.97E + 14}{8.20E + 11} \implies B \geq 240 = B_{\text{crit}}$$

If we quantize our weights or use lower precision FLOPs for the matrix multiplication, this critical batch size can change. For instance, if we quantize our weights to int8 or fp8, B_{crit} decreases by 2x. If we do our FLOPs in int8 or fp8, B_{crit} increases by 2x. Thus if we let $\beta =$ bits per param/bits per activation and $\alpha_{\text{hbm}} = C/W_{\text{hbm}}$, our critical batch size is actually $B_{\text{crit}} = \beta\alpha_{\text{hbm}}$.

Key Takeaway

Transformer matmuls are compute-bound *iff* the per-replica **token** batch size is greater than $B_{\text{crit}} = C/W_{\text{hbm}} \cdot$ (bits per param/bits per activation) $= \beta \cdot \alpha_{\text{hbm}}$ For bf16 activations on TPU v5e, this is 240 tokens. For an H100, it is about 280 tokens.

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.

Key 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.**

Key Takeaway

During generation, the total token batch size must be greater than B_{crit} 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 re-

quires 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 bf16, 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.

[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.

5. Write the resulting bf16[B, T, D] tensor back into HBM.

Putting it all together, we get:

Multiheaded Attention

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

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 $\Theta(T)$** . 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:

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

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!**

Key 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 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:

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

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?

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 Pope et al. [2022]):

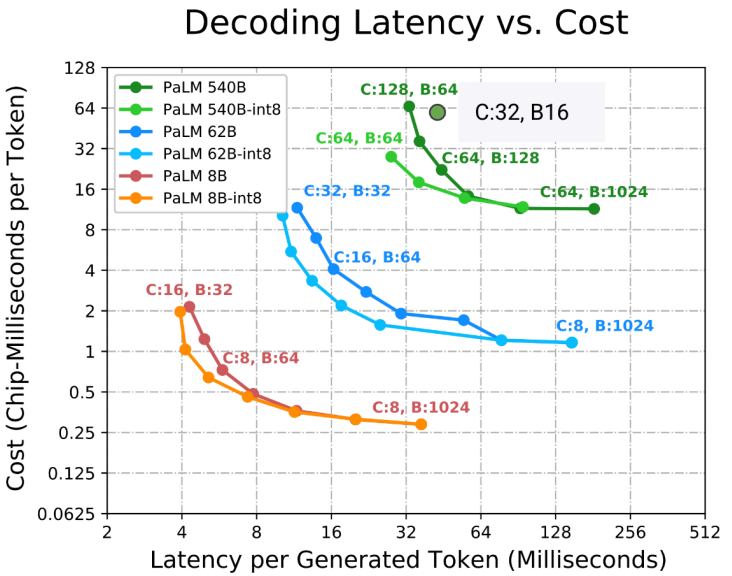


Figure 7.3: Pareto frontier of cost (read: throughput) versus latency for several PaLM models. Note how chip count (C) and batch size (B) moves you along the Pareto frontier, with the exception of the green dot (C:32 B:16 for PaLM 540B) where the available memory prevented the setup from supporting a 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](../applied-inference) explores this in more detail.

Key Takeaway

if you care about generation throughput, use the largest per-chip batch size possible. Any per-chip batch size above 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:

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

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

hyperparam	value
L (num_layers)	40
D (d_model)	5,120
F (ffw_dimension)	13,824
N (num_heads)	40
K (num_kv_heads)	40
H (qkv_dim)	128
V (num_embeddings)	32,000

param	formula	size (bytes)
FFW params	d_model^2 x ffw_mult. x 3 (gelu + out-proj) x n_layers	5,120 x 5,120 x 2.7 x 3 x 40 = **8.5e9**
Vocab params	2 (in & out emb.) x n_emb. x d_model	2 x 32,000 x 5,120 = **0.3e9**
Attn params	[2 (q & out) x d_model x n_heads x d_qkv + 2 (k & v) x d_model x n_kv_heads x d_qkv] x n_layers	(2 x 5,120 x 40 x 128 + 2 x 5,120 x 40 x 128) x 40 = **4.2e9**

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[4] and generate. If we prefill 8k tokens, a single activation only uses around `8,192 x 5,120 x 2 bytes = 80MB` 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

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

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

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

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.

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.

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
Step Time (ms)	4.98	12.13	20.30	36.65	69.33	249.09
Throughput (tok/s)	200.61	659.30	787.99	873.21	923.13	963.53

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](#tricks-for-improving-generation-throughput-and-latency), 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
Step Time (ms)	4.17	5.60	7.23	10.50	17.04	52.99
Throughput (tok/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 (source).

Key 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, 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.

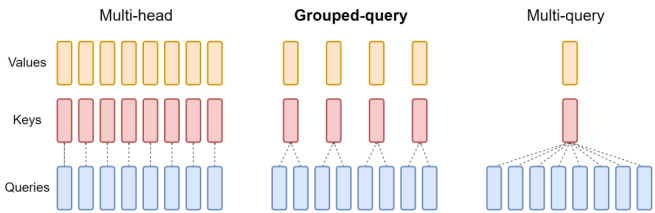


Figure 2: Overview of grouped-query method. Multi-head attention has H query, key, and value heads. Multi-query attention shares single key and value heads across all query heads. Grouped-query attention instead shares single key and value heads for each *group* of query heads, interpolating between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

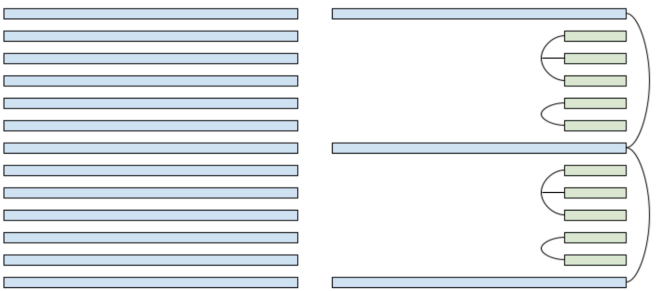


Figure 1. Left: Standard transformer design where every attention is global attention. Right: The attention design in our production model. Blue boxes indicate global attention, green boxes indicate local attention, and curves indicate KV-sharing. For global attention layers, we share KV across multiple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

Figure 7.4: 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 Kwon et al. [2023] 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.

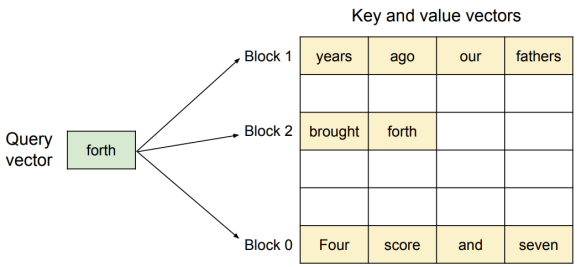


Figure 5. Illustration of the PagedAttention algorithm, where the attention key and values vectors are stored as non-contiguous blocks in the memory.

Figure 7.5: 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.

Key Takeaway

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.

7.3 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 $F/2200$ 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).

Key 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:

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.[5]

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.[6]

3. **No sequence = no sequence sharding.** Good luck sequence sharding.

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

[6]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.

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 (in terms of tokens/sec/chip). More model sharding gives us more HBM to load our weights over, and our FLOPs don't matter.[7] Let's look at how much model parallelism we can do before it becomes the bottleneck.

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

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

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.

[7] 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.

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.

Key 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, 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.

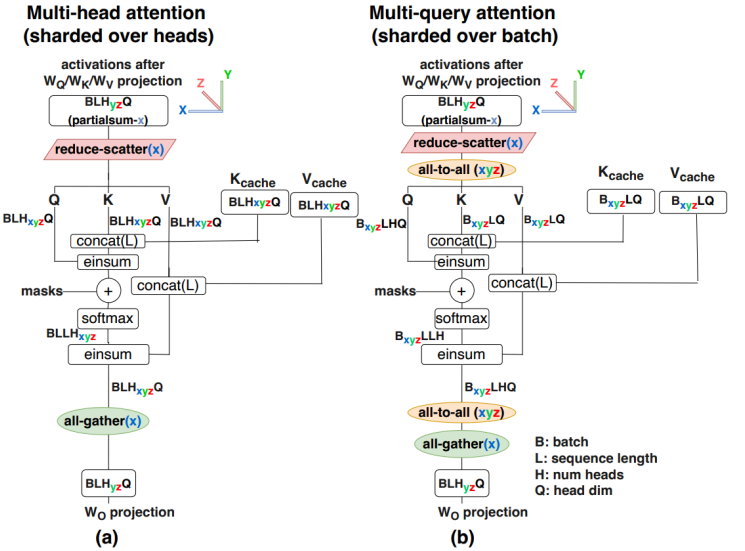


Figure 7.6: comparison of the attention mechanism with (a) Multi head attention with pure model sharding and (b) Multiquery attention with batch sharding of the KV cache. Notice how we need two extra AllToAlls to shift the activations from model sharding to batch sharding, so they can act on the KV caches.

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's the full algorithm!

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:

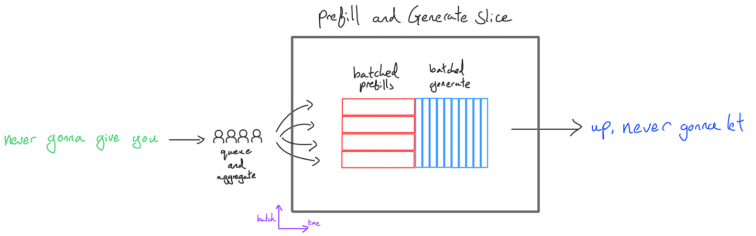


Figure 7.7: In the simplest setup, requests are aggregated, and the server alternates between running a batch of prefills and calling the generate function until completion for all sequences.

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.

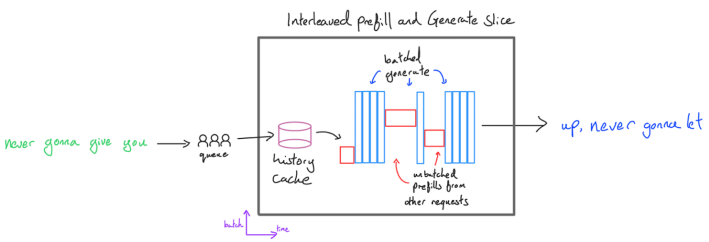
3. **Prefills are padded.** Prefills are padded to the longest sequence and we waste a lot of compute. There are solutions for this, but historically XLA made it quite difficult to skip

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).

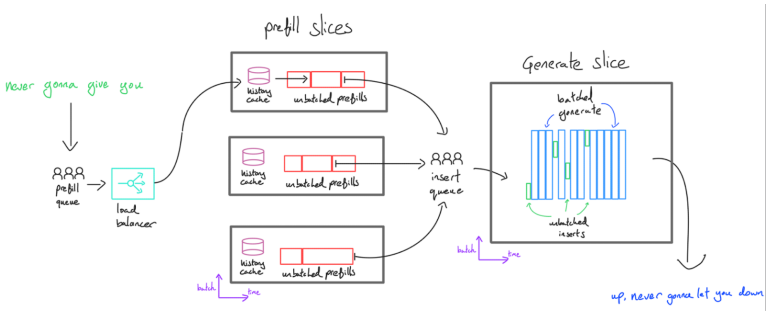
A slightly better approach involves performing prefill at batch size 1 (where it is compute-bound but has reasonable latency) but batch multiple requests together during generation:



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.

Key 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 optimize 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.

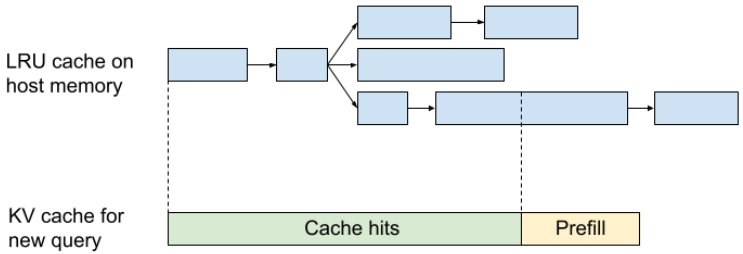


Figure 7.8: KV prefix cache implemented as an LRU trie. We can avoid duplicating KV memory by sharing prefixes.

7.4.3 Let's look at an implementation: JetStream

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

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
L (num_layers)	64
D (d_model)	4,096
F (ffw_dimension)	16,384
N (num_heads)	32
K (num_kv_heads)	8
H (qkv_dim)	256
V (num_embeddings)	32,128

Question 1: How many parameters does the above model have? How large are its KV caches per token in int8? *You can assume we share the input and output projection matrices.*

Question 2: 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 and want to support 128k sequences? What if we dropped the number of KV heads to 1?

Question 3: How long does it take to load all the parameters into the MXU from HBM assuming they're fully sharded on a TPU v5e 4x4 slice? Assume int8 parameters. *This is a good lower bound on the per-step latency.*

Question 4: Let's say we want to serve this model on a TPUv5e 4x4 slice using int8 FLOPs and parameters/activations. How would we shard it for both prefill and decode? *Hint: maybe answer these questions first:*

1. What does ICI look like on a 4x4?
2. What's the roofline bound on tensor parallelism?
3. How can we shard the KV caches?

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

Question 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 total and activated parameters does it have? *Activated means used by any given token.*
2. What batch size is needed to become FLOPs bound on TPU v5e?
3. How large are its KV caches per token?
4. How many FLOPs are involved in a forward pass with T tokens?

Question 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?

Question 7 [2D model sharding]: Here we'll work through the math of what the ESTI paper 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. $\text{In[B, D}_X\text{]} = \mathbf{AllGather}_{YZ}(\text{In[B, D}_{XYZ}\text{]})$
2. $\text{Tmp[B, F}_{YZ}\text{] \{U.X\}} = \text{In[B, D}_X\text{]} *_D \text{W}_{\text{in}}\text{[D}_X\text{, F}_{YZ}\text{]}$
3. $\text{Tmp[B, F}_{YZ}\text{]} = \mathbf{AllReduce}_X(\text{Tmp[B, F}_{YZ}\text{] \{U.X\}})$
4. $\text{Out[B, D}_X\text{] \{U.YZ\}} = \text{Tmp[B, F}_{YZ}\text{]} *_F \text{W2[F}_{YZ}\text{, D}_X\text{]}$
5. $\text{Out[B, D}_{XYZ}\text{]} = \mathbf{ReduceScatter}_{YZ}(\text{Out[B, D}_X\text{] \{U.YZ\}})$

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

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

$$\begin{aligned} T_{\text{2D comms}} &= \frac{2BD}{2X \cdot W_{\text{ici}}} + \frac{4BF}{YZ \cdot W_{\text{ici}}} + \frac{2BD}{2X \cdot W_{\text{ici}}} \\ &= \frac{2BD}{X \cdot W_{\text{ici}}} + \frac{4BF}{YZ \cdot W_{\text{ici}}} \end{aligned}$$

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

$$\begin{aligned} T_{\text{2D comms}} &= \frac{2B}{W_{\text{ici}}}\left(\frac{D}{X} + \frac{8D}{YZ}\right) \\ &= \frac{\sqrt{128}BD}{\sqrt{N} \cdot W_{\text{ici}}} \approx \frac{11.3BD}{\sqrt{N} \cdot W_{\text{ici}}} \end{aligned}$$

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

$$N > 32 \cdot \left(\frac{F}{D}\right) \cdot \left(\frac{3}{4}\right)^2$$

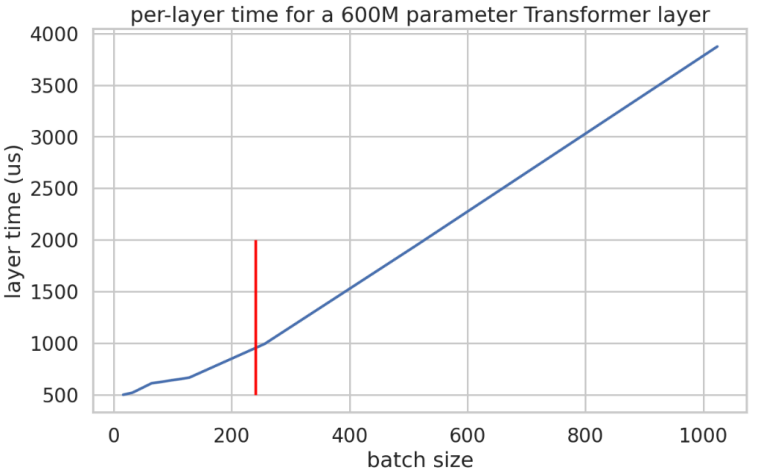
So that tells us if we have more than 81 chips, we're better off using this new scheme. Now this is a slightly weird result because we've historically found ourselves ICI bound at around ~20 way tensor parallelism. But here, even if we're bandwidth-bound, our total communication continues to decrease 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.

7.6 Appendix

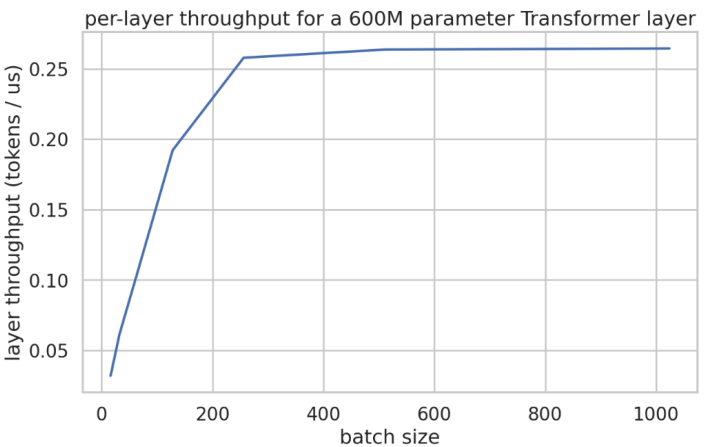
7.6.1 Appendix A: How real is the batch size > 240 rule?

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 this Colab notebook. 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.

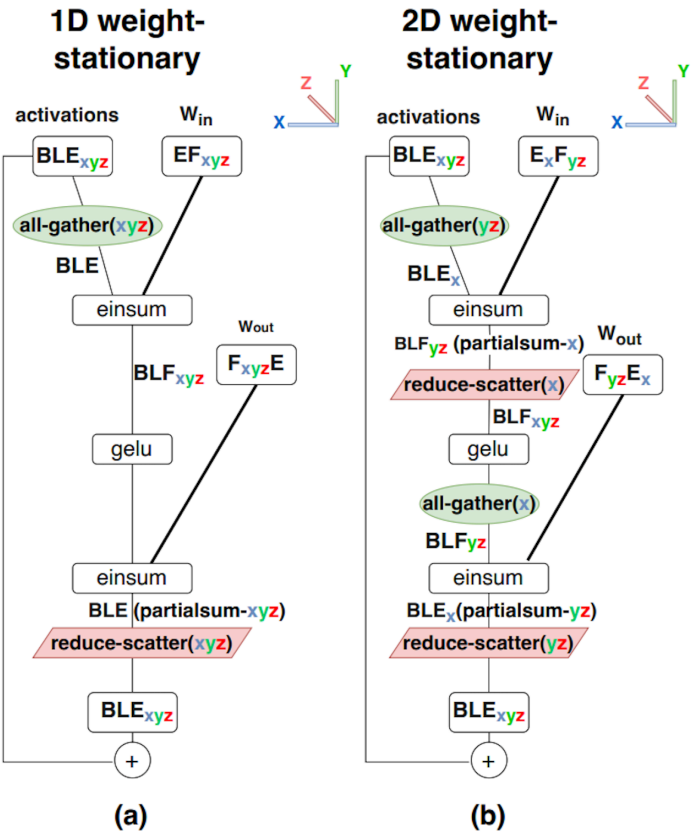


So at least in this model, we do in fact see throughput increase until about BS240 per data parallel shard.

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.

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 ac-

tivations 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.

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 WICI and latency Tmin.

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

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 `4.5e10` 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.

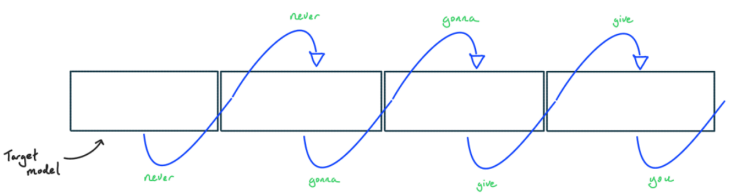
Key Takeaway

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

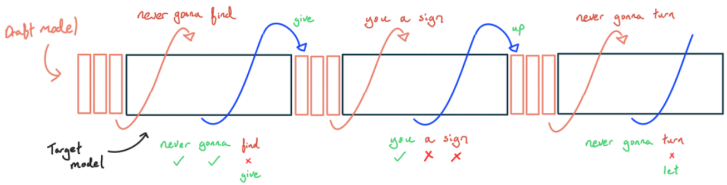
There's a parallel to be drawn here with the compute roofline—we are incurring the fixed cost of some small operations (latency for comms, memory bandwidth for matmuls).

7.6.4 Appendix D: Speculative Sampling

When we *really* care about end to end latency, there is one extra trick we can employ called speculative sampling Leviathan et al. [2022] Chen et al. [2023]. 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. We also share KV cache loads across multiple steps, so **speculative decoding can also be a throughput win for long context.** 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).

Traditionally, speculative decoding relies on the existence of a smaller model with a similar sampling distribution to the target model, e.g. LLaMA-2 2B for LLaMA-2 70B, which often doesn't exist. Even when this is available, the smaller drafter can still be too expensive if the acceptance rate is low. Instead, it can be helpful to embed a drafter within the main model, for instance by adding a dedicated drafter head to one of the later layers of the base model Li et al. [2024] Cai et al. [2024] DeepSeek-AI et al. [2024]. Because this head shares most of its parameters with the main model, it's faster to run and matches the sampling distribution more closely.

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).

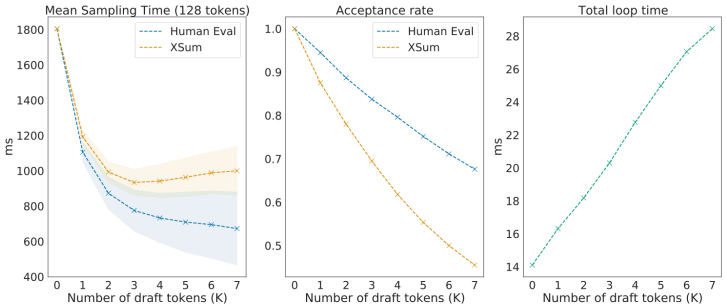


Figure 1 | **Left:** The average time to generate 128 tokens, with standard deviation. Note that as K increases, the overall speedup plateaus or even regresses, with XSum being optimal at $K = 3$. The variance consistently increases with K. **Middle:** The average number of tokens accepted divided by $K + 1$ – 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 7.9: 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 derived this concurrently and have good examples of how this works in practice.

Key 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.

Chapter 8

Serving LLaMA 3-70B on TPUs

This section will look at what it takes to serve LLaMA-3 and how efficiently it can be done. As in the previous "applied" section, try to work out the answers on your own with a pen and paper before looking them up!

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
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

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 as of February 2025):

TPU type	bfloat16 FLOPs/s	Google Cloud USD / hour	FLOPs / $
H100	9.9e14	$10.8	3.3e17
v5p	4.59e14	$4.2	3.9e17
v5e	1.97e14	$1.2	**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.*

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?

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.*

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.

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

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?

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.*

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

Key Takeaway

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.*

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

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?

Key Takeaway

Tip: 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?

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?

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.

8.2 Visualizing the Latency Throughput Tradeoff

Sticking with LLaMA 70B for a second, let's actually look at the latency and throughput for different batch sizes during generation. As we showed in the previous section for PaLM models, this gives us a Pareto frontier for throughput/latency. Let's assume 16-way tensor parallelism since that's a reasonable bound on what we can use while staying compute-bound in the MLP blocks. We'll use a TPU v5e 4x4 topology here. **The slider controls the sequence length so you can see the effect of larger KV caches.**

Interactive Visualization: Latency-Throughput Pareto Frontier

This visualization shows the tradeoff between per-token latency (ms) and throughput (tokens/s/chip) for different batch sizes and sequence lengths. The original online version contains an interactive slider to adjust sequence length dynamically.

Figure 8.1: Latency-throughput Pareto frontier for LLaMA 3-70B on TPU v5e 4x4 with 16-way tensor parallelism. Higher throughput requires accepting higher latency, with batch size as the primary control parameter.

- **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.

Interactive Visualization: Latency Breakdown by Component

This visualization decomposes per-step latency into three components: parameter loading (HBM bandwidth for weights), KV cache loading (attention bandwidth), and compute (FLOPs time). The original online version includes an interactive slider to explore different sequence lengths.

Figure 8.2: Breakdown of latency sources for LLaMA 3-70B showing parameter loading, KV cache loading, and compute time across different batch sizes. KV loading dominates at long context lengths.

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.**

Key Takeaway

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 re-

ducing KV cache size is for generation throughput. Also note just how dramatic the latency/throughput tradeoff remains here.

8.3 Worked Problems

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

Question 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.*

Question 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?

Question 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?

Chapter 9

How to Profile TPU Programs

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, 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[1]:

```
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 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,

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

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. 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[2].

```
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 tensorboard-plugin-profile
# %load_ext tensorboard
# %tensorboard --logdir=/tmp/tensorboard
#
# or externally with
#
# > tensorboard --logdir=/tmp/tensorboard
#
```

[2]See https://docs.jax.dev/en/latest/profiling.html#tensorboard-profiling for a step-by-step guide.

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

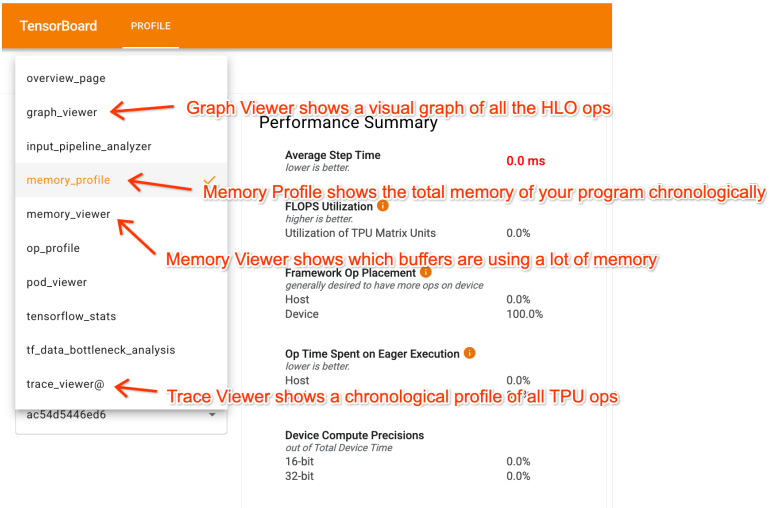


Figure 9.1: Overview of the TensorBoard profiler interface showing the main tabs and functionality available for analyzing TPU program performance.

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, a Perfetto link[3]

[3] https://ui.perfetto.dev/#!/?s=fa9f13b487bde622707c1a503f9227c34594760a

contains at least the Trace Viewer component for a simple Transformer.

An interactive Colab[4] 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.

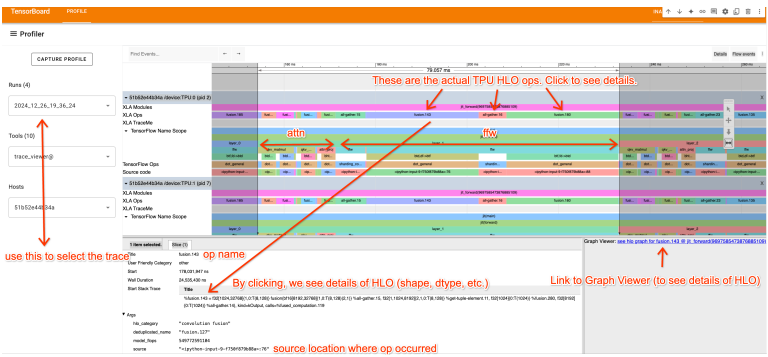


Figure 9.2: The Trace Viewer showing a chronological timeline of operations for a simple Transformer. The top row shows XLA operations (HLO names), while lower rows show approximate traces from JAX scope annotations.

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 approx-

[4] https://colab.research.google.com/drive/1_6krERgtolH7hbUIo7ewAMLlbA4fqEF8?usp=sharing

imate 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.

Key Takeaway

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:

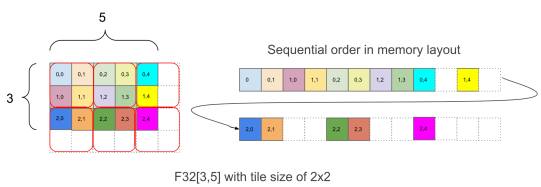


Figure 9.3: Memory layout for `f32[3,5]{1,0:T(2,2)}` showing how the 2D array is tiled in physical memory. The tiling `T(2,2)` means the array is laid out in 2×2 chunks, with row-major ordering within each tile.

Within `{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, `bf16[32,32,8192]{2,1,0:T(8,128)(2,1)S(1)}`, 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 `bf16[4,8]{1,0,T(2,4)(2,1)}` (colors are (2,4) tiles, red boxes are (2,1) tiles):

Figure 9.4: Nested tiling example for `bf16[4,8]{1,0,T(2,4)(2,1)}` showing outer (2,4) tiles in different colors and inner (2,1) tiles outlined in red.

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.[5]

[5]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"

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:

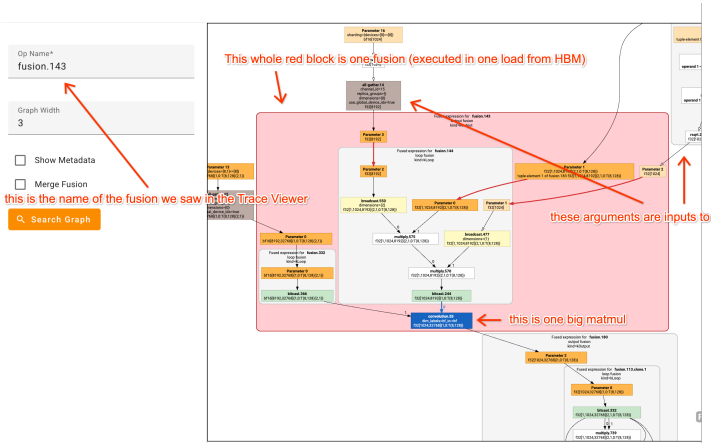


Figure 9.5: The XLA Graph Viewer showing the dataflow structure of a complex fusion operation. Each box represents an operation, with arrows showing data dependencies between operations.

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.

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.

9.2.4 Looking at a real(ish) example profile

An example Colab[6] has an example profile for a fake Transformer.

A Perfetto link[7] to at least see the Trace Viewer if you're in a hurry.

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.

Figure 9.6: Full profiler trace for a fake Transformer showing repeated layer patterns with attention and feedforward components clearly visible.

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:

[6] See: `colab.research.google.com/drive/1_6krERgtolH7hbUIo7ewAMLlbA4fqEF8`

[7] See: `ui.perfetto.dev/#!/?s=fa9f13b487bde622707c1a503f9227c34594760a`

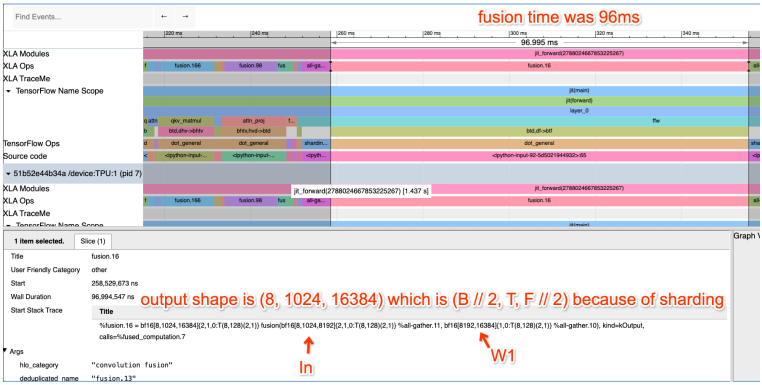


Figure 9.7: Zoomed view of the feedforward (FFW) block showing the up-projection fusion operation and subsequent operations.

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]` $\times$ $\mathbf{W}_{\text{in}}$: `bf16[8192, 32768]` $\rightarrow$ **Tmp**: `bf16[32, 1024, 32768]`

How long do we expect this to take? First of all, our batch size per data parallel shard is $8 \times 1024 = 8192$, 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 $2 \times 32 \times 1024 \times 8192 \times 32768/(23e12 \times 8) = 95.6$ms 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):

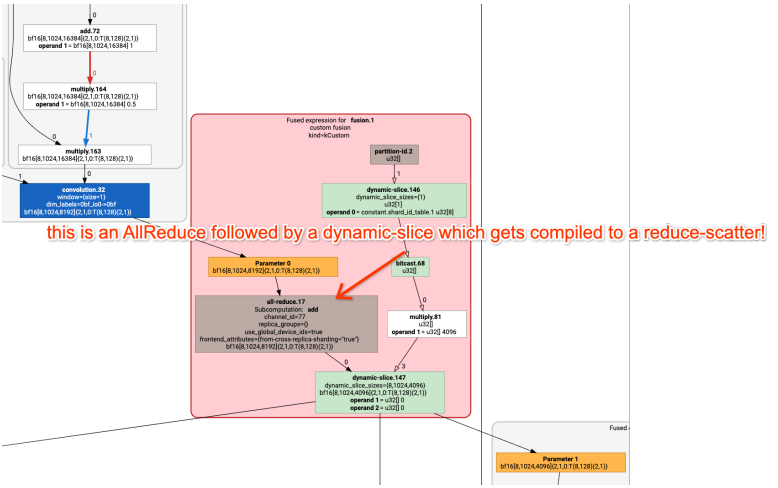


Figure 9.8: Graph view of the reduce-scatter operation showing how data is aggregated and distributed across TPU cores.

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 $2 \times 32 \times 1024 \times 8192$ with the batch axis sharded 4 ways, so each shard is $2 \times 8 \times 1024 \times 8192 = 134$MB. 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:

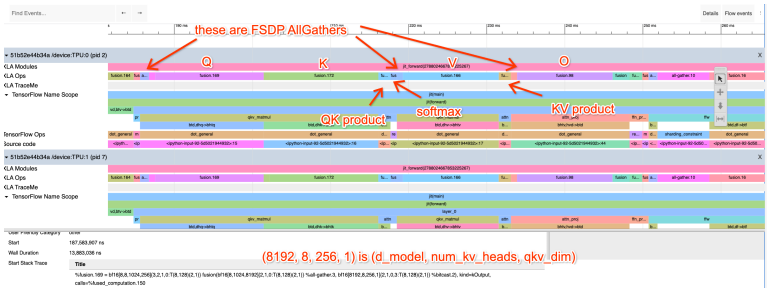


Figure 9.9: Profile of the attention mechanism showing Q, K, V projections and attention computation operations.

I've clicked on the Q projection op, which uses a matrix W_Q of shape $[d_{\text{model}} = 8192, n_{\text{heads}} = 32, d_{\text{qkv}} = 256]$. 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.

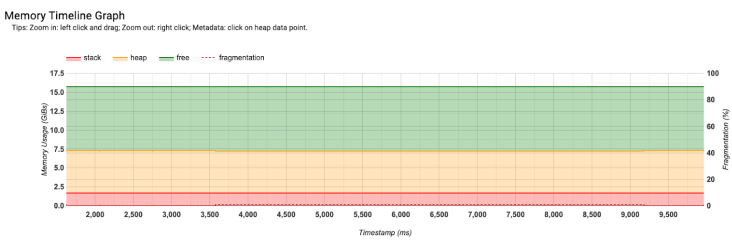


Figure 9.10: Memory Profile showing memory allocation over time, with approximately 7.5GB allocated to model parameters and 10GB free memory remaining.

9.3 Worked Problems

Question 1: take a look at this Colab/profile[8] and figure out what looks suspicious and what's going on here. Can you tell me exactly what computations are happening and what each operation is doing? What are the true shapes of each matrix involved and how are they sharded? *Try looking at the profile first without reading the code.*

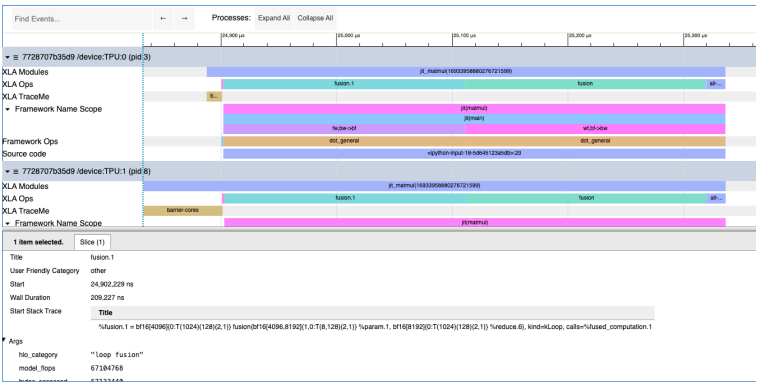


Figure 9.11: Profile showing a suspicious pattern with reduce operations and all-reduce communications. The challenge is to identify the computation structure and sharding strategy from the profile alone.

Question 2: The Transformer Colab from earlier[9] implements a simple mock Transformer. Follow the instructions in the Colab and get a benchmark of the naive Transformer with GSPMD partitioning. How long does each part take? How long should it take? What sharding is being used. Try fixing the sharding! *Hint: use*

[8] https://colab.research.google.com/drive/1LfLO3OTr-_MWFPxUN36KJ3cqH0BcAoli?usp=sharing

[9] https://colab.research.google.com/drive/1_6krERgtolH7hbUIo7ewAMLlbA4fqEF8?usp=sharing

`jax.lax.with_sharding_constraints` to constrain the behavior. With this fix, what's the best MXU you can get?

For reference, the initial version gets roughly 184ms / layer and the optimized profile gets 67 ms / layer. Once you've done this, try staring at the profile and see if you can answer these questions purely from the profile:

- What sharding strategy is this?
- What is the batch size, d_{model}, d_{ff}?
- What fraction of time is spent on attention vs. the MLP block?
- What fraction of time should be spent on each op at the roofline?

Note: since this problem was written, the XLA compiler has gotten better. The initial version is now at roughly 90ms / layer and the optimized profile is only about 10ms / layer better (80 ms / layer). Still, it's worth playing with and seeing if you can do better.

Chapter 10

Programming TPUs in JAX

10.1 How Does Parallelism Work in JAX?

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

1. **Compiler, take the wheel!** Let the XLA compiler automatically partition arrays and decide what communication to add to facilitate a given program. This lets you take a program that runs on a single device and automatically run it on thousands without changing anything.

2. **JAX, take the wheel!** Automatic parallelism is great, but sometimes the compiler does something crazy. Explicit sharding lets you write single-device code like usual, but have JAX handle sharding propagation (not the compiler). This means JAX can ask you for clarification when it's unclear what you want.

3. **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 explicit about exactly what communication you intend to run.

Mode	View?	Explicit shard-ing?	Explicit Collec-tives?
Auto	Global	×	×
Explicit	Global	✓	×
Manual	Per-device	✓	✓

Correspondingly, JAX provides APIs for each of these modes:

1. `jax.jit` (with `Auto` mesh axes) lets you take any existing JAX function and call it with sharded inputs. JAX then uses XLA's Shardy compiler which automatically parallelizes the program. XLA will add communication for you (AllGathers, ReduceScatters, AllReduces, etc.) when needed to facilitate existing operations. While it isn't perfect, it usually does a decent job at automatically scaling your program to any number of chips without code changes.

2. `jax.jit` with `Explicit` mesh axes looks similar to (1), but lets JAX handle the sharding propagation instead of XLA. That means the sharding of an array is actually part of the JAX type system, and JAX can error out when it detects ambiguous communication and lets the user resolve it.

3. `jax.shard_map` is the more manual 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 Auto sharding mode

`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

# Running on an TPU v5e 4x2. This assigns names to
    the two physical axes of the hardware.
mesh = jax.make_mesh(axis_shapes=(4, 2), axis_names=(
    'X', 'Y'))

# This tells JAX to use this mesh for all operations,
     so you can just specify the PartitionSpec P.
jax.set_mesh(mesh)

# We create a matrix W and input activations In
    sharded across our devices.
In = jnp.zeros((8, 2048), dtype=jnp.bfloat16, device=
    jax.NamedSharding(mesh, jax.P('X', 'Y')))
W = jnp.zeros((2048, 8192), dtype=jnp.bfloat16,
    device=jax.NamedSharding(mesh, jax.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=jax
    .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[1]. 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 $W[D_Y, F]$ 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)`, the 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. $Out[B_X, F] = \mathbf{AllReduce}(Out[B_X, F]\{U_Y\})$

`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[2,8192]{1,0:T(4,128)(2,1)S(1)} fusion(bf16
    [2,1024]{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[2,8192]{1,0:T(4,128)(2,1)} AllReduce
    (bf16[2,8192]{1,0:T(4,128)(2,1)S(1)} %fusion)
```

[1]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.

We can see the matmul (the fusion) and the AllReduce above. Pay particular attention to the shapes. `bf16[2, 1024]` is a local view of the activations, since our `batch_size=8` is split across 4 devices and our `d_model=2048` is likewise split 2 ways.

This is pretty magical! No matter how complicated our program is, Shardy and jit will attempt to find shardings for all the intermediate activations and add communication as needed. With that said, Shardy 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 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

mesh = jax.make_mesh((4, 2), ('X', 'Y'))

def matmul(x, Win, Wout):
  hidden = jnp.einsum('bd,df->bf', x, Win)
  hidden = jax.lax.with_sharding_constraint(hidden,
      jax.P('x', 'y'))
  return jnp.einsum('bf,df->bd', hidden, Wout)
```

This makes up like 60% of JAX parallel programming in the automatic partitioning world where you control the intermediate shardings via `jax.lax.with_sharding_constraint`. But "compiler tickling" is famously not a fun programming model. You could annotate every intermediate variable and still not know if you'll get the right outcome. Instead, what if JAX itself could handle and control sharding propagation?

10.1.2 Explicit sharding mode

Explicit sharding (or "sharding in types") looks a lot like automatic sharding, but sharding propagation happens at the JAX level! Each JAX operation has a sharding rule that takes the shardings of the op's arguments and produces a sharding for the op's result. You can see the resulting sharding using `jax.typeof`:

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

# Running on 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'),
                                          axis_types=(
                                              shd.
                                              AxisType.
                                              Explicit,
                                              shd.
                                              AxisType.
                                              Explicit))

# This tells JAX to use this mesh for all operations,
     so you can just specify the PartitionSpec P.
jax.set_mesh(mesh)

x = jax.device_put(np.arange(16).reshape(8, 2), P('X'
    , 'Y'))

@jax.jit
def f(x):
  print(jax.typeof(x))  # bfloat16[8@X,2@Y]
  out = x * 2
  print(jax.typeof(out))  # bfloat16[8@X,2@Y]
  return out

f(x)
```

As you can see, JAX propagated the sharding from input (`x`) to output (`x`) which are inspectable at trace-time via `jax.typeof`. For most operations these rules are simple and obvious because there's only one reasonable choice (e.g. elementwise ops retain the same sharding). But for some operations it's ambiguous how to shard the result in which case JAX throws a trace-time error and we ask the programmer to provide an `out_sharding` argument explicitly (e.g. `jnp.einsum`, `jnp.reshape`, etc). Let's see another example where you have conflicts:

```
# We create a matrix W and input activations In
    sharded across our devices.
In = jnp.zeros((8, 2048), dtype=jnp.bfloat16,
    out_sharding=jax.P('X', 'Y'))
W = jnp.zeros((2048, 8192), dtype=jnp.bfloat16,
    out_sharding=jax.P('Y', None))

@jax.jit
def matmul_square(In, W):
  print(jax.typeof(In))  # bfloat16[8@X, 2048@Y]
  print(jax.typeof(W))  # bfloat16[2048@Y, 8192]
  return jnp.einsum('bd,df->bf', jnp.square(In), W)

matmul_square(In, W)  # This will error
```

This code errors with `Contracting dimensions are sharded and it is ambiguous how the output should be sharded. Please specify the output sharding via the `out_sharding` parameter. Got lhs_contracting_spec=('Y',) and rhs_contracting_spec=('Y',)`

This is awesome because how the output of einsum should be sharded is ambiguous. The output sharding can be:

- `P('X', 'Y')` which will induce a reduce-scatter or
- `P('X', None)` which will induce an all-reduce

Unlike Auto mode, explicit mode errors out when it detects ambiguous communication and requires the users to resolve it. So here you can do:

```
@jax.jit
def matmul_square(In, W):
  return jnp.einsum('bd,df->bf', jnp.square(In), W,
      out_sharding=P('X', 'Y'))

out = matmul_square(In, W)
print(jax.typeof(out))  # bfloat16[8@X,8192@Y]
```

Auto mode and Explicit mode can be composed via `jax.sharding.auto_axes` and `jax.sharding.explicit_axes` APIs. This is a great doc to read[2] for more information.

10.1.3 shard_map: explicit parallelism control over a program

While Shardy is the "compiler take the wheel" mode, jax `shard_map`[3] 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

mesh = jax.make_mesh((2, 4), ('x', 'y'), (shd.
    AxisType.Explicit, shd.AxisType.Explicit))
jax.set_mesh(mesh)
```

[2]See JAX explicit sharding notebook at: `docs.jax.dev/en/latest/notebooks/explicit-sharding.html`

[3]See JAX shard_map documentation at: `jax.readthedocs.io/en/latest/jep/14273-shard-map.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 jax; jax.config.update('jax_num_cpu_devices', 8)`

```
x = jnp.arange(0, 512, dtype=jnp.int32, out_sharding=
    P(('x', 'y')))

# This function will operate on 1/8th of the array.
@jax.shard_map(in_specs=P(('x', 'y')), out_specs=P())
def slice_and_average(x):
  assert x.shape == (512 // 8,)
  return jax.lax.pmean(x[:4], axis_name=('x', 'y'))

out = slice_and_average(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.

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. $A[B_X, D_Y] * W[D, F_Y] \rightarrow Out[B_X, F_Y]$. 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

mesh = jax.make_mesh(axis_shapes=(2, 4), axis_names=(
    'X', 'Y'),
                                          axis_types=(
                                              shd.
                                              AxisType.
                                              Explicit,
                                              shd.
                                              AxisType.
                                              Explicit))
jax.set_mesh(mesh)

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, jax.P('X', 'Y'))
W = jax.device_put(W, jax.P(None, 'Y'))
```

[5] Wang, Z., Yi, X., Zhong, L., and Yang, C. (2023). Overlap Communication with Dependent Computation via Decomposition in Large Deep Learning Models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (pp. 93–106). ACM.

```
@functools.partial(jax.jit, out_shardings=jax.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.axis_size('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

  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 = jax.lax.pvary(accum, ('X', 'Y'))
  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(jax.shard_map(
  collective_matmul_allgather_lhs_contracting,
  in_specs=(jax.P('X', 'Y'), jax.P(None, 'Y')),
      out_specs=jax.P('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 the profile[6] with the default jit matmul which takes 311us with a big blocking AllGather at the beginning:

[6] See profile: `imgur.com/a/e9I6SrM`

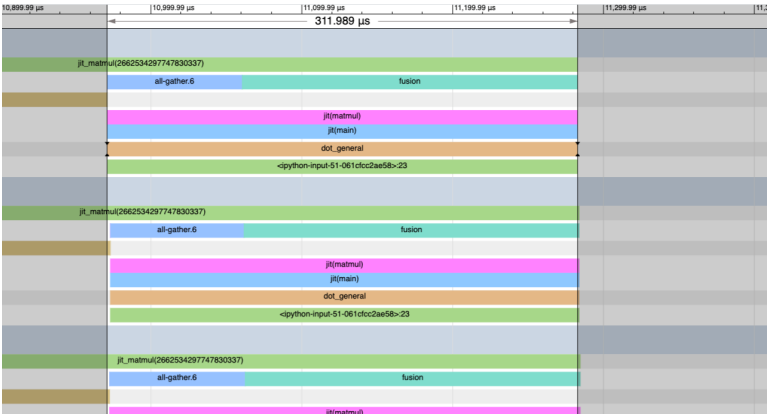


Figure 10.1: Profile showing non-overlapped computation with blocking AllGather taking 311us. The large communication operation at the start blocks all computation.

And here's the version[7] 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.

[7]See profile: `imgur.com/a/21iy0Sv`

Figure 10.2: Profile showing overlapped computation taking 244us. Communication is interleaved with computation, eliminating the blocking AllGather and achieving much higher FLOPs utilization.

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, see the documentation.[9]

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.

[8] See unsharded baseline profile: `imgur.com/a/i3gNKfq`

[9] JAX docs: `jax.readthedocs.io/en/latest/notebooks/shard_map.html`

You can use a public Colab with TPUv2-8. From now on, we'll assume you have N devices available.

Problem 1: Let **A** be an array of activations of shape float32$[S_X, D_Y]$ with `X * Y = N`. Do the following:

1. Write a function in JAX that computes the average within each `(X, Y)` shard, i.e. it returns an array of size `[X, Y]` where `arr[i, j]` is the average over shard `(i, j)`. Do this with both `jax.jit` and `shard_map`. Profile each and see how long they took. Was there any communication added? *Hint: there shouldn't be, but sometimes XLA adds it anyway.*

2. Write a function in JAX that returns `roll(x, shift, axis=0) - x` for some shift **within each shard X**. I'm not enough of a masochist to make you do this in `jax.jit`, so just do this with `shard_map`.

Problem 2: Here we'll make a basic "mixture of experts" model together. Let **W**: float32$[E_X, D, F_Y]$ be a set of E "expert" matrices. Let **A**: float32$[S_X, D_Y]$ (our activations) and let **B** be a set of "routing assignments" where `B[i]` is an integer in the range `[0, E)` telling us which matrix we want to process that activation. We want to write a function in JAX that returns `Out[i] = W[B[i]] @ A[i]`.

1. Let's start by ignoring sharding altogether. Make all of these tensors small enough so they fit in one device. Write a local implementation of this function. *Make sure you don't materialize an array of shape `[S, D, F]`! Hint: try sorting the tokens into a new buffer of shape `[E, S, D]` with some attention to masking (why do we need the second dimension to have size S?).*

2. If you just `jax.jit` the above method, something will happen. Profile this and see what communication it decided to do. How long does it take?

3. One problem you'll notice with the above is that it likely gathers the full set of activations **A** locally, i.e. $\text{AllGather}_X([S_X, D_Y])$, Not only is this expensive communication-wise, it's also incredibly expensive memory-wise if we can't fit the full set of activations locally. Implement the above using `shard_map` and explicit communication.

   (a) For a first pass, it might be easiest to use a `jax.lax.all_gather` and reorder as in (a).

   (b) For a second pass, try to avoid materializing any array of size `[E, S, D]`, i.e. try to perform the computation in a ragged fashion using a `jax.lax.all_to_all` inside a `jax.lax.while_loop`. This way, you can avoid materializing the full activations and wasting compute on padding. How much faster is this than your original implementation?

4. Most MoEs route to multiple (k) experts and then average the result. Refactor the above to implement this. Let **B**: int32[S, k] in this case for the k experts to route to.

Problem 3: The collective matmul example above is actually super relevant for real LLMs. Let's tweak the example to do the full Transformer stack.

1. As an exercise, let's start by implementing an AllReduce collective matmul, i.e. $\text{A}[B_X, D_Y] *_D \text{W}[D_Y, \text{F}] \rightarrow \text{Out}[B_X, \text{F}]$. 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.*

2. The complement to the AllReduce collective matmul above is a ReduceScatter collective matmul, as in $\text{Tmp}[B_X, F_Y]$

$*_F$ W2[F_Y, D] $\rightarrow$ 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.*

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] $\rightarrow$ Out[B_X, D_Y] with overlapped communication.[10] How much faster is this than a `jax.jit` implementation?

Problem 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 $W_{in} \cdot W_{out}$ first because of a non-linearity we've omitted here.

Chapter 11

Conclusions and Further Reading

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!

Before you go, you might also enjoy reading the new Section 12 on NVIDIA GPUs!

11.2 Further Reading

There is a bunch of related writing, including the following:

- **TPU Deep Dive**[1]: a wonderful in-depth look at the TPU architecture in the spirit of this book.
- **Making Deep Learning Go Brrrr From First Principles**[2]: a more GPU and PyTorch-focused tutorial on LLM rooflines and performance engineering.
- **Writing TPU Kernels with Pallas**[3]: 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.

[1] See: `henryhmko.github.io/posts/tpu/tpu.html`
[2] See: `horace.io/brrr_intro.html`
[3] See: `jax.readthedocs.io/en/latest/pallas/tpu/details.html`

- **How to Optimize a CUDA Matmul Kernel for cuBLAS-like Performance: a Worklog**[4]: 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**[5]: 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**[6]: 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**[7]: a detailed paper on the mathematics of Transformer inference. This is the inspiration for a lot of this document.

- **Huggingface Ultra-Scale Playbook**[8]: 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**[9]: a blog with many of the same ideas as this book and some excellent illustrations.

- **Stanford CS336 Slides and Videos**[10]: a fantastic Stanford course covering many details of LLM training and serv-

[4] See: `siboehm.com/articles/22/CUDA-MMM`

[5] See JAX documentation at: `jax.readthedocs.io/en/latest/notebooks/Distributed_arrays_and_automatic_parallelization.html`

[6] See: `github.com/rwitten/HighPerfLLMs2024`

[7] arXiv preprint: `arxiv.org/abs/2211.05102`

[8] See: `huggingface.co/spaces/nanotron/ultrascale-playbook`

[9] See: `kipp.ly/transformer-inference-arithmetic/`

[10] See: `stanford-cs336.github.io/spring2025/index.html#coursework`

ing, with some useful exercises. Assignments 1 and 2 are particularly relevant.

- **Stas Bekman's ML Engineering Handbook**[11]: a highly practical guide to ML infrastructure, covering topics not addressed in this book like how to negotiate with cloud providers, cluster management, and empirical measurements of GPU throughput.

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]

[11] See: `github.com/stas00/ml-engineering`
[12] See: `github.com/jax-ml/scaling-book`

Chapter 12

How to Think About GPUs

12.1 What Is a GPU?

A modern ML GPU (e.g. H100, B200) is basically a bunch of compute cores that specialize in matrix multiplication (called **Streaming Multiprocessors** or **SMs**) connected to a stick of fast memory (called **HBM**). Here's a diagram:

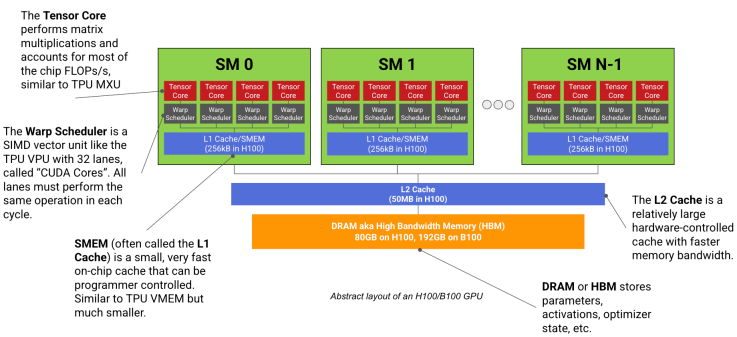


Figure 12.1: A diagram showing the abstract layout of an H100 or B200 GPU. An H100 has 132 SMs while a B200 has 148. We use the term 'Warp Scheduler' somewhat broadly to describe a set of 32 CUDA SIMD cores *and* the scheduler that dispatches work to them. Note how much this looks like a TPU!

Each SM, like a TPU's Tensor Core, has a dedicated matrix multiplication core (unfortunately also called a **Tensor Core**[1]), a vector arithmetic unit (called a **Warp Scheduler**[2]), and a fast on-chip cache (called **SMEM**). Unlike a TPU, which has at most 2 independent "Tensor Cores", a modern GPU has more than 100 SMs (132 on an H100). Each of these SMs is much less powerful than a TPU Tensor Core but the system overall is more flexible. Each SM is more or less totally independent, so a GPU can do hundreds of separate tasks at once.[3]

Let's take a more detailed view of an H100 SM:

[1]The GPU Tensor Core is the matrix multiplication sub-unit of the SM, while the TPU TensorCore is the umbrella unit that contains the MXU, VPU, and other components.

[2]NVIDIA doesn't have a good name for this, so we use it only as the best of several bad options. The Warp Scheduler is primarily the unit that dispatches work to a set of CUDA cores, but we use it here to describe the control unit and the set of cores it controls.

[3]Although SMs are independent, they are often forced to coordinate for peak performance because they all share a capacity-limited L2 cache.

Figure 12.2: A diagram of an H100 SM (source: `wccftech.com/nvidia-hopper-gh100-gpu-official-5nm-process-worlds-fastest-hpc-chip-80-billion-transistors-hbm3-memory/`) showing the 4 *subpartitions*, each containing a Tensor Core, Warp Scheduler, Register File, and sets of CUDA Cores of different precisions. The 'L1 Data Cache' near the bottom is the 256kB SMEM unit. A B200 looks similar, but adds a substantial amount of Tensor Memory (TMEM) for feeding the bulky Tensor Cores.

Each SM is broken up into 4 identical quadrants, which NVIDIA calls **SM subpartitions**, each containing a Tensor Core, 16k 32-bit registers, and a SIMD/SIMT vector arithmetic

unit called a Warp Scheduler, whose lanes (ALUs) NVIDIA calls **CUDA Cores**. The core component of each partition is arguably the Tensor Core, which performs matrix multiplications and makes up the vast majority of its FLOPs/s, but it's not the only component worth noting.

- **CUDA Cores:** each subpartition contains a set of ALUs called CUDA Cores that do SIMD/SIMT vector arithmetic. Each ALU can generally do 1 arithmetic op each cycle, e.g. f32.add.[4] Each subpartition contains 32 fp32 cores (and a smaller number of int32 and fp64 cores) that all execute the same instruction in each cycle. Like the TPU's VPU, CUDA cores are responsible for ReLUs, pointwise vector operations, and reductions (sums).[5]

- **Tensor Core (TC):** each subpartition has its own Tensor Core, which is a dedicated matrix multiplication unit like a TPU MXU. The Tensor Core represents the vast majority of the GPUs FLOPs/s (e.g. on an H100, we have 990 bf16 TC TFLOP/s compared to just 66 TFLOPs/s from the CUDA cores).

  - 990 bf16 TFLOPs/s with 132 SM running at 1.76GHz means each H100 TC can do 7.5e12/1.76e9/4 $\sim$ 1024 bf16 FLOPs/cycle, roughly an 8x8x8 matmul.[6]

[4]Newer GPUs support FMA (Fused-Multiply Add) instructions which technically do two FLOPs each cycle, a fact NVIDIA uses ruthelessly to double their reported specs.

[5]Historically, before the introduction of the Tensor Core, the CUDA cores were the main component of the GPU and were used for rendering, including ray-triangle intersections and shading. On today's gaming GPUs, they still do a bulk of the rendering work, while TensorCores are used for up-sampling (DLSS), which allows the GPU to render at a lower resolution (fewer pixels = less work) and upsample using ML.

[6]NVIDIA doesn't share many TC hardware details, so this is more a guess than definite fact – certainly, it doesn't speak to how the TC is implemented. We know that a V100 can perform 256 FLOPs/TC/cycle. An A100 can do 512, H100 can do 1024, and while the B200 details aren't published, it seems likely it's about 2048 FLOPs/TC/cycle, since

- Like TPUs, GPUs can do lower precision matmuls at higher throughput (e.g. H100 has 2x fp8 FLOPs/s vs. fp16). Low-precision training or serving can be significantly faster.

- Each GPU generation since Volta has increased the TC size over the previous generation (see good article on this at: `semianalysis.com/2025/06/23/nvidia-tensor-core-evolution-from-volta-to-blackwell/`). With B200 the TC has gotten so large it can no longer fit its inputs in SMEM, so B200s introduce a new memory space called TMEM.[7]

CUDA cores are more flexible than a TPU's VPU: GPU CUDA cores (since V100) use what is called a SIMT (*Single Instruction Multiple Threads*) programming model, compared to the TPU's SIMD (*Single Instruction Multiple Data*) model. Like ALUs in a TPU's VPU, CUDA cores within a subpartition must execute the same operation in each cycle (e.g. if one core is adding two floats, then every other CUDA core in the subpartition must also do so). Unlike the VPU, however, each CUDA core (or "thread" in the CUDA programming model) has its own instruction pointer and can be *programmed* independently. When two threads in the same warp are instructed to perform different operations, you effectively do *both* operations, masking out the cores that don't need to perform the divergent operation.

2250e12/(148 * 4 * 1.86e9) is about 2048. Some more details are confirmed at: `forums.developer.nvidia.com/t/how-to-calculate-the-tensor-core- fp16-performance-of-h100/244727`.

[7] In Ampere, the Tensor Core could be fed from a single warp, while in Hopper it requires a full SM (warpgroup) and in Blackwell it's fed from 2 SMs. The matmuls have also become so large in Blackwell that the arguments (specifically, the accumulator) no longer fit into register memory/SMEM, so Blackwell adds TMEM to account for this.

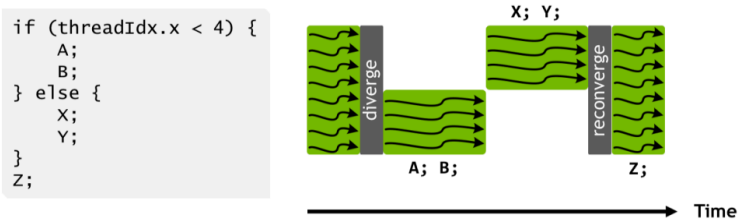


Figure 12.3: An example of warp divergence within a set of threads (source: `images.nvidia.com/ content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`). White spaces indicate stalls of at least some fraction of the physical CUDA cores.

This enables flexible programming at the thread level, but at the cost of silently degrading performance if warps diverge too often. Threads can also be more flexible in what memory they can access; while the VPU can only operate on contiguous blocks of memory, CUDA cores can access individual floats in shared registers and maintain per-thread state.

CUDA core scheduling is also more flexible: SMs run a bit like multi-threaded CPUs, in the sense that they can "schedule" many programs (**warps**) concurrently (up to 64 per SM) but each *Warp Scheduler* only ever executes a single program in each clock cycle.[8] The Warp Scheduler automatically switches between active warps to hide I/O operations like memory loads. TPUs are generally single threaded by comparison.

12.1.1 Memory

Beyond the compute units, GPUs have a hierarchy of memories, the largest being HBM (the main GPU memory), and then a series of smaller caches (L2, L1/SMEM, TMEM, register memory).

- **Registers:** Each subpartition has its own register file con-

[8]Warps scheduled on a given SM are called "resident".

taining 16,384 32-bit words on H100/B200 ($4 \times 16384 \times 4 = 256$kiB per SM) accessible by the CUDA cores.

  - Each CUDA core can only access up to 256 registers at a time, so although we can schedule up to 64 "resident warps" per SM, you can only fit 8 ($256 \times 1024/(4 \times 32 \times 256)$) at a time if each thread uses 256 registers.

- **SMEM (L1 Cache):** each SM has its own 256kB on-chip cache called SMEM, which can either be programmer controlled as "shared memory" or used by the hardware as an on-chip cache. SMEM is used for storing activations and inputs to TC matmuls.

- **L2 Cache:** all SMs share[9] a relatively large 50MB L2 cache used to reduce main memory accesses.

  - This is similar in size to a TPU's VMEM but it's **much** slower and isn't programmer controlled. This leads to a bit of "spooky action at a distance" where the programmer needs to modify memory access patterns to ensure the L2 cache is well used.[10]
  - NVIDIA does not publish the L2 bandwidth for their chips, but it's been measured (see: `chipsandcheese.com/p/nvidias-h100-funny-l2-and-tons-of-bandwidth`) to be about 5.5TB/s. Thus is roughly 1.6x the HBM bandwidth but it's full-duplex, so the effective bidirectional bandwidth is closer to 3x. By comparison, a TPU's VMEM is 2x larger *and* has much more bandwidth (around 40TB/s).

- **HBM:** the main GPU memory, used for storing model weights, gradients, activations, etc.

[9] Technically, the L2 cache is split in two, so half the SMs can access 25MB a piece on an H100. There is a link connecting the two halves, but at lower bandwidth.

[10] The fact that the L2 cache is shared across all SMs effectively forces the programmer to run the SMs in a fairly coordinated way anyway, despite the fact that, in principle, they are independent units.

- The HBM size has increased a lot from 32GB in Volta to 192GB in Blackwell (B200).
- The bandwidth from HBM to the CUDA Tensor Core is called HBM bandwidth or memory bandwidth, and is about 3.35TB/s on H100 and 9TB/s on B200.

12.1.2 Summary of GPU specs

Here is a summary of GPU specs for recent models. The number of SMs, clock speed, and FLOPs differ somewhat between variants of a given GPU. Here are memory capacity numbers:

GPU	Generation	Clock Speed	SMs/chip	SMEM/SM	L2/chip
V100	Volta	1.25/1.38GHz	80	96kB	6MB
A100	Ampere	1.10/1.41GHz	108	192kB	40MB
H100	Hopper	1.59/1.98GHz	132	256kB	50MB
H200	Hopper	1.59/1.98GHz	132	256kB	50MB
B200	Blackwell	?	148	256kB	126MB

Table 12.1: GPU memory specifications across generations (part 1).

GPU	HBM/chip
V100	32GB
A100	80GB
H100	80GB
H200	141GB
B200	192GB

Table 12.2: GPU memory specifications across generations (part 2).

All generations have 256kB of register memory per SM. Blackwell adds 256kB of TMEM per SM as well. Here are the FLOPs and bandwidth numbers for each chip:

We exclude B100 since it wasn't mass-produced.[11] Some specs depend slightly on the precise version of the GPU, since NVIDIA

[11]While NVIDIA made a B100 generation, they were only briefly sold and produced, allegedly due to design flaws that prevented them from running

GPU	Gen	HBM BW	bf16/fp16 FLOPs/s	fp8/int8 FLOPs/s
V100	Volta	9.0e11	—	—
A100	Ampere	2.0e12	3.1e14	6.2e14
H100	Hopper	3.4e12	9.9e14	2.0e15
H200	Hopper	4.8e12	9.9e14	2.0e15
B200	Blackwell	8.0e12	2.3e15	4.5e15

Table 12.3: GPU FLOPs and bandwidth specifications. B200 also supports 9.0e15 fp4 FLOPs/s.

GPUs aren't as standard as TPUs.

Here's a helpful cheat sheet comparing GPU and TPU components:

GPU	TPU	What is it?
SM	Tensor Core	Core "cell" with other units
Warp Scheduler	VPU	SIMD vector arithmetic unit
CUDA Core	VPU ALU	SIMD ALU
SMEM (L1)	VMEM	Fast on-chip cache memory
Tensor Core	MXU	Matrix multiplication unit
HBM (GMEM)	HBM	High BW high capacity memory

Table 12.4: GPU vs TPU component mapping.

12.1.3 GPUs vs. TPUs at the chip level

GPUs started out rendering video games, but since deep learning took off in the 2010s, they've started acting more and more like dedicated matrix multiplication machines – in other words, more like TPUs.[12] To an extent, this history explains why modern GPUs

close to their claimed specifications. They struggled to achieve peak FLOPs without throttling due to heat and power concerns.

[12]Before the deep learning boom, GPUs ("Graphics Processing Units") did, well, graphics – mostly for video games. Video games represent objects with millions of little triangles, and the game renders (or "rasterizes") these triangles into a 2D image that gets displayed on a screen 30-60 times a second (this

look the way they do. They weren't designed purely for LLMs or ML models but as general-purpose accelerators, and the hardware aims for level of "generality" that can be both a blessing and a curse. GPUs much more often "just work" when applied to new tasks and lean far less on a good compiler than TPUs do. But this also makes them much harder to reason about or get roofline performance out of, since so many compiler features can cause bottlenecks.

GPUs are more modular. TPUs have 1-2 big Tensor Cores, while GPUs have hundreds of small SMs. Likewise, each Tensor Core has 4 big VPU with 1024 ALUs each, while GPUs have an H100 has 132 * 4 = 528 small independent SIMD units. Here is a 1:1 comparison of GPUs to TPU that highlights this point:

GPU	TPU	H100 #	TPU v5p #
SM	Tensor Core	132	2
Warp Scheduler	VPU	528	8
SMEM (L1)	VMEM	32MB	128MB
Registers	VRegs	32MB	256kB
Tensor Core	MXU	528	8

Table 12.5: GPU vs TPU modularity comparison.

This difference in modularity on the one hand makes TPUs much cheaper to build and simpler to understand, but it also puts more burden on the compiler to do the right thing. Because TPUs have a single thread of control and only support vectorized VPU-

frequency is called the framerate). Rasterization involves projecting these triangles into the coordinate frame of the camera and calculating which triangles overlap which pixels, billions of times a second. As you can imagine, this is very expensive, and it's just the beginning. You then have to color each pixel by combining the colors of possibly several semi-opaque triangles that intersect the ray. GPUs were designed to do these operations extremely fast, with an eye towards versatility; you need to run many different GPU workloads (called "shaders") at the same time, with no single operation dominating. As a result, consumer graphics-focused GPUs can do matrix multiplication, but it's not their primary function.

wide instructions, the compiler needs to manually pipeline all memory loads and MXU/VPU work to avoid stalls. A GPU programmer can just launch dozens of different kernels, each running on a totally independent SM. On the other hand, those kernels might get horrible performance because they are thrashing the L2 cache or failing to coalesce memory loads; because the hardware controls so much of the runtime, it becomes hard to reason about what's going on behind the scenes. As a result, TPUs can often get closer to peak roofline performance with less work.

Historically, individual GPUs are more powerful (and more expensive) than a comparable TPU: A single H200 has close to 2x the FLOPs/s of a TPU v5p and 1.5x the HBM. At the same time, the sticker price on Google Cloud is around $10/hour for an H200 compared to $4/hour for a TPU v5p. TPUs generally rely more on networking multiple chips together than GPUs.

TPUs have a lot more fast cache memory. TPUs also have a lot more VMEM than GPUs have SMEM (+TMEM), and this memory can be used for storing weights and activations in a way that lets them be loaded and used extremely fast. This can make them faster for LLM inference if you can consistently store or prefetch model weights into VMEM.

12.1.4 Quiz 1: GPU hardware

Here are some problems to work through that test some of the content above. Answers are provided, but it's probably a good idea to try to answer the questions before looking, pen and paper in hand.

Question 1 [CUDA cores]: How many fp32 CUDA cores (ALUs) does an H100 have? B200? How does this compare to the number of independent ALUs in a TPU v5p?

Question 2 [Vector FLOPs calculation]: A single H100 has 132 SMs and runs at a clock speed of 1.59GHz (up to 1.98GHz boost). Assume it can do one vector op per cycle per ALU. How many vector fp32 FLOPs can be done per second? With boost? How does this compare to matmul FLOPs?

Question 3 [GPU matmul intensity]: What is the peak fp16 matmul intensity on an H100? A B200? What about fp8? *By intensity we mean the ratio of matmul FLOPs/s to memory bandwidth.*

Question 4 [Matmul runtime]: Using the answer to Question 3, how long would you expect an `fp16[64, 4096] * fp16[4096, 8192]` matmul to take on a single B200? How about `fp16[512, 4096] * fp16[4096, 8192]`?

Question 5 [L1 cache capacity]: What is the total L1/SMEM capacity for an H100? What about register memory? How does this compare to TPU VMEM capacity?

Question 6 [Calculating B200 clock frequency]: NVIDIA reports (see: `resources.nvidia.com/ en-us-blackwell-architecture`) that a B200 can perform 80TFLOPs/s of vector fp32 compute. Given that each CUDA core can perform 2 FLOPs/cycle in a FMA (fused multiply add) op, estimate the peak clock cycle.

Question 7 [Estimating H100 add runtime]: Using the figures above, calculate how long it ought to take to add two `fp32[N]` vectors together on a single H100. Calculate both T_{math} and T_{comms}. What is the arithmetic intensity of this operation? If you can get access, try running this operation in PyTorch or JAX as well for `N = 1024` and `N=1024 * 1024 * 1024`. How does this compare?

12.2 Networking

Networking is one of the areas where GPUs and TPUs differ the most. As we've seen, TPUs are connected in 2D or 3D tori, where each TPU is only connected to its neighbors. This means sending a message between two TPUs must pass through every intervening TPU, and forces us to use only uniform communication patterns over the mesh. While inconvenient in some respects, this also means the number of links per TPU is constant and we can scale to arbitrarily large TPU "pods" without loss of bandwidth.

GPUs on the other hand use a more traditional hierarchical

tree-based switching network. Sets of 8 GPUs called **nodes** (up to 72 for GB200[13]) are connected within 1 hop of each other using high-bandwidth interconnects called NVLinks, and these nodes are connected into larger units (called **SU**s or Scalable Units) with a lower bandwidth InfiniBand (IB) or Ethernet network using NICs attached to each GPU. These in turn can be connected into arbitrarily large units with higher level switches.

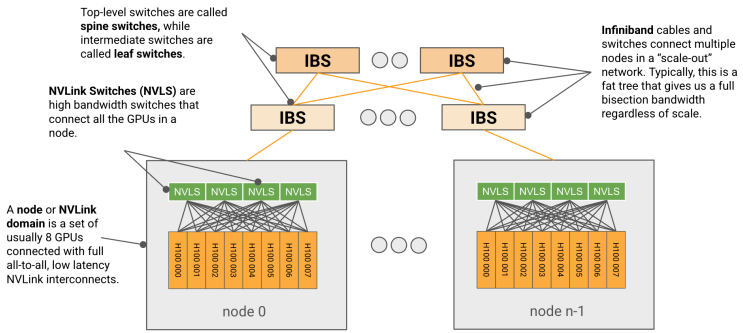


Figure 12.4: A diagram showing a typical H100 network. A set of 8 GPUs is connected into a node or NVLink domain with NVSwitches (also called NVLink switches), and these nodes are connected to each other with a switched InfiniBand fabric. H100s have about 450GB/s of egress bandwidth each in the NVLink domain, and each node has 400GB/s of egress bandwidth into the IB network.

12.2.1 At the node level

A GPU node is a small unit, typically of 8 GPUs (up to 72 for GB200), connected with all-to-all, full-bandwidth, low la-

[13]The term node is overloaded and can mean two things: the NVLink domain, aka the set of GPUs fully connected over NVLink interconnects, or the set of GPUs connected to a single CPU host. Before B200, these were usually the same, but in GB200 NVL72, we have an NVLink domain with 72 GPUs but still only 8 GPUs connected to each host. We use the term node here to refer to the NVLink domain, but this is controversial.

tency NVLink interconnects.[14] Each node contains several high-bandwidth NVSwitches which switch packets between all the local GPUs. The actual node-level topology has changed quite a bit over time, including the number of switches per node, but for H100, we have 4 NVSwitches per node with GPUs connected to them in a $5 + 4 + 4 + 5$ link pattern, as shown:

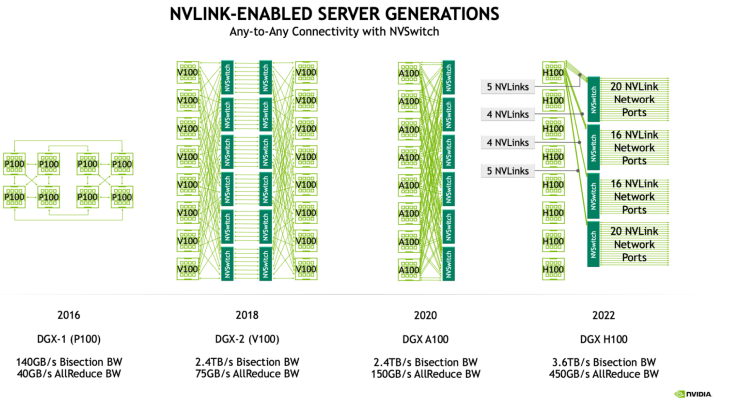


Figure 12.5: Node aka NVLink domain diagrams from Pascall (P100) onward. Since Volta (V100), we have had all-to-all connectivity within a node using a set of switches. The H100 node has 4 NVSwitches connected to all 8 GPUs with 25GB/s links.

For the Hopper generation (NVLink 4.0), each NVLink link has 25GB/s of full-duplex[15] bandwidth (50GB/s for B200), giving us $18 \times 25 = 450$GB/s of full-duplex bandwidth from each GPU into the network. The massive NVSwitches have up to 64 NVLink

[14]NVLink has been described to me as something like a souped-up PCIe connection, with low latency and protocol overhead but not designed for scalability/fault tolerance, while InfiniBand is more like Ethernet, designed for larger lossy networks.

[15]Full-duplex here means 25GB/s each way, with both directions independent of each other. You can send a total of 50GB/s over the link, but at most 25GB/s in each direction.

ports, meaning an 8xH100 node with 4 switches can handle up to $64 \times 25e9 \times 4 = 6.4$TB/s of bandwidth. Here's an overview of how these numbers have changed with GPU generation:

NVLink Gen	NVSwitch Gen	GPU Gen	BW (GB/s)	Ports / GPU	Node BW (GB/s)	Node size
3.0	2.0	Ampere	25	12	300	8
4.0	3.0	Hopper	25	18	450	8
5.0	4.0	Blackwell	50	18	900	8/72

Table 12.6: NVLink evolution across generations. BW is full-duplex bandwidth per link. Node BW is GPU-to-GPU bandwidth within the node. Blackwell supports nodes of 8 or 72 GPUs (with 2 or 18 NVSwitches respectively).

Blackwell (B200) has nodes of 8 GPUs. GB200NVL72 support larger NVLink domains of 72 GPUs. We show details for both the 8 and 72 GPUs systems.

12.2.2 Quiz 2: GPU nodes

Here are some more Q/A problems on networking. I find these particularly useful to do out, since they make you work through the actual communication patterns.

Question 1 [Total bandwidth for H100 node]: How much total bandwidth do we have per node in an 8xH100 node with 4 switches? *Hint:* consider both the NVLink and NVSwitch bandwidth.

Question 2 [Bisection bandwidth]: Bisection bandwidth is defined as the smallest bandwidth available between any even partition of a network. In other words, if split a network into two equal halves, how much bandwidth crosses between the two halves? Can you calculate the bisection bandwidth of an 8x H100 node? *Hint:* bisection bandwidth typically includes flow in both directions.

Question 3 [AllGather cost]: Given an array of B bytes, how long would a (throughput-bound) AllGather take on an

8xH100 node? Do the math for `bf16[`D_X`, F]` where `D=4096`, `F=65,536`. *It's worth reading the TPU collectives section before answering this. Think this through here but we'll talk much more about collectives next.*

12.2.3 Beyond the node level

Beyond the node level, the topology of a GPU network is less standardized. NVIDIA publishes a reference DGX SuperPod architecture[16] that connects a larger set of GPUs than a single node using InfiniBand, but customers and datacenter providers are free to customize this to their needs.[17]

Here is a diagram for a reference 1024 GPU H100 system, where each box in the bottom row is a single 8xH100 node with 8 GPUs, 8 400Gbps CX7 NICs (one per GPU), and 4 NVSwitches.

[16] See NVIDIA documentation: `docs.nvidia.com/dgx-superpod/reference-architecture-scalable-infrastructure-h100/latest/network-fabrics.html`

[17] For instance, Meta trained LLaMA-3 on a datacenter network that differs significantly from this description, using Ethernet, a 3 layer switched fabric, and an oversubscribed switch at the top level.

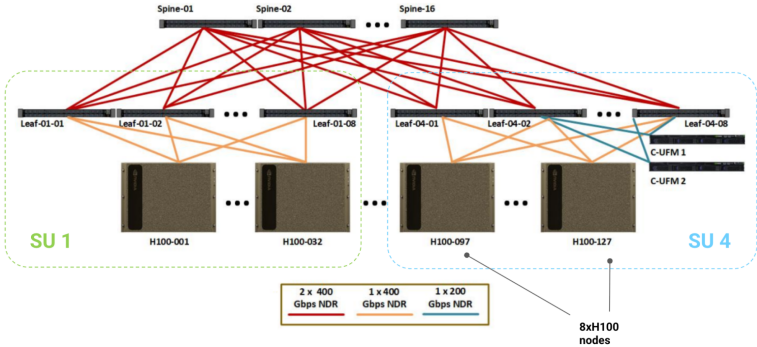


Figure 12.6: Diagram of the reference 1024 H100 DGX SuperPod with 128 nodes (sometimes 127), each with 8 H100 GPUs, connected to an InfiniBand scale-out network. Sets of 32 nodes (256 GPUs) are called 'Scalable Units' or SUs. The leaf and spine IB switches provide enough bandwidth for full bisection bandwidth between nodes.

Scalable Units: Each set of 32 nodes is called a "Scalable Unit" (or SU), under a single set of 8 leaf InfiniBand switches. This SU has 256 GPUs with 4 NVSwitches per node and 8 Infiniband leaf switches. All the cabling shown is InfiniBand NDR (50GB/s full-duplex) with 64-port NDR IB switches (also 50GB/s per port). *Note that the IB switches have 2x the bandwidth of the NVSwitches (64 ports with 400 Gbps links).*

SuperPod: The overall SuperPod then connects 4 of these SUs with 16 top level "spine" IB switches, giving us 1024 GPUs with 512 node-level NVSwitches, 32 leaf IB switches, and 16 spine IB switches, for a total of $512 + 32 + 16 = 560$ switches. Leaf switches are connected to nodes in sets of 32 nodes, so each set of 256 GPUs has 8 leaf switches. All leaf switches are connected to all spine switches.

How much bandwidth do we have? The overall topology of the InfiniBand network (called the "scale out network") is that of a **fat tree**, with the cables and switches guaranteeing full bisection bandwidth above the node level (here, 400GB/s). That means if

we split the nodes in half, each node can egress 400GB/s to a node in the other partition at the same time. More to the point, this means we should have a roughly constant AllReduce bandwidth in the scale out network! While it may not be implemented this way, you can imagine doing a ring reduction over arbitrarily many nodes in the scale-out network, since you can construct a ring including every one.

Level	GPUs	Switches per Unit	Type	BW/Unit (TB/s)
Node	8	4	NVL	3.6
Leaf	256	8	IB	12.8
Spine	1024	16	IB	51.2

Table 12.7: H100 SuperPod bandwidth levels. BW/Unit is full-duplex bandwidth per unit. GPU-to-GPU bandwidth is 450GB/s at node level and 50GB/s at leaf/spine levels. Fat tree bandwidth is 450GB/s within node and 400GB/s between nodes.

By comparison, a TPU v5p has about 90GB/s egress bandwidth per link, or 540GB/s egress along all axes of the 3D torus. This is not point-to-point so it can only be used for restricted, uniform communication patterns, but it still gives us a much higher TPU to TPU bandwidth that can scale to arbitrarily large topologies (at least up to 8960 TPUs).

The GPU switching fabric can in theory be extended to arbitrary sizes by adding additional switches or layers of indirection, at the cost of additional latency and costly network switches.

Key Takeaway

Takeaway: Within an H100 node, we have a full fat tree bandwidth of 450GB/s from each GPU, while beyond the node, this drops to 400GB/s node-to-node. This will turn out to be critical for communication primitives.

GB200 NVL72s: NVIDIA has recently begun producing new GB200 NVL72 GPU clusters that combine 72 GPUs in a single NVLink domain with full 900GB/s of GPU to GPU bandwidth. These domains can then be linked into larger SuperPods with proportionally higher (9x) IB fat tree bandwidth. Here is a diagram of that topology:

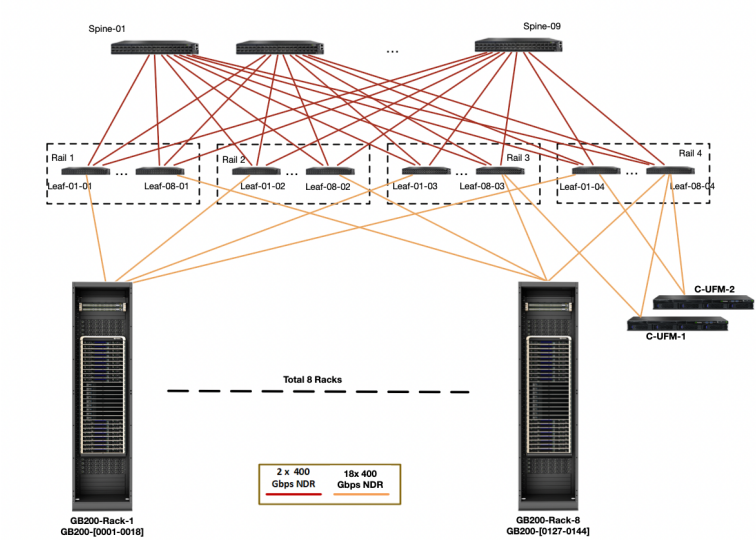


Figure 12.7: A diagram showing a GB200 DGX SuperPod of 576 GPUs. Each rack at the bottom layer contains 72 GB200 GPUs.

Counting the egress bandwidth from a single node (the orange lines above), we have $4 \times 18 \times 400/8 = 3.6$TB/s of bandwidth to the leaf level, which is 9x more than an H100 (just as the node contains 9x more GPUs). That means the critical node egress bandwidth is much, *much* higher and our cross-node collective bandwidth can actually be *lower* than within the node. See Appendix A for more discussion.

Node Type	GPUs per node	GPU egress BW (GB/s)	Node egress BW (GB/s)
H100	8	450	400
B200	8	900	400
GB200 NVL72	72	900	3600

Table 12.8: Node types comparison. GPU egress BW is full-duplex bandwidth from each GPU within the node. Node egress BW is full-duplex bandwidth from the entire node to the scale-out network.

Key Takeaway

Takeaway: GB200 NVL72 SuperPods drastically increase the node size and egress bandwidth from a given node, which changes our rooflines significantly.

12.2.4 Quiz 3: Beyond the node level

Question 1 [Fat tree topology]: Using the DGX H100 diagram above, calculate the bisection bandwidth of the entire 1024 GPU pod at the node level. Show that the bandwidth of each link is chosen to ensure full bisection bandwidth. *Hint: make sure to calculate both the link bandwidth and switch bandwidth.*

Question 2 [Scaling to a larger DGX pod]: Say we wanted to train on 2048 GPUs instead of 1024. What would be the simplest/best way to modify the above DGX topology to handle this? What about 4096? *Hint: there's no single correct answer, but try to keep costs down. Keep link capacity in mind. See NVIDIA documentation at:* `docs.nvidia.com/ dgx-superpod-reference-architecture-dgx-h100.pdf` *for details.*

12.3 How Do Collectives Work on GPUs?

GPUs can perform all the same collectives as TPUs: ReduceScatters, AllGathers, AllReduces, and AllToAlls. Unlike TPUs, the way these work changes depending on whether they're performed at the node level (over NVLink) or above (over InfiniBand). These collectives are implemented by NVIDIA in the NVSHMEM[18] and NCCL[19] (pronounced "nickel") libraries. NCCL is open-sourced at https://github.com/NVIDIA/nccl. While NCCL uses a variety of implementations depending on latency requirements/topology,[20] from here on, we'll discuss a theoretically optimal model over a switched tree fabric.

12.3.1 Intra-node collectives

AllGather or ReduceScatter: For an AllGather or ReduceScatter at the node level, you can perform them around a ring just like a TPU, using the full GPU-to-GPU bandwidth at each hop. Order the GPUs arbitrarily and send a portion of the array around the ring using the full GPU-to-GPU bandwidth.[21] The cost of each hop is $T_{\text{hop}} = \text{bytes}/(N \times \text{GPU egress bandwidth})$, so the overall cost is

$$T_{\text{AG or RS comms}} = \frac{\text{bytes} \cdot (N-1)}{N \cdot \text{GPU egress bandwidth}} \to \frac{\text{bytes}}{\text{GPU egress bandwidth}} \tag{12.1}$$

You'll note this is exactly the same as on a TPU. For an AllReduce, you can combine an RS + AG as usual for twice the cost.

[18] https://developer.nvidia.com/nvshmem

[19] https://developer.nvidia.com/nccl

[20] See https://github.com/NVIDIA/nccl/issues/1415#issuecomment-2310650081 for details.

[21] You can also think of each GPU sending its chunk of size bytes/N to each of the other $N-1$ GPUs, for a total of $(N-1) \times N \times \text{bytes}/N$ bytes communicated, which gives us the same result.

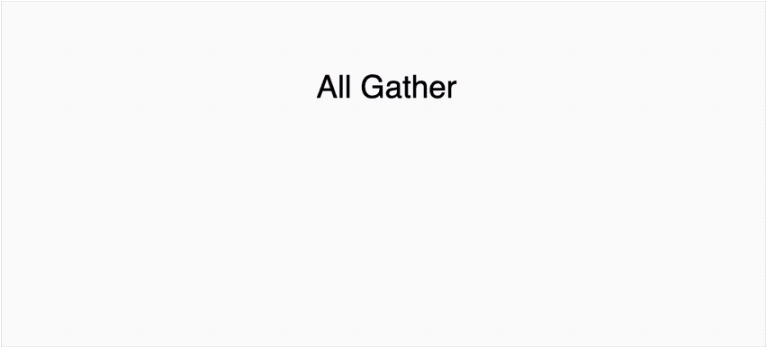


Figure 12.8: Bandwidth-optimal 1D ring AllGather algorithm. For B bytes, this sends V/X bytes over the top-level switches $X-1$ times. Note: This is a static version of an animated diagram in the original source showing the progression of data movement around the ring.

If you're concerned about latency (e.g. if your array is very small), you can do a tree reduction, where you AllReduce within pairs of 2, then 4, then 8 for a total of $\log(N)$ hops instead of $N-1$, although the total cost is still the same.

Key Takeaway

The cost to AllGather or ReduceScatter an array of B bytes within a single node is about $T_{\text{comms}} = B \times (8-1)/(8 \times W_{\text{GPU egress}}) \approx B/W_{\text{GPU egress}}$. This is theoretically around B/450e9 on an H100 and B/900e9 on a B200. An AllReduce has 2x this cost unless in-network reductions are enabled.

Pop Quiz 1 [AllGather time]: Using an 8xH100 node with 450 GB/s full-duplex bandwidth, how long does **AllGather**(`bf16`$[B_X, F]$) take? Let $B = 1024$, $F = 16{,}384$.

AllToAlls: GPUs within a node have all-to-all connectivity, which makes AllToAlls, well, quite easy. Each GPU just sends

directly to the destination node. Within a node, for B bytes, each GPU has B/N bytes and sends (B/N^2) bytes to $N-1$ target nodes for a total of

$$T_{\text{AllToAll comms}} = \frac{B \cdot (N-1)}{W \cdot N^2} \approx \frac{B}{W \cdot N} \tag{12.2}$$

Compare this to a TPU, where the cost is $B/(4W)$. Thus, within a single node, we get a 2X theoretical speedup in runtime ($B/4W$ vs. $B/8W$).

For Mixture of Expert (MoE) models, we frequently want to do a *sparse or ragged AllToAll*, where we guarantee at most k of N shards on the output dimension are non-zero, that is to say $T_{\text{AllToAll}} \to K[B, N]$ where at most k of N entries on each axis are non-zero. The cost of this is reduced by k/N, for a total of about $\min(k/N, 1) \cdot B/(W \cdot N)$. For an MoE, we often pick the non-zero values independently at random, so there's some chance of having fewer than k non-zero, giving us approximately $(N-1)/N \cdot \min(k/N, 1) \cdot B/(W \cdot N)$.[22]

Pop Quiz 2 [AllToAll time]: Using an 8xH100 node with 450 GB/s unidirectional bandwidth, how long does **AllToAll**$_{X \to N}$(`bf16`$[B_X, N]$) take? What if we know only 4 of 8 entries will be non-zero?

Key Takeaway

The cost of an AllToAll on an array of B bytes on GPU within a single node is about $T_{\text{comms}} = (B \cdot (8-1))/(8^2 \cdot W_{\text{GPU egress}}) \approx B/(8 \cdot W_{\text{GPU egress}})$. For a ragged (top-$k$) AllToAll, this is decreased further to $(B \cdot k)/(64 \cdot W_{\text{GPU egress}})$.

Empirical measurements: here is an empirical measurement of AllReduce bandwidth over an 8xH100 node. The Algo BW

[22] The true cost is actually $(1-((Z-1)/Z)^K) \cdot (Z-1)/Z$, the expected number of distinct outcomes in K dice rolls, but it is very close to the approximation given. See the Appendix for more details.

is the measured bandwidth (bytes / runtime) and the Bus BW is calculated as $2 \cdot W \cdot (8-1)/8$, theoretically a measure of the actual link bandwidth. You'll notice that we do achieve close to 370GB/s, less than 450GB/s but reasonably close, although only around 10GB/device. This means although these estimates are theoretically correct, it takes a large message to realize it.

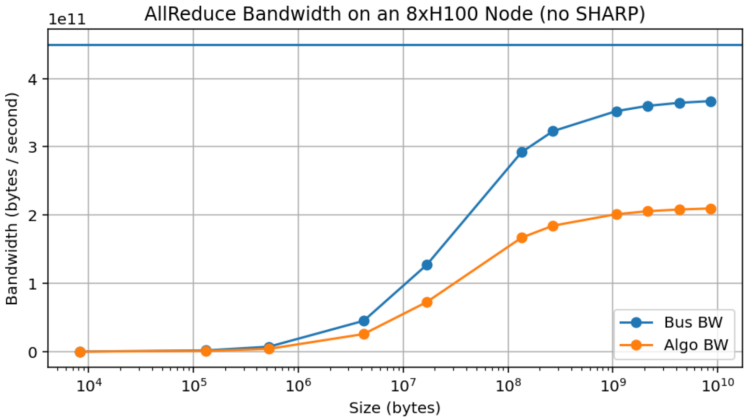


Figure 12.9: AllReduce throughput for an 8xH100 node with SHARP disabled. The blue curve is the empirical link bandwidth, calculated as $2 \times \text{bytes} \times (N-1)/(N \times \text{runtime})$ from the empirical measurements. Note that we do not get particularly close to the claimed bandwidth of 450GB/s, even with massive 10GB arrays.

This is a real problem, since it meaningfully complicates any theoretical claims we can make, since e.g. even an AllReduce over a reasonable sized array, like LLaMA-3 70B's MLPs (of size `bf16[8192, 28672]`, or with 8-way model sharding, `bf16[8192, 3584] = 58MB`) can achieve only around 150GB/s compared to the peak 450GB/s. By comparison, TPUs achieve peak bandwidth at much lower message sizes (see Appendix B).

Key Takeaway

Although NVIDIA claims bandwidths of about 450GB/s over an H100 NVLink, it is difficult in practice to exceed 370 GB/s, so adjust the above estimates accordingly.

In network reductions: Since the Hopper generation, NVIDIA switches have supported SHARP[23] (Scalable Hierarchical Aggregation and Reduction Protocol) which allows for "in-network reductions". This means *the network switches themselves* can do reduction operations and multiplex or "MultiCast" the result to multiple target GPUs:

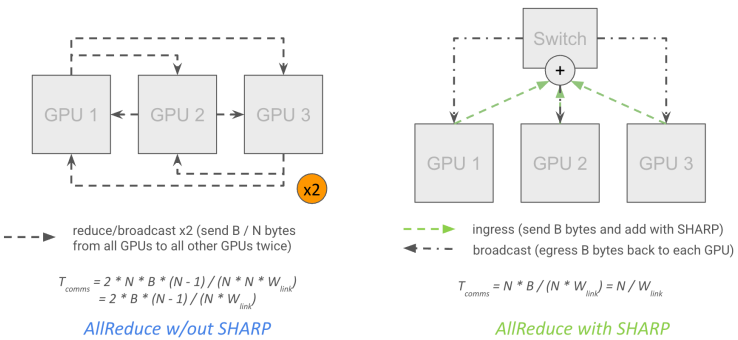


Figure 12.10: An AllReduce without SHARP has 2x the theoretical cost because it has to pass through each GPU twice. In practice, speedups are only about 30% (from NCCL 2.27.5).

Theoretically, this close to halves the cost of an AllReduce, since it means each GPU can send its data to a top-level switch which itself performs the reduction and broadcasts the result to each GPU without having to egress each GPU twice, while also

[23] "SHARP" stands for Scalable Hierarchical Aggregation and Reduction Protocol. See https://developer.nvidia.com/blog/advancing-performance-with-nvidia-sharp-in-network-computing/

reducing network latency.

$$T_{\text{SHARP AR comms}} = \frac{\text{bytes}}{\text{GPU egress bandwidth}} \tag{12.3}$$

Note that this is exact and not off by a factor of $1/N$, since each GPU egresses $B{\cdot}(N{-}1)/N$ first, then receives the partially reduced version of its local shard (ingress of B/N), finishes the reductions, then egresses B/N again, then ingresses the fully reduced result (ingress of $B \cdot (N-1)/N$), resulting in exactly B bytes ingressed.

However, in practice we see about a 30% increase in bandwidth with SHARP enabled, compared to the predicted 75%. This gets us up merely to about 480GB/s effective collective bandwidth, not nearly 2x.

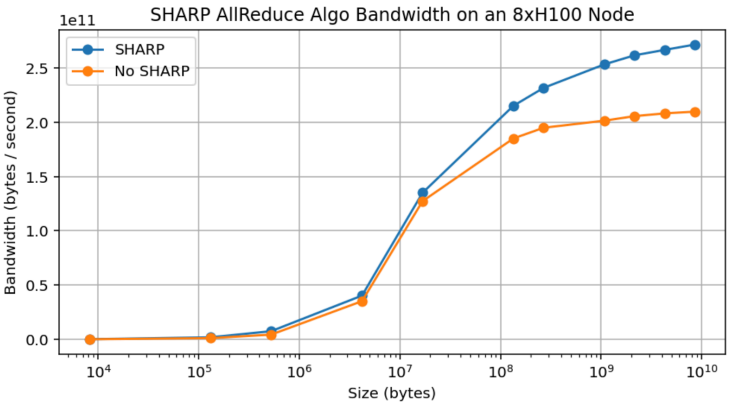


Figure 12.11: Empirical measurements of AllReduce algo bandwidth with and without NVIDIA SHARP enabled within a node. The gains amount to about 30% throughput improvement at peak, even though algorithmically it ought to be able to achieve closer to a 75% gain.

Key Takeaway

In theory, NVIDIA SHARP (available on most NVIDIA switches) should reduce the cost of an AllReduce on B bytes from about $2 \times B/W$ to B/W. However, in practice we only see a roughly 30% improvement in bandwidth. Since pure AllReduces are fairly rare in LLMs, this is not especially useful.

12.3.2 Cross-node collectives

When we go beyond the node-level, the cost is a bit more subtle. When doing a reduction over a tree, you can think of reducing from the bottom up, first within a node, then at the leaf level, and then at the spine level, using the normal algorithm at each level. For an AllReduce especially, you can see that this allows us to communicate less data overall, since after we AllReduce at the node level, we only have to egress B bytes up to the leaf instead of $B \times N$.

How costly is this? To a first approximation, because we have full bisection bandwidth, the cost of an AllGather or ReduceScatter is roughly the buffer size in bytes divided by the node egress bandwidth (400GB/s on H100) *regardless of any of the details of the tree reduction.*

$$T_{\text{AG or RS comms}} = \frac{\text{bytes}}{W_{\text{node egress}}} \underset{\text{H100}}{=} \frac{\text{bytes}}{400\text{e}9} \tag{12.4}$$

where $W_{\text{node egress}}$ is generally 400GB/s for the above H100 network (8x400Gbps IB links egressing each node). The cleanest way to picture this is to imagine doing a ring reduction over *every node in the cluster*. Because of the fat tree topology, we can always construct a ring with $W_{\text{node egress}}$ between any two nodes and do a normal reduction. The node-level reduction will (almost) never be the bottleneck because it has a higher overall bandwidth and better latency, although in general the cost is

$$T_{\text{total}} = \max(T_{\text{comms at node}}, T_{\text{comms in scale-out network}})$$

$$= \max\left[\frac{\text{bytes}}{W_{\text{GPU egress}}}, \frac{\text{bytes}}{W_{\text{node egress}}}\right] \tag{12.5}$$

More precise derivation: We can be more precise in noting that we are effectively doing a ring reduction at each layer in the network, which we can mostly overlap, so we have:

$$T_{\text{AG or RS comms}} = \text{bytes} \cdot \max_{\text{depth } i}\left[\frac{D_i - 1}{D_i \cdot W_{\text{link } i}}\right] \tag{12.6}$$

where D_i is the degree at depth i (the number of children at depth i), $W_{\text{link } i}$ is the bandwidth of the link connecting each child to node i.

Using this, we can calculate the available AllGather/AllReduce bandwidth as $\min_{\text{depth } i}(D_i \times W_{\text{link } i}/(D_i - 1))$ for a given topology. In the case above, we have:

- **Node:** $D_{\text{node}} = 8$ since we have 8 GPUs in a node with $W_{\text{link } i}$ = 450GB/s. Thus we have an AG bandwidth of `450e9 * 8 / (8 - 1) = 514GB/s`.
- **Leaf:** $D_{\text{leaf}} = 32$ since we have 32 nodes in an SU with $W_{\text{link } i}$ = 400GB/s (8x400Gbps IB links). Thus our bandwidth is `400e9 * 32 / (32 - 1) = 413GB/s`.
- **Spine:** $D_{\text{spine}} = 4$ since we have 4 SUs with $W_{\text{link } i}$ = 12.8TB/s (from `8 * 16 * 2 * 400Gbps` links above). Our bandwidth is `12.8e12 * 4 / (4 - 1) = 17.1TB/s`.

Hence our overall AG or RS bandwidth is `min(514GB/s, 413GB/s, 17.1TB/s) = 413GB/s` at the leaf level, so in practice $T_{\text{AG or RS comms}} = B/413\text{GB/s}$, i.e. we have about 413GB/s of AllReduce bandwidth even at the highest level. For an AllReduce with SHARP, it will be slightly lower than this (around 400GB/s) because we don't have the $(N-1)/N$ factor. Still, 450GB/s and 400GB/s are close enough to use as approximations.

Other collectives: AllReduces are still 2x the above cost unless SHARP is enabled. NVIDIA sells SHARP-enabled IB switches

as well, although not all providers have them. AllToAlls do change quite a bit cross-node, since they aren't "hierarchical" in the way AllReduces are. If we want to send data from every GPU to every other GPU, we can't use take advantage of the full bisection bandwidth at the node level. That means if we have an N-way AllToAll that spans $M = N/8$ nodes, the cost is

$$T_{\text{AllToAll comms}} = \frac{B \cdot (M-1)}{M^2 \cdot W_{\text{node egress}}} \approx \frac{B}{M \cdot W_{\text{node egress}}} \tag{12.7}$$

which effectively has 50GB/s rather than 400GB/s of bandwidth. We go from $B/(8 \times 450\text{e}9)$ within a single H100 node to $B/(2 \times 400\text{e}9)$ when spanning 2 nodes, a more than 4x degradation.

Here is a summary of the 1024-GPU DGX H100 SuperPod architecture:

Table 12.9: DGX H100 SuperPod architecture summary showing collective bandwidths at different levels.

Level	# GPUs	Degree (Children)	Switch BW (TB/s)	Cable BW (TB/s)	Coll. BW (GB/s)
Node	8	8	6.4	3.6	450
Leaf (SU)	256	32	25.6	12.8	400
Spine	1024	4	51.2	51.2	400

We use the term "Collective Bandwidth" to describe the effective bandwidth at which we can egress either the GPU or the node. It's also the bisection bandwidth $\times\ 2/N$.

Key Takeaway

Beyond the node level, the cost of an AllGather or ReduceScatter on B bytes is roughly $B/W_{\text{node egress}}$, which is $B/400\text{e}9$ on an H100 DGX SuperPod, while AllReduces cost twice as much unless SHARP is enabled. The overall topology is a fat tree designed to give constant bandwidth between any two pairs of nodes.

Reductions when array is sharded over a separate axis: Consider the cost of a reduction like

$$\textbf{AllReduce}_X(A[I_Y, J]\ \{U_X\}) \tag{12.8}$$

where we are AllReducing over an array that is itself sharded along another axis Y. On TPUs, the overall cost of this operation is reduced by a factor of $1/Y$ compared to the unsharded version since we're sending $1/Y$ as much data per axis. On GPUs, the cost depends on which axis is the "inner" one (intra-node vs. inter-node) and whether each shard spans more than a single node. Assuming Y is the inner axis, and the array has bytes total bytes, the overall cost is reduced effectively by Y, but only if Y spans multiple nodes:

$$T_{\text{comms at node}} = \frac{\text{bytes}}{W_{\text{GPU egress}}} \cdot \frac{1}{\min(Y, D_{\text{node}})} \tag{12.9}$$

$$T_{\text{comms in scale-out network}} = \frac{\text{bytes}}{W_{\text{node egress}}} \cdot \frac{D_{\text{node}}}{\max(D_{\text{node}}, Y)} \tag{12.10}$$

$$T_{\text{total}} = \max(T_{\text{comms at node}}, T_{\text{comms in scale-out network}}) \tag{12.11}$$

where N is the number of GPUs and again D_{node} is the number of GPUs in a node (the degree of the node). As you can see, if $Y < D_{\text{node}}$, we get a win at the node level but generally don't see a reduction in overall runtime, while if $Y > D_{\text{node}}$, we get a speedup proportional to the number of nodes spanned.

If we want to be precise about the ring reduction, the general rule for a tree $\textbf{AllGather}_X(A_Y\ \{\ U_X\ \})$ (assuming Y is the inner axis) is

$$T_{\text{AR or RS comms}} = \text{bytes} \cdot \max_{\text{depth } i} \left[\frac{D_i - 1}{D_i \cdot \max(Y, S_{i-1}) \cdot W_{\text{link } i}} \right] \tag{12.12}$$

where S_i is $M \times N \times ...$, the size of the subnodes below level i in the tree. This is roughly saying that the more GPUs or nodes we span, the greater our available bandwidth is, but only within that node.

Pop Quiz 3 [Sharding along 2 axes]: Say we want to perform $\textbf{AllGather}_X(\texttt{bf16}[D_X, F_Y])$ where Y is the inner axis over a single SU (256 chips). How long will this take as a function of D, F, and Y?

Key Takeaway

When we have multiple axes of sharding, the cost of the outer reduction is reduced by a factor of the number of nodes spanned by the inner axis.

12.3.3 Quiz 4: Collectives

Question 1 [SU AllGather]: Consider only a single SU with M nodes and N GPUs per node. Precisely how many bytes are ingressed and egressed by the node level switch during an AllGather? What about the top-level switch?

Question 2 [Single-node SHARP AR]: Consider a single node with N GPUs per node. Precisely how many bytes are ingressed and egressed by the switch during an AllReduce using SHARP (in-network reductions)?

Question 3 [Cross-node SHARP AR]: Consider an array $\texttt{bf16}[D_X, F_Y]$ sharded over a single node of N GPUs. How long does $\textbf{AllReduce}(\texttt{bf16}[D, F_Y]\ \{ U_X \})$ take? You can assume we do in-network reductions. Explain how this differs if we have more than a single node?

Question 4 [Spine level AR cost]: Consider the same setting as above, but with $Y = 256$ (so the AR happens at the spine level). How long does the AllReduce take? Again, feel free to assume in-network reductions.

Question 5 [2-way AllGather cost]: Calculate the precise cost of an AllGather of B bytes over exactly 2 nodes. *Make sure to calculate the precise cost and not the approximation, and consider both the intra-node and cross-node cost.*

12.4 Rooflines for LLM Scaling on GPUs

Now let's look at what this has all been building towards: understanding rooflines for LLM scaling on GPU. This is to complement the TPU training chapter. As we did there, the goal here is to look at the total T_{math} and T_{comms} for different parallelism strategies and understand at what point $T_{\text{comms}} > T_{\text{math}}$. As before, we consider only the MLP block with operations

$$\text{MLP}(x) \equiv x[B, D] *_D W_{\text{in}}[D, F] \cdot_F W_{\text{out}}[F, D]$$

where B is the global batch size **in tokens** (i.e. B = batch size$\cdot$ sequence length).

Here we'll reproduce the table above showing effective bandwidths at both the GPU and node level:

Node Type	GPUs per node	GPU egress BW (GB/s)	Node egress BW (GB/s)
H100	8	450	400
B200	8	900	400
GB200 NVL72	72	900	3600

Table 12.10: Node types comparison showing effective bandwidths. GPU egress BW is bandwidth from each GPU within the node. Node egress BW is bandwidth from the entire node to the scale-out network.

Note: Both the GPU and node egress bandwidths determine rooflines for our LLMs. We'll use the term $W_{\text{collective}}$ to describe either the GPU or node bandwidths depending on whether we are operating within or above the node level.

Let's look at the compute communication rooflines as we did for TPUs for **data parallelism, tensor parallelism, pipeline parallelism, expert parallelism,** and combinations thereof. For the rest of this section we'll focus on H100 rooflines for specific calculations. GB200-NVL72 has the same general rooflines but

because we have a larger node egress bandwidth, we can sometimes be bottlenecked at the node level instead.

12.4.1 Data Parallelism

As noted before, DP and ZeRO sharding involve either a weight AllReduce or a ReduceScatter + AllGather in the backward pass. Since these both have the same cost, to be compute-bound for pure data parallelism or FSDP *without in-network reductions*, we have, per layer, in the backward pass, with an axis of size X:

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

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

Therefore, for $T_{\text{math}} > T_{\text{comms}}$, we need $B/(XC) > 1/W_{\text{collective}}$ or

$$\frac{B}{X} > \frac{C}{W_{\text{collective}}}$$

where $W_{\text{collective}}$ is either the GPU or node level egress bandwidth depending on whether we're sharding within a node or across nodes. Thus:

- **Within a node**, we just need the per-device **token** batch size $> 990\text{e}12/450\text{e}9 = 2200$.
- **Within an SU or at the spine level**, BS $> 990\text{e}12/400\text{e}9 = 2475$.

This is quite a bit higher than on a TPU, where the number is 850 with all three axes. For instance, LLaMA-3, which trained on 16000 H100s would need a batch size of at least 40M tokens (for reference, they used 16M). DeepSeek v3 trained on 2048 H800 GPUs with lower 300GB/s of bandwidth (instead of 450GB/s on

H100) would need 990e12/300e9 = 3300 tokens per GPU, or about 6.7M (in practice, they used 4M).

With in-network reductions enabled and using pure data parallelism, theoretically we have 2x the AllReduce bandwidth, which would halve both of these numbers. However, in practice the benefit is closer to 30%, which only really makes up for the fact that we typically struggle to reach the reported numbers. Furthermore, because pure data parallelism is rarely useful, this basically doesn't matter in practice.

MoE models: For a Mixture of Experts (MoE) model, where we have E experts and k experts per token, this increases to

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

$$T_{\text{comms}} = \frac{2 \cdot 2 \cdot 2 \cdot EDF}{W_{\text{collective}}}$$

which inflates the per-device token batch size by a factor of E/k, i.e.

$$\frac{B}{X} > \frac{E}{k} \frac{C}{W_{\text{collective}}}$$

For example, the new OpenAI OSS model with $k = 4$ and $E = 128$, this increases to `32 * 2475 = 79,200` across nodes, a kind of ridiculously high number.

What happens when X is small? When we do only e.g. 2-node data parallelism, we benefit from the $(X-1)/X$ scaling, which gives us

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

$$T_{\text{comms}} = \frac{2 \cdot 2 \cdot 2 \cdot DF \cdot (X-1)}{X \cdot W_{\text{collective}}}$$

where X is the number of nodes and $N = 8 \cdot X$. Then for a dense model we have $B/N > \alpha \cdot (X-1)/X$, or e.g. $B/N > 1237$,

half the above value. You'll notice 2-way data parallelism fairly often for this reason.

Key Takeaway

Takeaway: Data parallelism and ZeRO sharding require a per-device batch size of about 2500 tokens to be compute-bound on an H100 or B200, assuming perfect overlap and FLOPs utilization. For MoE models, this increases by a factor of E/k, the ratio of total to activated parameters. When doing a small amount of data parallelism, the critical batch size decreases.

12.4.2 Tensor Parallelism

Tensor parallelism requires an AllGather and ReduceScatter over the activations, which we need to overlap with the MLP FLOPs. In other words, in the forward pass, we have

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

$$T_{\text{comms}} = \frac{2 \cdot 2 \cdot BD}{W_{\text{collective}}}$$

which to be compute-bound gives us the rule

$$Y < \frac{F \cdot W_{\text{collective}}}{C}$$

Within a node, this gives us about $F/2200$ or $F/2475$ beyond a node. For $F = 28000$ like LLaMA-3, this is about 11-way TP (or rounding down, about 8-way, which is how large a node is). As with above, we get an extra 2X bandwidth when we span exactly 2 nodes, so we can generally do 16-way tensor parallelism ($F > 2475 \cdot (Y - 8)$), which gives us up to 19-way model parallelism in theory.

Key Takeaway

Takeaway: Tensor parallelism over an axis of size Y with feed-forward dimension F becomes bandwidth-bound when the $Y > F/2475$, which generally constrains us to only intra-node TP or at most 2-node TP.

12.4.3 Expert Parallelism

As we've already noted above, Mixture of Expert (MoE) models come with E times more model weights with only k times more FLOPs, making data parallelism significantly harder. We can mitigate this somewhat by sharding the our weights along the expert dimension, i.e. $W_{\text{in}}[E_Z, D, F]$. To do the MLP block, we need to introduce 2x AllToAll to send our activations to the corresponding experts.

As noted above, the cost of this $\text{AllToAll}_{Z \to k}([B, D, k])$ if it spans multiple nodes is roughly $T_{\text{AllToAll}} = 2 \cdot B \cdot D \cdot (Z-8)/Z \min(8 * k/Z, 1)$, so for pure expert parallelism we need

$$T_{\text{math}} = \frac{4 \cdot B \cdot k \cdot D \cdot F}{Z \cdot C}$$

$$T_{\text{comms}} = \frac{4 \cdot B \cdot D \cdot (Z-8)}{W \cdot Z} \cdot \min\left(\frac{8 \cdot k}{Z}, 1\right)$$

We either need $K > Z/8$ with $F > \alpha \cdot (Z-8)/k$ or $Z \gg K$ and $F > 8 \cdot \alpha$, where $\alpha = C/W$. This gives you two domains in which expert parallelism is possible, one with a small amount of expert parallelism (roughly 2-node) and small F, or one with large F and Z arbitrarily large (up to E-way expert parallelism).

You'll see both cases in practice, either a small amount of expert-parallelism (like DeepSeek v3 which has very small F and relatively small, restricted cross-node expert parallelism), or models with large F, in which case we can do significant cross-node EP alongside TP.

Key Takeaway

Takeaway: if $F < 8 * C/W_{\text{node}}$, expert parallelism can span 1-2 nodes with similar (slightly lower) cost to TP, or if $F > 8*C/W_{\text{node}}$, we can do a significant amount of expert parallelism (up to E nodes) with relatively low cost.

12.4.4 Pipeline Parallelism

Pipeline parallelism splits layers across nodes with an extremely low communication cost, since we are just sending small micro-batches of activations every couple layers. Historically pipelining has suffered from "pipeline bubbles", but with new zero-bubble pipelining approaches, it is typically possible to do without.

The overall communication cost of pipelining is tiny: with N_{MB} microbatches and N_{stages}, we have $T_{\text{comms per hop}} = 2 \cdot B \cdot D/(W \cdot N_{\text{MB}})$ and $N_{\text{MB}} + N_{\text{stages}} - 2$ hops, so roughly

$$T_{\text{total PP comms}} = \frac{2BD}{W \cdot N_{\text{MB}}} \cdot (N_{\text{MB}} + N_{\text{stages}} - 2)$$

$$T_{\text{per-layer comms}} \approx 1.5 \cdot \frac{2BD}{W \cdot N_{\text{layers}}}$$

Since we are dividing by N_{layers}, this is vastly smaller than any of the other costs. In other words, from a communication standpoint, pipelining is basically free. So why don't we just do pipelining? There are a few reasons:

(1) **Code complexity:** pipelining doesn't fit nicely as nicely into automatic parallelism frameworks (like XLA's GSPMD) as other approaches. Because it introduces microbatching to hide pipeline bubbles, it changes the structure of the program, and custom zero-bubble pipeline schedules exacerbate this problem by requiring complicated interleaving of the forward and backward pass.

(2) **Pipelining makes data parallelism and FSDP hard:** probably the biggest reason not to do pipelining is that it plays

badly with FSDP and data parallelism. ZeRO-3 sharding in particular works badly, since it requires us to AllGather the weights on every microbatch which doesn't work when we have only $B/N_{\text{microbatches}}$ tokens to amortize the AllGather cost. Furthermore, during the backward pass, *we can't AllReduce or ReduceScatter the gradients until the last microbatch has passed a given stage, which means we have significant non-overlapped communication time.*

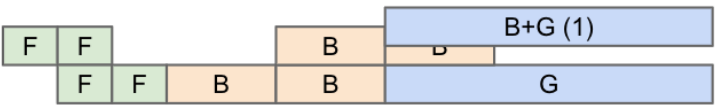


Figure 12.12: An example 2 stage, 2 microbatch pipeline. F denotes a stage forward pass and B is a stage backward pass (2x the cost). G denotes the data-parallel AllReduces, which can be significantly longer than the time of a single microbatch.

(3) **Pipeline bubbles and step imbalance:** As you can see in the (bad) pipeline schedule above, it is easy to have significant bubbles (meaning wasted compute) during a naive pipeline schedule. Above, the second stage is idle on step 0, the first stage is idle from step 2 to 3, and the second stage is again idle on the last step. While we can avoid these somewhat with careful scheduling, we still often have some bubbles. We also have to pass activations from one stage to the next on the critical path, which can add overhead:

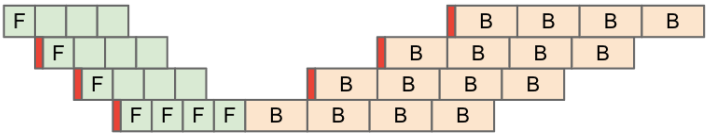


Figure 12.13: An example pipeline showing transfer cost in red. This shifts stages relative to each other and increases the pipeline bubble overhead.

There are workarounds for each of these issues, but they tend to be complicated to implement and difficult to maintain, but pipelining remains a technique with low communication cost relative to other methods.

Caveat about latency: As noted before, GPUs struggle to achieve full AllReduce bandwidth even with fairly large messages. This means even if we in theory can scale e.g. expert-parallel AllToAlls across multiple nodes, we may struggle to achieve even 50% of the total bandwidth. This means we do try to keep TP or EP within a smaller number of nodes to minimize latency overhead.

12.4.5 Examples

What does DeepSeek do? For reference, DeepSeek V3[24] is trained with 2048 H800 GPUs with:

- 64-way Expert Parallelism (EP) spanning 8 nodes
- 16-way Pipeline Parallelism (PP)
- 2-way ZeRO-1 Data Parallelism (DP)

They had a steady state batch size of `4096 * 15360 = 62,914,560` tokens, or 30k tokens per GPU. You can see that this is already quite large, but their model is also very sparse (k=8, E=256) so you need a fairly large batch size. You can see that

[24] See `https://arxiv.org/abs/2412.19437`

with 64-way EP and 16-way PP, we end up with 1024-way model parallelism in total, which means the AllReduce is done at the spine level, and because it's only 2-way, we end up with $2/(2-1) = 2$ times more bandwidth in practice. This also helps reduce the cost of the final data-parallel AllReduce overlapping with the final pipeline stages.

What does LLaMA-3 do? LLaMA-3 trains with a BS of 16M tokens on 16k GPUs, or about 1k tokens per GPU. They do:

- 8-way Tensor Parallelism within a node (TP)
- 16-way Pipeline Parallelism (PP)
- 128-way ZeRO-1 Data Parallelism

This is also a dense model so in general these things are pretty trivial. The 16-way PP reduces the cost of the data parallel AllReduce by 16x, which helps us reduce the critical batch size.

12.4.6 TLDR of LLM Scaling on GPUs

Let's step back and come up with a general summary of what we've learned so far:

- **Data parallelism or FSDP (ZeRO-1/3) requires a per-device batch size of about 2500 tokens per GPU**, although in theory in-network reductions + pure DP can reduce this somewhat.
- **Tensor parallelism is compute-bound up to about 8-ways** but we lack the bandwidth to scale much beyond this before becoming comms-bound. This mostly limits us to a single NVLink domain (i.e. single-node or need to use GB200NVL72 with to 72 GPUs).
- **Any form of model parallelism that spans multiple nodes can further reduce the cost of FSDP**, so we often want to mix PP + EP + TP to cross many nodes and reduce the FSDP cost.

- **Pipeline parallelism works well if you can handle the code complexity of zero-bubble pipelining and keep batch sizes fairly large to avoid data-parallel bottlenecks.** Pipelining usually makes ZeRO-3 impossible (since you would need to AllGather on each pipeline stage), but you can do ZeRO-1 instead.

At a high level, this gives us a recipe for sharding large models on GPUs:

- For relatively small dense models, aggressive FSDP works great if you have the batch size, possibly with some amount of pipelining or tensor parallelism if needed.
- For larger dense models, some combination of 1-2 node TP + many node PP + pure DP works well.
- For MoEs, the above rule applies but we can also do expert parallelism, which we prefer to TP generally. If $F > 8 * C/W_{\text{node}}$, we can do a ton of multi-node expert parallelism, but otherwise we're limited to roughly 2-node EP.

12.4.7 Quiz 5: LLM rooflines

Question 1 [B200 rooflines]: A B200 DGX SuperPod (**not GB200 NVL72**) has 2x the bandwidth within a node (900GB/s egress) but the same amount of bandwidth in the scale-out network (400GB/s) (source: `docs.nvidia.com/dgx-superpod/reference-architecture-scalable-infrastructure-b200/latest/ network-fabrics.html`). The total FLOPs are reported above. How does this change the model and data parallel rooflines?

Question 2 [How to shard LLaMA-3 70B]: Consider LLaMA-3 70B, training in bfloat16 with fp32 optimizer state with Adam.

1. At a minimum, how many H100s would we need simply to store the weights and optimizer?

2. Say we want to train on 4096 H100 GPUs for 15T tokens. Say we achieved 45% MFU (Model FLOPs Utilization). How long would it take to train?

3. LLaMA-3 70B has `F = 28,672` and was trained with a batch size of about 4M tokens. What is the most model parallelism we could do without being bandwidth-bound? With this plus pure DP, could we train LLaMA-3 while staying compute-bound on 4k chips? What about ZeRO-3? What about with 8-way pipelining? *Note: consider both the communication cost and GPU memory usage.*

Question 3 [Megatron-LM hyperparams]: Consider this figure from the Megatron-LM repository[25] highlighting their high MFU numbers.

Model size	Attention heads	Hidden size	Number of layers	Tensor MP size	Pipeline MP size	Data-parallel size	Number of GPUs	Batch size	Per-GPU teraFLOP/s	MFU	Aggregate petaFLOP/s
1.7B	16	2048	24	1	1	48	48	192	408.8	41%	19.6
7.1B	32	4096	30	2	1	48	96	192	465.9	47%	44.7
16B	48	6144	32	4	1	48	192	192	489.1	49%	93.9
32B	56	7168	48	8	1	48	384	192	459.6	46%	176.5
70B	64	8192	84	8	2	48	768	384	419.7	42%	322.3
119B	80	10240	92	8	4	48	1536	768	420.5	43%	645.9
177B	96	12288	96	8	6	48	2304	1152	432.8	44%	997.2
314B	128	16384	96	8	8	48	3072	1536	474.4	48%	1457.4
462B	144	18432	112	8	16	48	6144	3072	459.9	47%	2825.6

Figure 12.14: Megatron-LM hyperparameters showing high MFU configurations.

Note that their sequence length is 4096 everywhere. For the 16B, 70B, and 314B models, what is the per-device token batch size? Assuming data parallelism is the outermost axis and assuming bfloat16 reductions, determine whether each of these is theoretically compute-bound or bandwidth-bound, and whether there is a more optimal configuration available?

[25] See `https://github.com/NVIDIA/Megatron-LM`

12.5 Acknowledgements and Further Reading

This chapter relied heavily on help from many knowledgeable GPU experts, including:

- Adam Paszke, who helped explain the realities of kernel programming on GPUs.
- Swapnil Patil, who first explained how GPU networking works.
- Stas Bekman, who pointed out that the empirical realities of GPUs are often different from the purported specs.
- Reiner Pope, who helped clarify how GPUs and TPUs compare at a hardware level.
- Frédéric Bastien, who gave detailed feedback on the chip-level story.
- Nouamane Tazi, whose experience with LLM training on GPUs helped improve the roofline section.
- Sanford Miller, who helped me understand how GPUs are networked and how NVIDIA's specifications compare to what's often deployed in the field.

There's a great deal of good reading on GPUs, but some of my favorites include:

- **SemiAnalysis' History of the NVIDIA Tensor Core**[26]: a fantastic article describing how GPUs transformed from video game engines to ML accelerators.

[26] See: `semianalysis.com/2025/06/23/nvidia-tensor-core-evolution-from-volta-to-blackwell/`

- **SemiAnalysis' Analysis of Blackwell Performance**[27]: worth reading to understand the next generation of NVIDIA GPUs.

- **H100 DGX SuperPod Reference**[28]: dry but useful reading on how larger GPU clusters are networked. A similar document about the GB200 systems is also available.[29]

- **Hot Chips Talk about the NVLink Switch**[30]: fun reading about NVLink and NCCL collectives, especially including in-network reductions.

- **DeepSeek-V3 Technical Report**[31]: a good example of a large semi-open LLM training report, describing how they picked their sharding setup.

- **How to Optimize a CUDA Matmul**[32]: a great blog describing how to implement an efficient matmul using CUDA Cores, with an eye towards cache coherence on GPU.

- **HuggingFace Ultra-Scale Playbook**[33]: a guide to LLM parallelism on GPUs, which partly inspired this chapter.

- **Making Deep Learning Go Brrrr From First Principles**[34]: a more GPU and PyTorch-focused tutorial on LLM rooflines and performance engineering.

[27] See: `semianalysis.com/2024/04/10/nvidia-blackwell-perf-tco-analysis/`

[28] See NVIDIA documentation: `docs.nvidia.com/ dgx-superpod-reference-architecture-dgx-h100.pdf`

[29] See: `docs.nvidia.com/dgx-superpod/reference-architecture-scalable-infrastructure-gb200/ latest/network-fabrics.html#compute-fabric-576`

[30] See: `hc34.hotchips.org/assets/program/conference/day2/Network%20and%20Switches/NVSwitch%20HotChips%202022%20r5.pdf`

[31] arXiv preprint: `arxiv.org/pdf/2412.19437`

[32] See: `siboehm.com/articles/22/CUDA-MMM`

[33] See: `huggingface.co/spaces/nanotron/ultrascale-playbook`

[34] See: `horace.io/brrr_intro.html`

- **Cornell Understanding GPU Architecture site**[35]: a similar guide to this book, comparing GPU and CPU internals more specifically.

12.6 Appendix A: How does this change with GB200?

Blackwell introduces a bunch of major networking changes, including NVLink 5 with twice the overall NVLink bandwidth (900GB/s). B200 still has 8-GPU nodes, just like H100s, but GB200 systems (which combine B200 GPUs with Grace CPUs) introduce much larger NVLink domain (72 GPUs in NVL72 and in theory up to 576). This bigger NVLink domain also effectively increases the node egress bandwidth, which reduces collective costs above the node level.

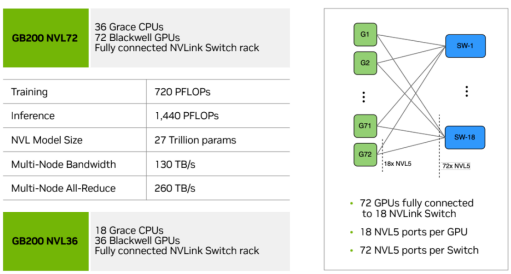


Figure 12.15: A diagram showing how a GB200 NVL72 unit is constructed, with 18 switches and 72 GPUs.

Within a node, this increased bandwidth (from 450GB/s to 900GB/s) doesn't make much of a difference because we also double the total FLOPs/s of each GPU. Our rooflines mostly stay the same, although because NVLink has much better bandwidth, Expert Parallelism becomes easier.

[35]See: `cvw.cac.cornell.edu/gpu-architecture`

Beyond a node, things change more. Here's a SuperPod diagram.[36]

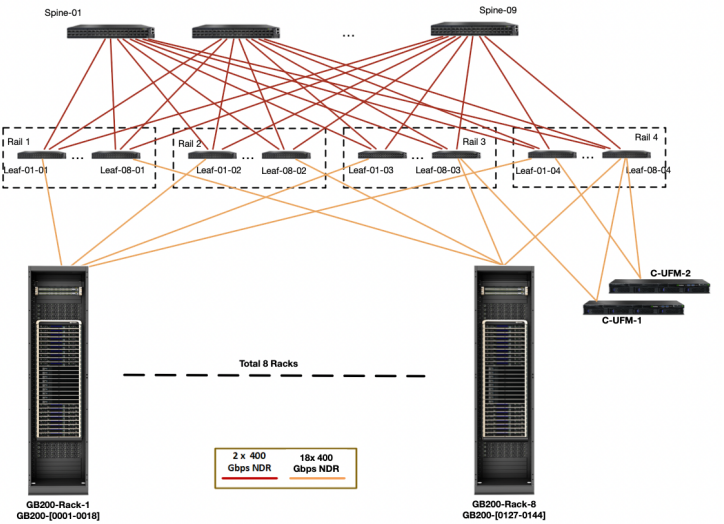


Figure 12.16: A diagram showing a GB200 DGX SuperPod of 576 GPUs.

As you can see, the per-node egress bandwidth increases to $4 \times 18 \times 400/8 = 3.6$TB/s, up from 400GB/s in H100. This improves the effective cross-node rooflines by about 4x since our FLOPs/chip also double. Now we may start to worry about whether we're bottlenecked at the node level rather than the scale-out level.

Grace Hopper: NVIDIA also sells GH200 and GB200 systems which pair some number of GPUs with a Grace CPU. For instance, a GH200 has 1 H200 and 1 Grace CPU, while a GB200

[36] See NVIDIA documentation: `docs.nvidia.com/dgx-superpod/reference-architecture-scalable-infrastructure-gb200/latest/network-fabrics.html`

system has 2 B200s and 1 Grace CPU. An advantage of this system is that the CPU is connected to the GPUs using a full bandwidth NVLink connection (called NVLink C2C), so you have very high CPU to GPU bandwidth, useful for offloading parameters to host RAM. In other words, for any given GPU, the bandwidth to reach host memory is identical to reaching another GPU's HBM.

12.7 Appendix B: More networking details

Here's a diagram of an NVLink 4 switch. There are 64 overall NVLink4 ports (each uses 2 physical lanes), and a large crossbar that handles inter-lane switching. TPUs by contrast use optical switches with mirrors that can be dynamically reconfigured.

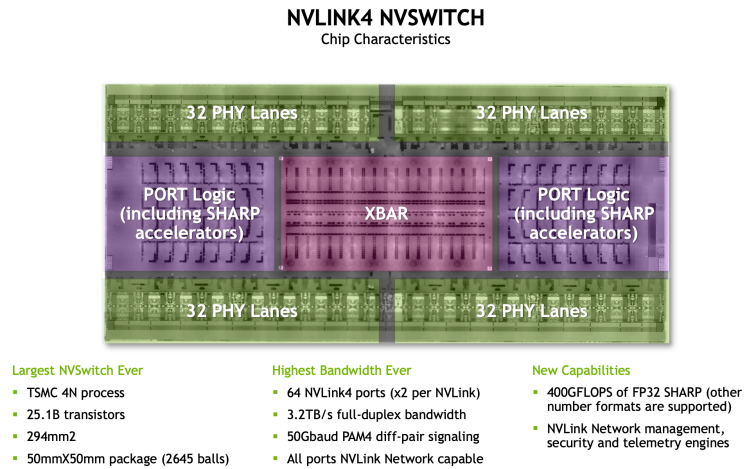


Figure 12.17: A lower level view of a single NVLink4 Switch.

At each level we can be bottlenecked by the available link bandwidth or the total switch bandwidth.

- **Node level:** at the node level, we have $4 \times 1.6\text{TB/s} = 6.4\text{TB/s}$ of NVSwitch bandwidth, but each of our 8 GPUs can only egress 450GB/s into the switch, meaning we actually have a peak bandwidth of $450\text{e}9 \times 8 = 3.6\text{TB/s}$ (full-duplex) within the node.

- **SU/leaf level:** at the SU level, we have 8 switches connecting 32 nodes in an all-to-all fashion with 1x400 Gbps Infiniband. This gives us $8 \times 32 \times 400/8 = 12.8\text{TB/s}$ of egress bandwidth from the nodes, and we have $8 \times 1.6\text{TB/s} = 12.8\text{TB/s}$ at the switch level, so both agree precisely.

- **Spine level:** at the spine level, we have 16 switches connecting 32 leaf switches with 2x400 Gbps links, so we have $32 \times 16 \times 400 \times 2/8 = 51.2\text{TB/s}$ of egress bandwidth. The 16 switches give us $16 \times 1.6\text{TB/s} = 25.6\text{TB/s}$ of bandwidth, so this is the bottleneck at this level.

Per GPU, this gives us 450GB/s of GPU to GPU bandwidth at the node level, 50GB/s at the SU level, and 25 GB/s at the spine level.

GPU empirical AR bandwidth:

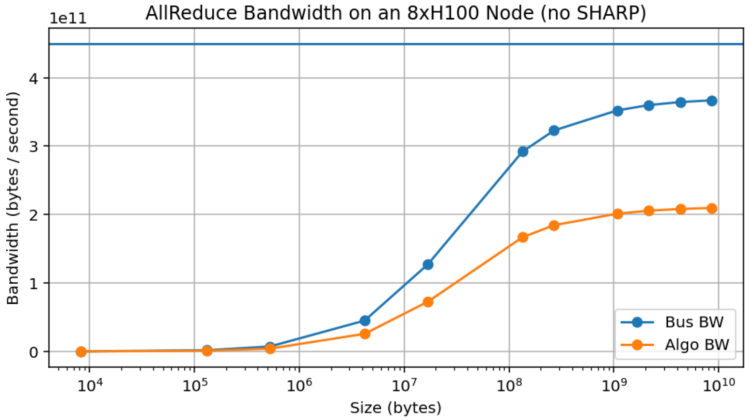


Figure 12.18: AllReduce bandwidth on an 8xH100 cluster (intra-node, SHARP disabled).

TPU v5p bandwidth (1 axis):

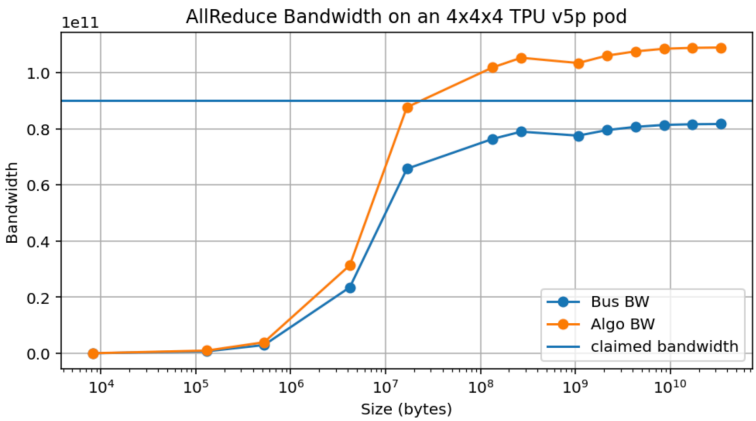


Figure 12.19: AllReduce bandwidth on a TPU v5p 4x4x4 cluster (along one axis).

Here's AllGather bandwidth as well:

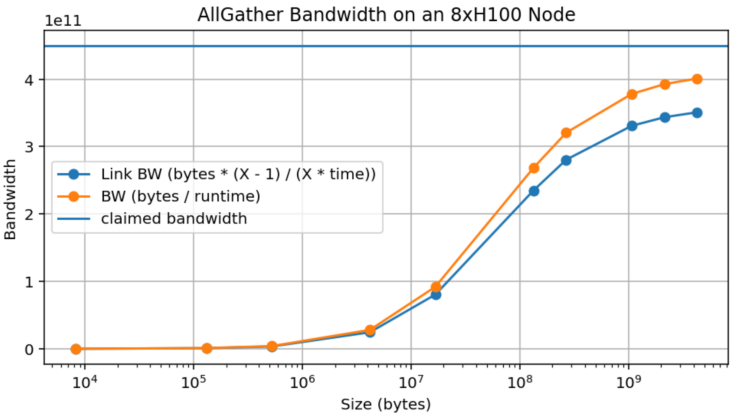


Figure 12.20: AllGather bandwidth on an 8xH100 cluster (intra-node).

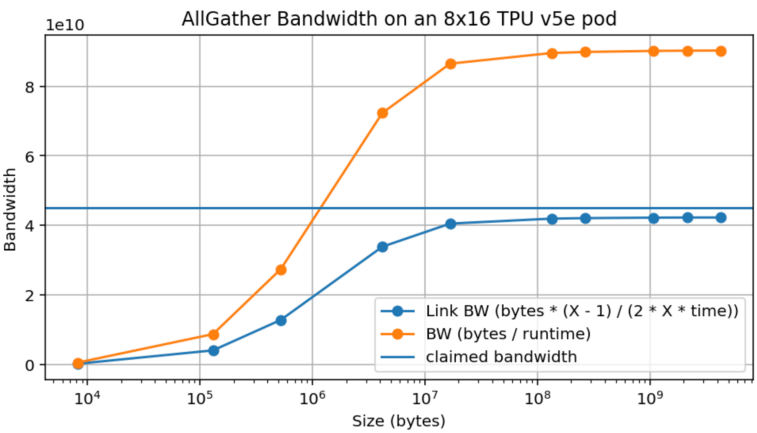


Figure 12.21: AllGather bandwidth on a TPU v5e 8x16 cluster (along one axis).

More on AllToAll costs:

Here we can compare the approximation $\min(K/Z) \cdot (Z-1)/Z$ to the true value of $(1-((Z-1)/Z)^K) \cdot (Z-1)/Z$. They're similar except for small values of Z.

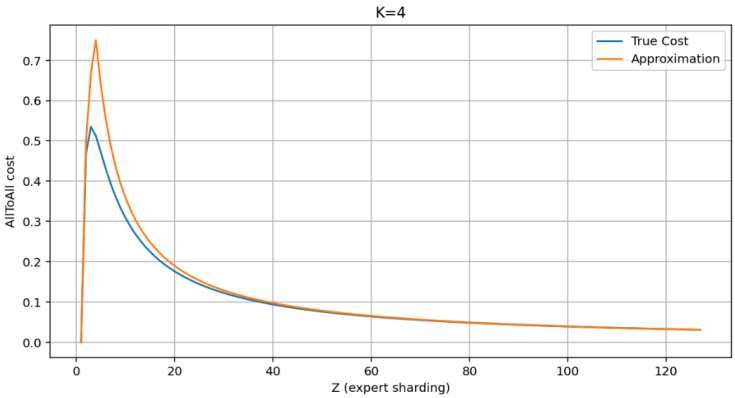


Figure 12.22: A comparison of the approximate and true cost of a ragged AllToAll as the number of shards increases.

Bibliography

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv [cs.CL]*, May 2023.

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason Lee, Deming Chen, and Tri" Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv [cs.LG]*, January 2024.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv [cs.CL]*, February 2023.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J L Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R J Chen, R L Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S S Li, Shanghao Lu,

Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T Wang, Tao Yun, Tian Pei, Tianyu Sun, W L Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X Q Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y K Li, Y Q Wang, Y X Wei, Y X Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z F Wu, Z Z Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. Deepseek-v3 technical report. *arXiv [cs.CL]*, December 2024.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller,

Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui

Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery,

Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy,

Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. The llama 3 herd of models. *arXiv [cs.AI]*, July 2024.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. *arXiv [cs.LG]*, (1), September 2023.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. *arXiv [cs.LG]*, November 2022.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle: Speculative sampling requires rethinking feature uncertainty. *arXiv [cs.LG]*, January 2024.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *arXiv [cs.LG]*, November 2022.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. *arXiv [cs.LG]*, October 2019.

Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv [cs.NE]*, November 2019.

Noam Shazeer. Glu variants improve transformer. *arXiv [cs.LG]*, February 2020.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv [cs.LG]*, January 2017.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv [cs.CL]*, September 2019.