How to Think About TPUs

Part 2 of How To Scale Your Model (Part 1: Rooflines | Part 3: Sharding)

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

AUTHORS

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

AFFILIATION

Google DeepMind

PUBLISHED

Feb. 4, 2025

What 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) [1] **.** Here's a diagram:

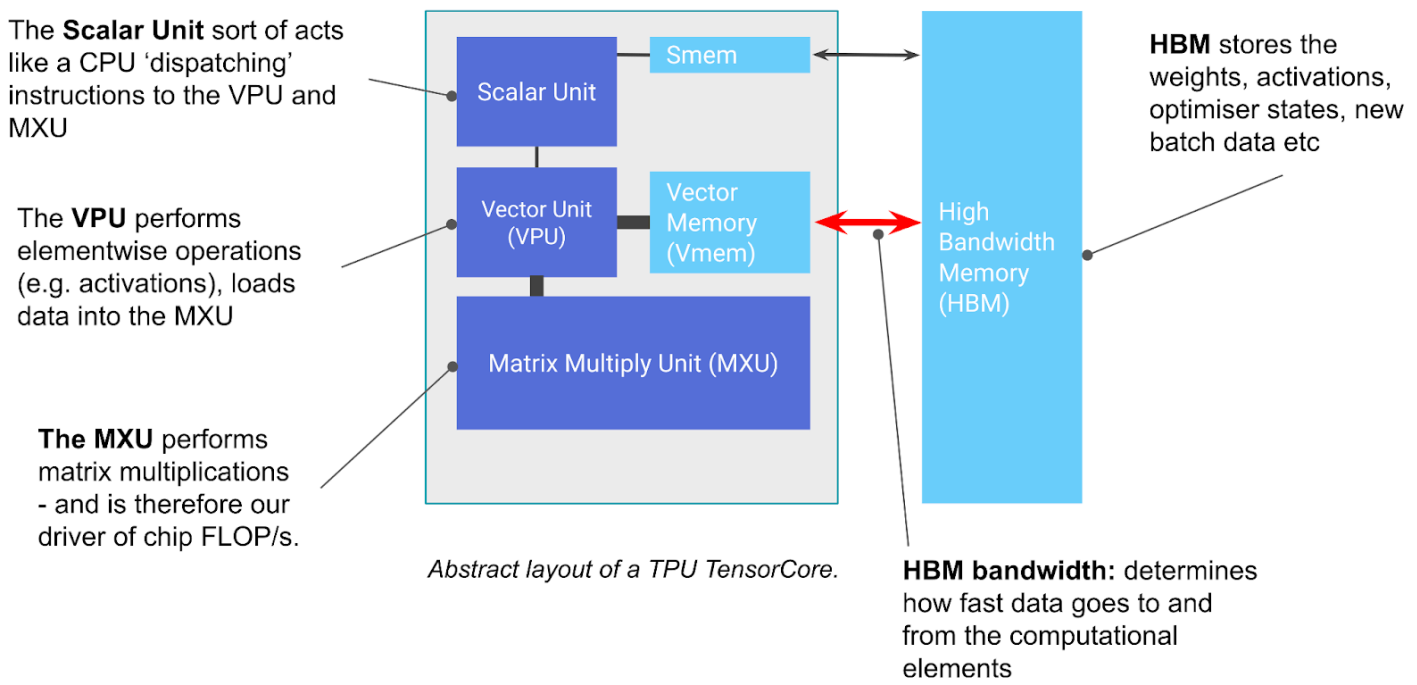


Figure: the basic components of a TPU chip. The TensorCore is the gray left-hand box, containing the matrix-multiply unit (MXU), vector unit (VPU), and vector memory (VMEM).

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 C provides more details.
- **VMEM** (Vector Memory) is an on-chip scratchpad located in the TensorCore, close to the compute units. It is much smaller than HBM (for example, 128 MiB on TPU v5e) but has a much higher bandwidth to the MXU. VMEM operates somewhat like an L1/L2 cache on CPUs but is much larger and programmer-controlled. Data in HBM needs to be copied into VMEM before the TensorCore can do any computation with it.

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

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

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

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

Here's an example of how you might perform an elementwise product from HBM:

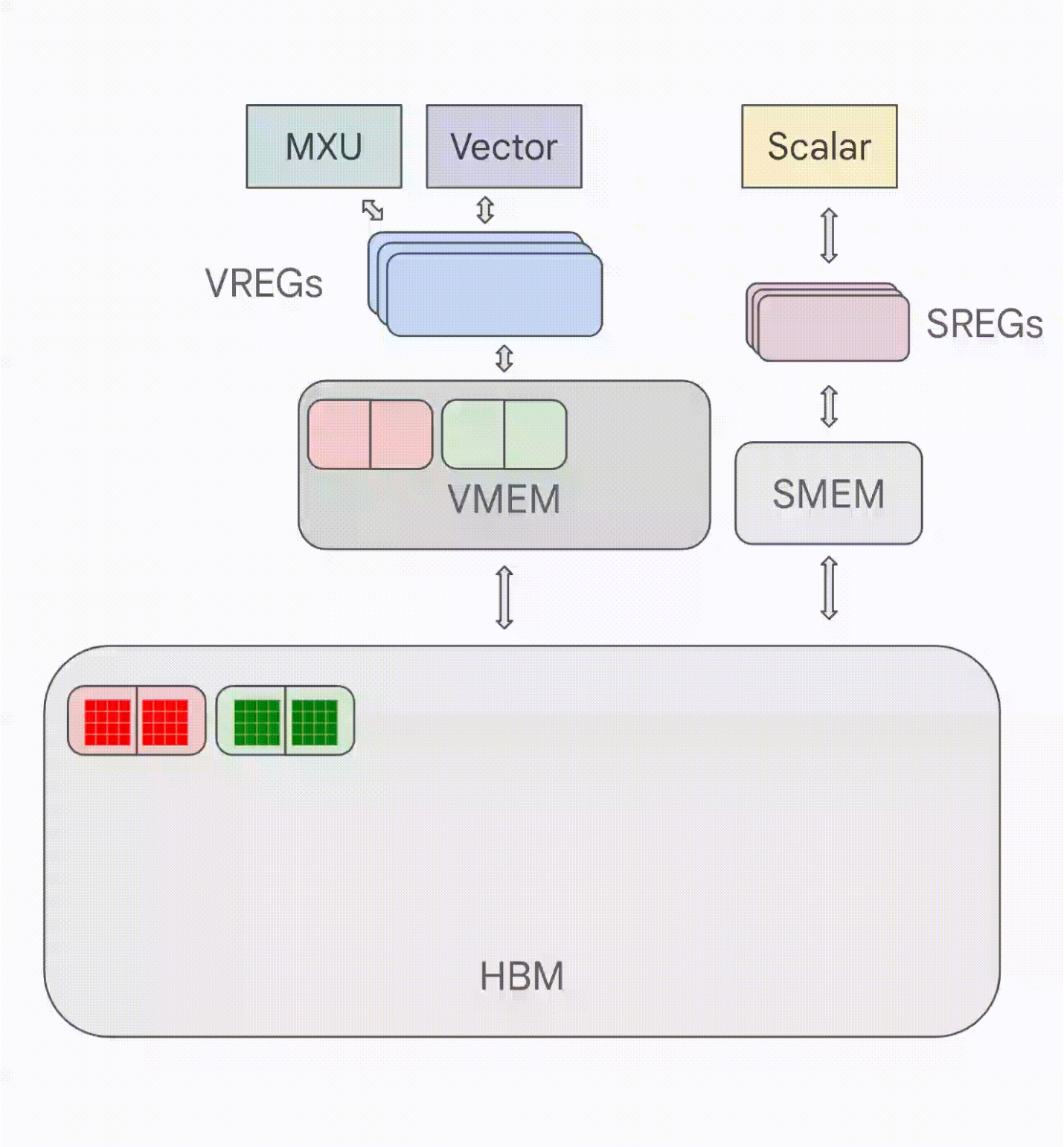
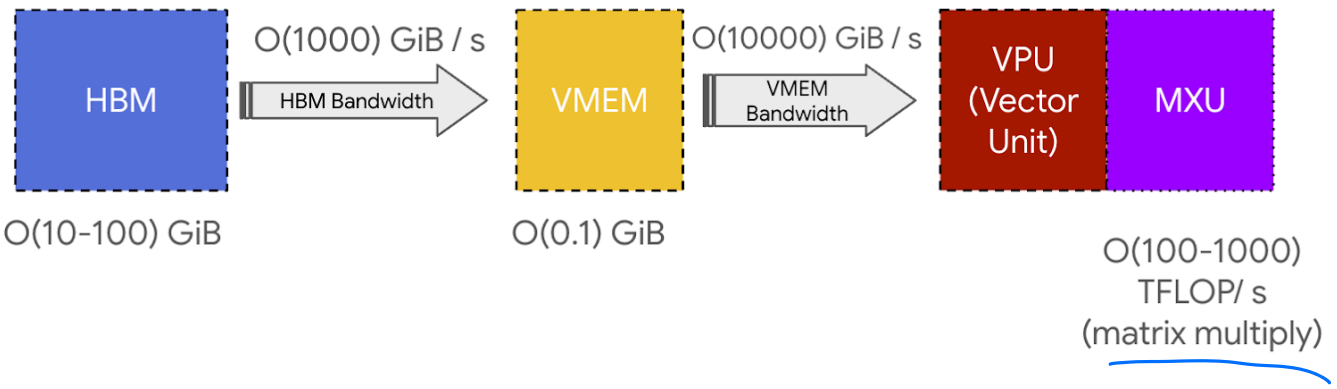


Figure: an animation showing a pointwise product performed on TPU, with bytes loaded from HBM. Note how bytes are streamed out of memory in chunks and partial results are pipelined back without waiting for the full array to be materialized.

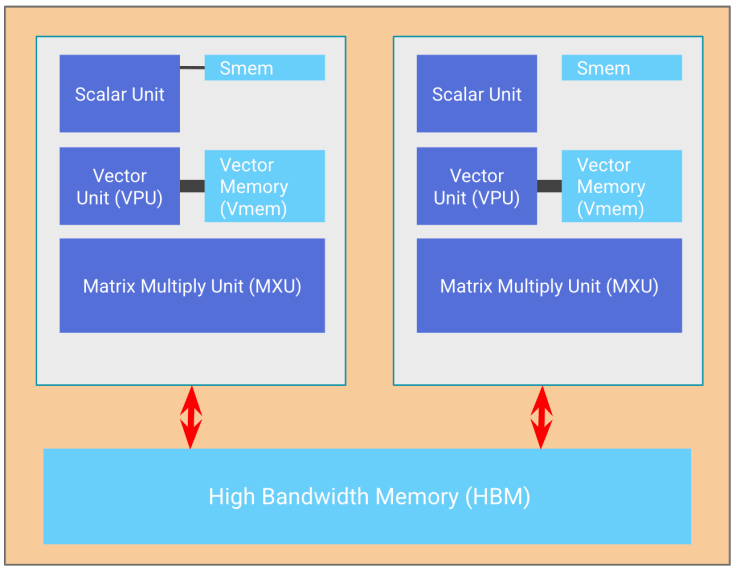
A matmul would look nearly identical except it would load into the MXU instead of the VPU/Vector unit, and the loads and stores would occur in a different order, since the same weight chunk is used for multiple chunks of activations. You can see chunks of data streaming into VMEM, then into the VREGs (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 ↔ VMEM and VMEM ↔ systolic array bandwidths set fundamental limits on what computations TPUs can do efficiently.

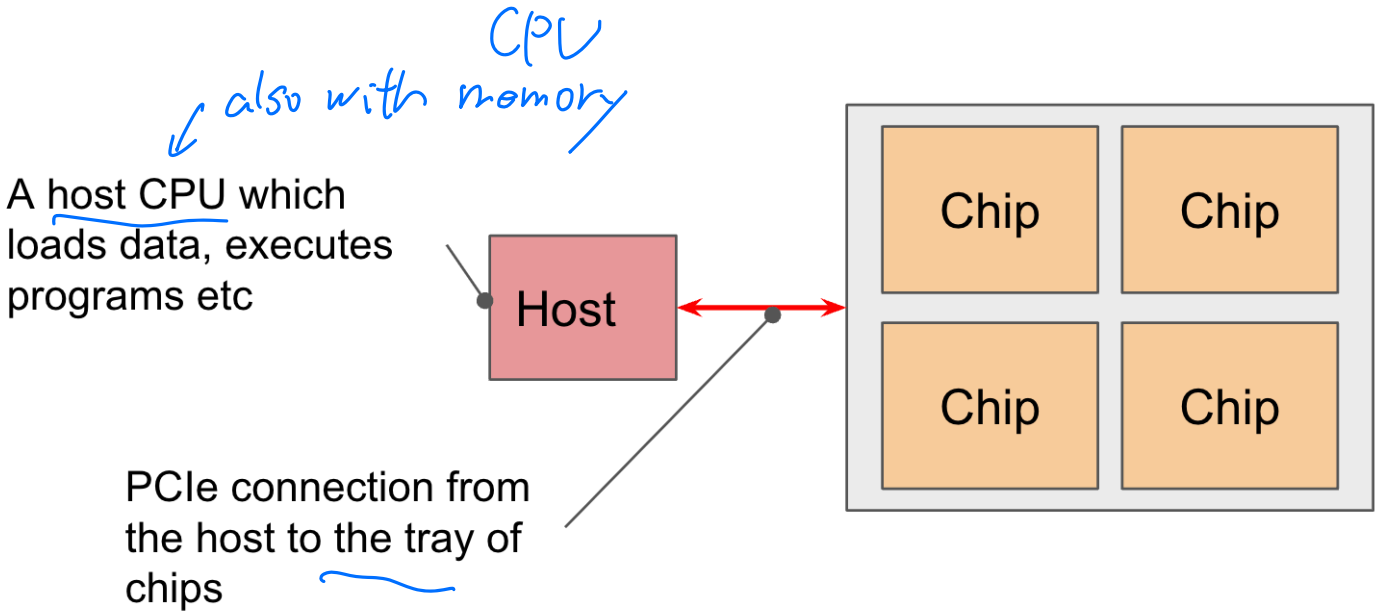
VMEM and arithmetic intensity: VMEM is much smaller than HBM but it has a much higher bandwidth to the MXU. As we saw in Section 1, this means if an algorithm can fit all its inputs/outputs in VMEM, it's much less likely to hit communication bottlenecks. This is particularly helpful when a computation has poor arithmetic intensity: VMEM bandwidth is around 22x higher than HBM bandwidth which means an MXU operation reading from/writing to VMEM requires an arithmetic intensity of only 10-20 to achieve peak FLOPs utilization. That means if we can fit our weights into VMEM instead of HBM, our matrix multiplications can be FLOPs bound at much smaller batch sizes. And it means algorithms that fundamentally have a lower arithmetic intensity can still be efficient. VMEM is just so small this is often a challenge.[3]



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



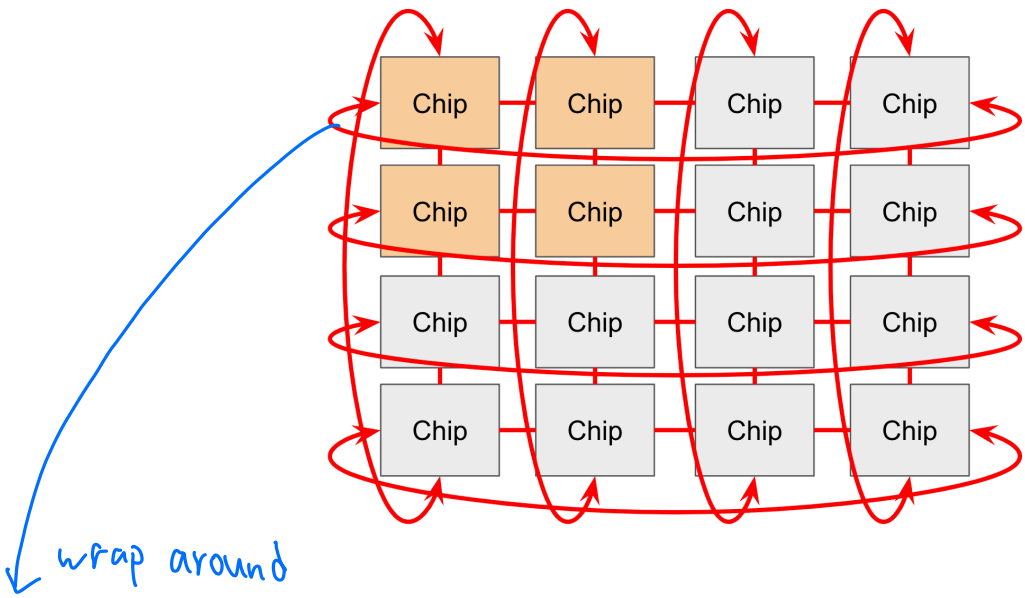
Chips are arranged in **sets of 4 on a 'tray'** connected to a **CPU host via PCIe network.** This is the format most readers will be familiar with, 4 chips (8 cores, though usually treated as 4 logical megacores) exposed through Colab or a single TPU-VM. For inference chips like the TPU v5e, we have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



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.

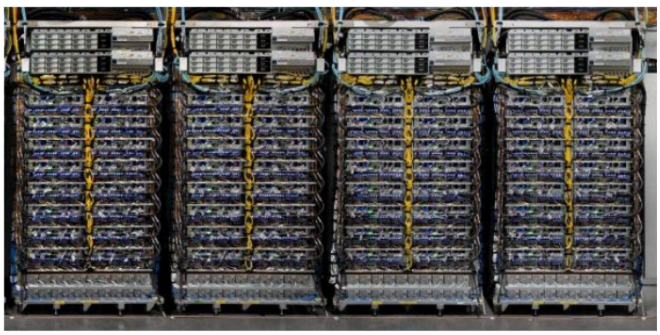
TPU Networking

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.



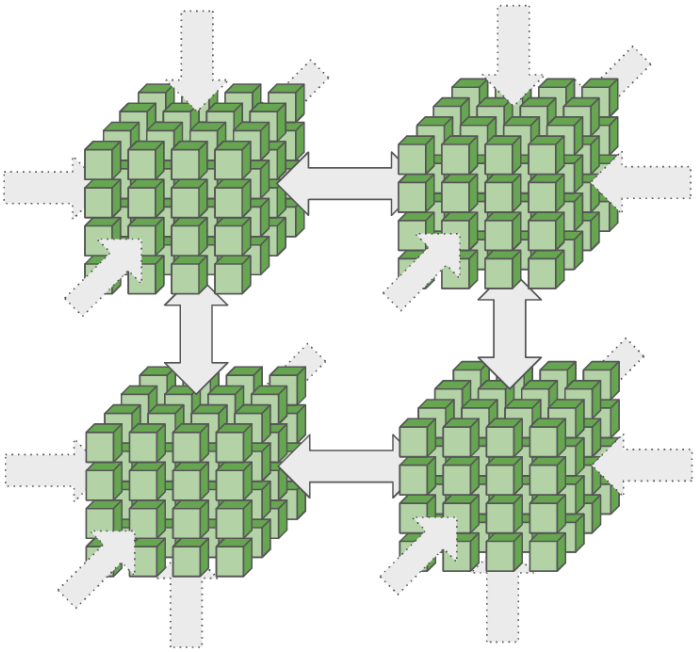
The toroidal structure reduces the maximum distance between any two nodes from N to $N/2$, making communication much faster. TPUs also have a "twisted torus" configuration that wraps the torus in a Mobius-strip like topology to further reduce the average distance between nodes.

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.

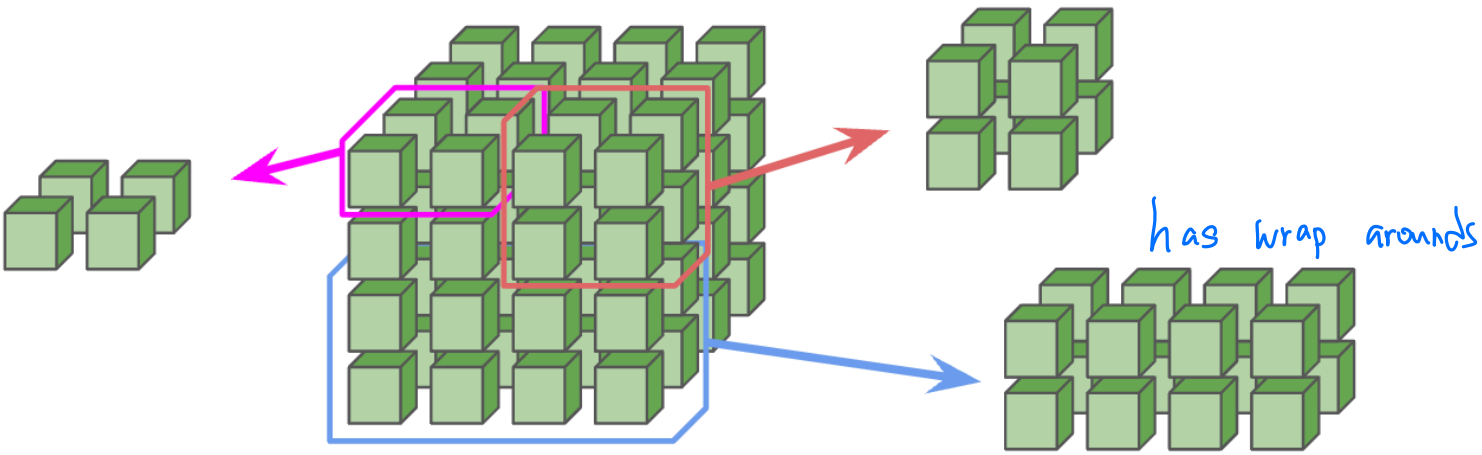


Each rack corresponds to 4x4x4 TPUv4s

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

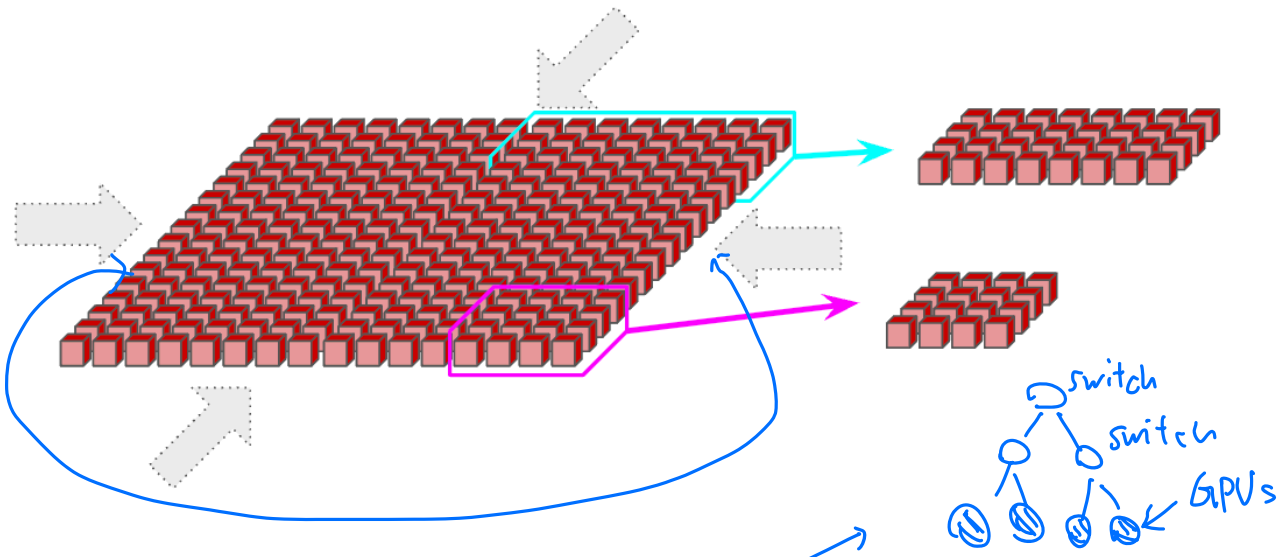


Smaller topologies (e.g. `2x2x1`, `2x2x2`) can also be requested, albeit with no wraparounds. This is an important caveat, since it typically doubles the time of most communication. Any multiple of a full cube (e.g. `4x4x4` or `4x4x8`) will have wraparounds provided by the optical switches.[6]



Smaller slices can also be requested.

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



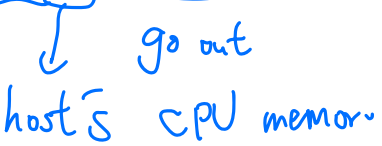
This nearest-neighbor connectivity is a key difference between TPUs and GPUs. GPUs are connected with a hierarchy of switches that approximate a point-to-point connection between every GPU, rather than using local connections like a TPU. Typically, GPUs within a node (8 GPUs for H100 or as many as 500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

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

- `2.5e12` bytes/s (2.5 TB/s) of HBM bandwidth per chip.
- `9e10` bytes/s (90^7 GB/s) of ICI bandwidth per axis, with 3 axes per chip.
- `2.5e10` bytes/s (25 GB/s) of DCN (egress) bandwidth per host. Since we typically have 8 TPUs per host, this is really closer to `3.1e9` bytes / s / chip.

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

Model	Pod size	Host size	HBM capacity/chip	HBM BW/chip (bytes/s)	FLOPs/s/chip (bf16)	FLOPs/s/chip (int8)
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.[8]

PCIe bandwidth is typically around `1.5e10` bytes / second per chip[9], while DCN bandwidth is typically around `2.5e10` bytes / second per host. We include both unidirectional and bidirectional bandwidth for completeness. Typically bidirectional bandwidth is the more useful number when we have access to a full wraparound ring, while one-way bandwidth is more true to the hardware.

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.*

▶ Click here for the answer.

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.

▶ Click here for the answer.

Question 3 [PCIe operational intensity]: Imagine we're forced to store a big weight matrix A of type $\text{bfloat16}[D, F]$, and a batch of activations x of type $\text{bfloat16}[B, D]$ 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.

▶ Click here for the answer.

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 TPUv5e to start.

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

▶ Click here for the answer.

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?

▶ Click here for the answer.

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.*

▶ Click here for the answer.

That's it for Part 2! For Part 3, covering partitioning and cross-TPU communication, click here.

Appendix

Appendix A: All about GPUs

Since the Volta generation (V100), TPUs and GPUs have started to looked a lot alike: *they both aim to do matrix multiplication very fast*. They both act as an accelerator attached to a CPU and many components are roughly analogous (don't worry if you don't know all the terminology, we'll introduce them all later):

TPU	GPU
Tensor Core	SM ("Streaming Multiprocessor")
HBM	DRAM
VMEM	SMEM (often used as an L1 cache)
VPU	Warp scheduler (a set of SIMD CUDA cores)
MXU	Tensor Core
ICI	NVLink/NVSwitch

The core unit of a GPU is an SM, or "streaming multiprocessor", which is roughly analogous to the whole TPU Tensor Core described above. Compared to TPUs, though, GPUs have *many* more of them (an H100 has about 144). Each SM has its own matrix multiplication unit, confusingly called a Tensor Core, which acts like the TPU MXU, and a set of 4 narrow SIMD units called Warp schedulers that act like the TPU VPUs (with 32 lanes instead of 1024). More independent SMs makes computation more flexible (since each can do totally independent work) but also makes the hardware more expensive and complex to reason about.

Figure: the basic components of a Blackwell (B100) SM. The diagram shows 4 SIMD compute units (which we call warp schedulers), each with a Tensor Core for matrix multiplication. This also shows per-warp scheduler registers, an SM-level L1 cache, and TMEM or tensor memory which is a new addition in Blackwell.

Each SM also has an O(256kB) L1 cache (also called SMEM) used to speed data access and for register spilling. A section of the memory used for the L1 cache can also be declared as shared memory allowing access from any thread in the thread-block, and is used for user-defined caches, parallel reductions and synchronization, etc. (similar to VMEM on a TPU).

GPUs also have an additional L2 cache that is shared by all SMs. Unlike VMEM, this is hardware managed and optimizing cache hits is often important for performance.

Networking:

- Primary difference is that NVIDIA GPUs are typically in 'cliques' of 8-256 GPUs via switches (NVLink → NVSwitch), which allow for point-to-point communication between any GPU within that 'clique', but that means communication between more than 256 is significantly slower - this means training on more than 256 typically requires pipeline parallelism to scale, which is more complex (by contrast, PaLM was trained on two cliques of 3072 TPU chips each).
- For common neural net operations such as AllReduce, all-to-all connections do not hold an advantage (as the same communication patterns must occur regardless), but it does allow for storing MoE models across more GPUs and transmitting the experts around more efficiently.
- Each GPU requires a switch that costs similar to the GPU itself, making on chip interconnect like ICI cheaper.
- NVIDIA deep learning performance
- NVSwitch
- Very different Tensor Parallelism / Pipeline Parallelism transition point!

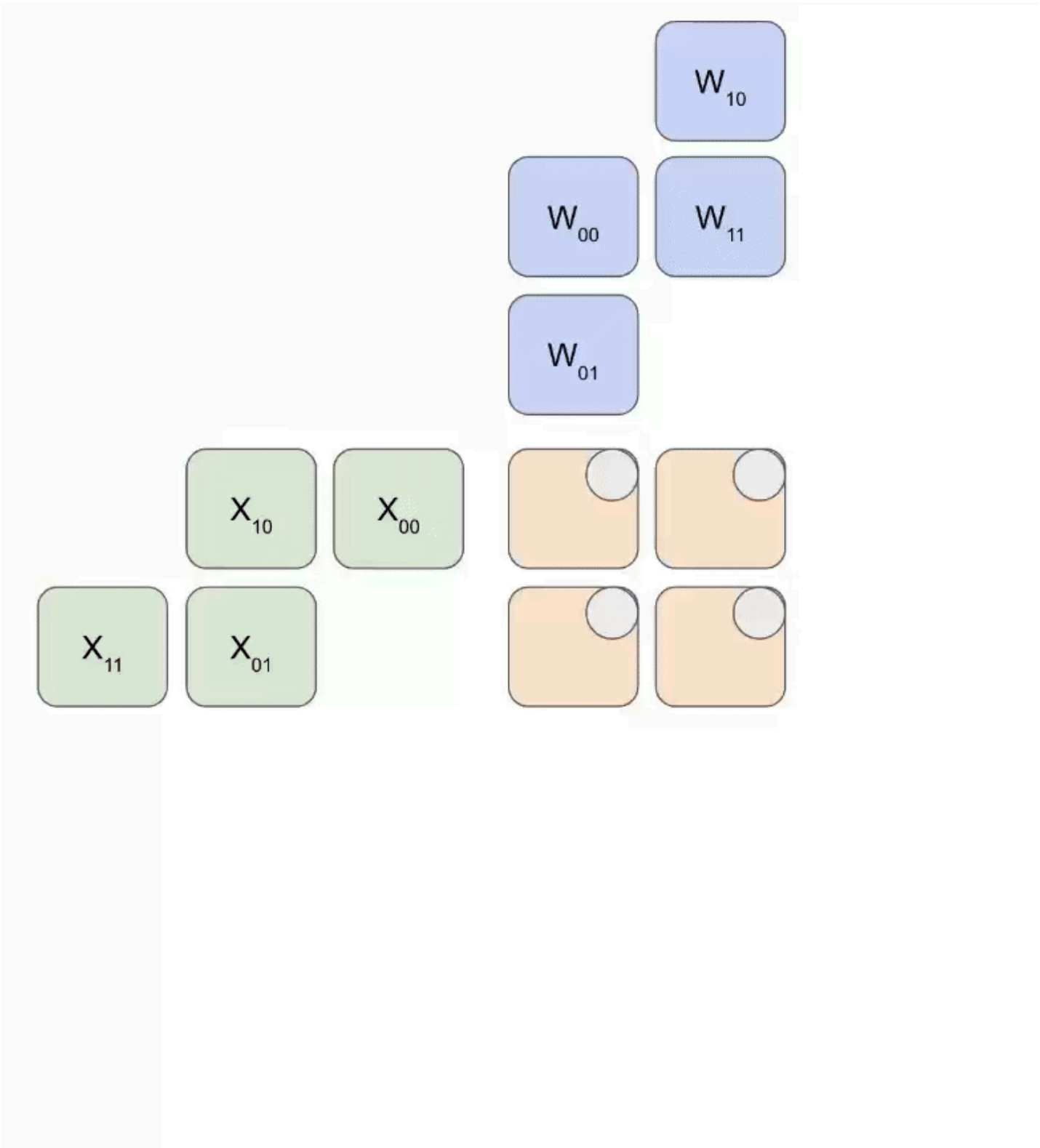
Appendix B: How does a systolic array work?

At the core of the TPU MXU is a `128x128` systolic array (`256x256` on TPU v6e). When fully saturated the systolic array can perform one `bfloat16[8,128] @ bf16[128x128] -> f32[8,128]` [10] multiplication per 8 clock cycles.

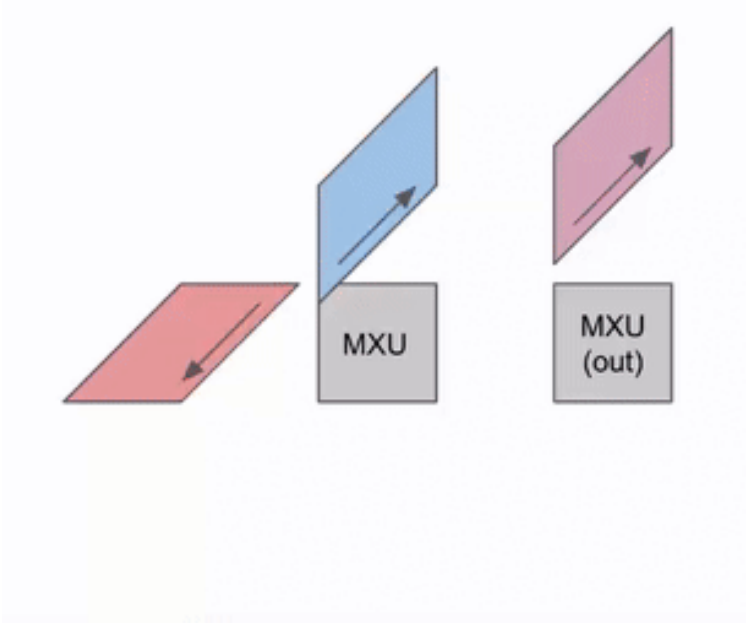
- At its core, the systolic array is a 2D `128x128` (`=16,384`) grid of ALUs each capable of performing a multiply and add operation.

- 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).

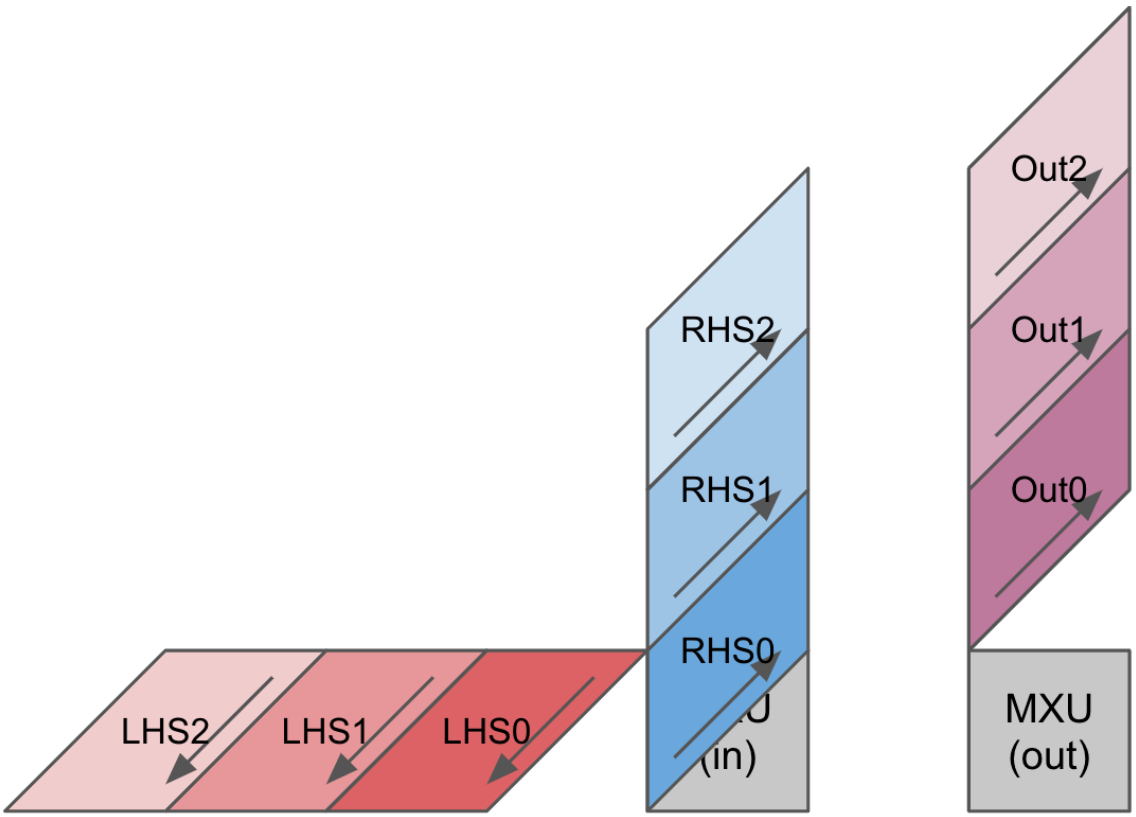
Here is a simplified animation of multiplying a set of weights (blue) with a set of activations (green). You'll notice that the weights (RHS) are partially loaded first, diagonally, and then the activations are fed in, also diagonally. In each frame below, we multiply all the overlapped green and blue 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:

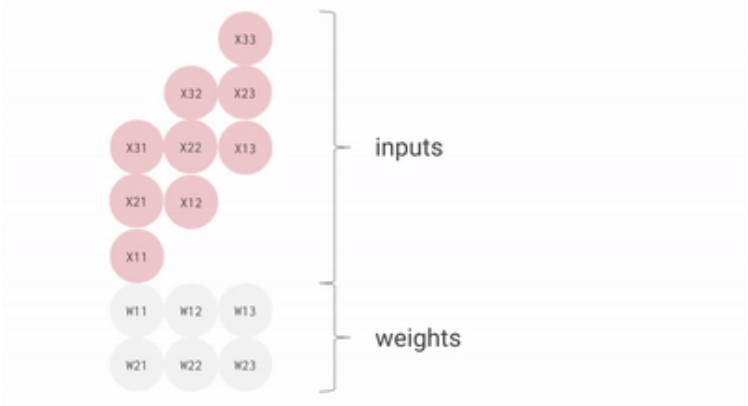


Here's a diagram showing how this can be pipelined across multiple RHS and LHS arrays:



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.

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

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

Appendix C: More on TPU internals

Scalar Core

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

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

VPU

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

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

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

Footnotes

1. TPU v6e (Trillium) has a 256x256 MXU, while all previous generations use 128x128 [↩]
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. [↩]
3. We sometimes talk about VMEM prefetching, which refers to loading weights ahead of time in VMEM so we can mask the cost of loading for our matmuls. For instance, in a normal Transformer we can sometimes load our big feed-forward weights into VMEM during attention, which can hide the cost of the weight load if we're memory bandwidth bound. This requires our weights to be small enough or sharded enough to fit a single layer into VMEM with space to spare. [↩]
4. On Cloud TPU VMs, each tray is exposed as part of a separate VM, so there are once again 4 cores visible. [↩]

5. The optical switch is simply a reconfigurable connection with the same ICI bandwidth. It just lets us connect cubes while retaining a wraparound link. [↩]
6. Note that a `2x2x4` won't have any wraparounds since they are provided by the optical switches which are only available on a full cube. A TPU v5e 8x16 _will_ have a wraparound on the longer axis, however, since it doesn't use reconfigurable optical networking. [↩]
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. 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. [↩]
9. Trillium (TPU v6e) has 32GB/s, about 2x higher than v5. [↩]
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. [↩]

References

1. **TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings**
   Jouppi, N.P., Kurian, G., Li, S., Ma, P., Nagarajan, R., Nai, L., Patil, N., Subramanian, S., Swing, A., Towles, B., Young, C., Zhou, X., Zhou, Z. and Patterson, D., 2023. arXiv [cs.AR].

Miscellaneous

*Work done at Google DeepMind, now at MatX.

Citation

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

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

or as a BibTeX entry:

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

2 reactions

❤️ 1 🚀 1

18 comments · 27 replies — *powered by giscus*

Oldest Newest

mitchellgoffpc Feb 4 Contributor

In the solution for question 5, it looks like bytes transferred should be 1.7e7 rather than 1.7e10, and transfer time should be 170µs rather than 170ms

↑ 1

1 reply

jacobaustin123 Feb 4 Collaborator

Very good catch, several parts of this answer are wrong. I've updated the repo with the fix, will take a few minutes to update the website.

Write a reply

kishorepv Feb 6

Multiplication is more expensive than an addition right? In multiplying a matrix when we say the number of operations is ~ 2 x B x D x F (BDF for multiplication, and B(D-1)F for addition)?, do we consider multiplication and addition as equally expensive ?

↑ 1

4 replies

jacobaustin123 Feb 6 Collaborator

The hardware typically treats them the same (each cycle it does an equal number of multiplies and adds to perform a matmul). They may require different numbers of transistors but the overall speed is the same.

With that said, the TPU MXU does matrix multiplications and not, say, pure sums/reductions. You can use the MXU speed to calculate how long matmuls will take, but you can just say "hey I want to do 1e10 adds, let me just divide that by the TPU/GPU bfloat16 FLOPs figures".

kishorepv Feb 6

Got it. Thanks.
If the hardware treats multiplication and additions as the same, then do techniques like Karatsuba / Toom-Cook algorithm (which reduces the number of multiplications at the cost of increased additions / subtractions) have any use in speeding up matmul on a GPU/TPU?

kerrickstaley Feb 7

@kishorepv: The Karatsuba algorithm is for multiplying integers. Did you mean the Strassen algorithm which is a fast algorithm for multiplying matrices?

👍 1

fedelebron Feb 7 Collaborator

You can implement Strassen, but trading what the TPU is excellent at (matrix multiplications) for something it's merely OK at (pointwise operations) might not be a net win, on matrix sizes you care about (i.e. before the asymptotics totally dominate). Sharding (the next chapter) is another way to speed up large matrix multiplications that scales with better constants, at the cost of, well, more hardware :)

👍 1

Write a reply

kerrickstaley Feb 7

In question 5, the answer mentions v4e; I think it should be v5e. Also, I don't understand why the total transfer time is multiplied by the number of hops. Streaming from one core to another should take num_bytes / bandwidth + latency_of_first_byte, which is approximately num_bytes / bandwidth when num_bytes is large. This means it should take about num_bytes / bandwidth = 170 us for the whole transfer.

↑ 1

1 reply

jacobaustin123 Feb 7 Collaborator

Thanks for catching this. This got changed around a few times and we missed this. Fixed.

Write a reply

kishorepv Feb 8

There is a typo in the line "In each frame above, 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...". It should be "In each frame below" instead of "In each frame above"

↑ 1

3 replies

burichh Feb 9

In the first gif in "Appendix B: How does a systolic array work?", the last image (can't embed into this message:( ) shows an incorrect calculation of the bottom left output value (in purple). It shows $W_{00} \cdot X_{01} + W_{10} \cdot X_{00}$ while actually it should be $W_{01} \cdot X_{01} + W_{00} \cdot X_{00}$.

It's a bit strange (but maybe only for me) that for both W and X it is the second index that runs across the multiplication, while in general the notation is such that the second and first indices are running:

$$(W \cdot X)_{nm} = \sum_i W_{ni} \cdot X_{im}$$

But that's actually just a question of notation.

burichh Feb 9

Ups, sorry, I wanted to add a new comment, not reply to yours. Upsie!

manavgarg Feb 15

I actually found this to be more intuitive when understanding systolic arrays.

Write a reply

Shua1 Feb 9

H100 spec says:

> INT8 Tensor Core* 3,958 TOPS
> FP16 Tensor Core* 1,979 teraFLOPS
> , meaining INT8 is 2x of FP16 flops. Simplistically, understandable since FP16 is 2x the size of INT8. Though I expect integer operation is faster than FP operations.

But the text sasy:

> This is about 5e13 bf16 FLOPs/s per MXU at 1.5GHz on TPU v5e.
> TPUs also support lower precision matmuls with higher throughput (e.g. each TPU v5e MXU can do 4e14 int8 OPs/s)

int8 is about 8x faster than FP16. Any idea why the difference between 2x on H100 and 8x on TPU v5e? Thanks!

↑ 1 👍 1 1 reply

jacobaustin123 Feb 10 Collaborator

This was supposed to say "each TPU v5e chip can do 4e14 FLOPs", i.e. it's only 2x the FLOPs of bfloat16. Fixed.

Write a reply

Kumawat-Akhilesh Feb 11

I am still grappling with the idea of when I should use TPU (say V6e) versus an Nvidia B200. The computation FLOPs/s is higher for B200 (4500 TFLOPs v 920 TFLOPs, bf16), HBM bandwidth is higher for B200 (8TB/s v 1.6TB/s), HBM size is higher for B200 (192GB v 32GB), interconnect is faster for B200 (NVlink at 1800GB/s v ICI BW at 180GB/s)...
I understand the idea of larger pods for TPUs but even if B200 connects to say 576 GPUs pod, beyond that they connect at 800GB/s InfiniBand or Ethernet and even this DCN is faster than the ICI speed of TPU (180GB/s)...

Hence, I am a bit confused on the tradeoff. Isn't B200 outperforming on all fronts (except costs perhaps due to switch costs)?
In fact, even H100 outperforms any TPU on all these metrics.. so I am not understanding when do TPUs outperform these GPUs.. for what workloads?

↑ 1 1 reply

jacobaustin123 Feb 11 Collaborator

I think the key is that nothing matters except performance / dollar. FLOPs / dollar. ICI bandwidth / dollar. It's relatively easy to package more FLOPs or bandwidth in a package. What's hard is making a product that does this cheaply. That's partly why Google's inference chips (e.g. TPU v5e) have worse specs than an H100 in almost every possible way (and have worse specs than many other TPUs) but are still the best inference solution on the market. Because they're incredibly cheap for what they do.

Even on Google Cloud, which has a markup over real costs, we have `918e12` bfloat16 FLOPs/s at $2.7 / hour, compared to about `1000e12` FLOPs/s from an H100 for $12 / hour. B100s aren't available on AWS yet but even a TPU v5e has `200e12` FLOPs/s at $1.2 / hour, so 1/5 the FLOPs for 1/10th the price.

👍 1

Write a reply

s2bk Feb 12

Answer 5. "the first byte will arrive in about 6us and the total transfer will take 188us." Wouldn't the total transfer time will be 188 + 6us = 194us?

↑ 1 👍 1 0 replies

Write a reply

samuela Feb 12

What's the difference between a "slice" and a "pod"? The article simultaneously states:

- "Chips are connected to each other through the ICI network in a Pod" when introducing a "pod"
- "A set of ICI-connected TPUs is called a slice." when defining a "slice"

↑ 1 3 replies

jacobaustin123 Feb 12 Collaborator

A pod is the maximum slice for a given topology. So e.g. 16x16x16 for TPU v4, or 16x16 for TPU v5e

👍 2

sshkhr Apr 8

@jacobaustin123 Then what is the difference between a pod and a super pod? From the text:

> the maximum pod size (called a superpod) is 16x16x16 for TPU v4 and 16x20x28 for TPU v5p

This is what I found from a different reference (TPU architecture: https://cloud.google.com/tpu/docs/system-architecture-tpu-vm):

> **TPU Pod**
> A TPU Pod is a contiguous set of TPUs grouped together over a specialized network. The number of TPU chips in a TPU Pod is dependent on the TPU version.
>
> **Slice**
> A slice is a collection of chips all located inside the same TPU Pod connected by high-speed inter chip interconnects (ICI). Slices are described in terms of chips or TensorCores, depending on the TPU version.

What is this 'specialized network' that pods are grouped over: is it just ICI (I assume not, since that would be a slice)? Or does TPU pods include both the TPUs and host devices (hence PCIe is also included perhaps)?

1

jacobaustin123 Apr 8 Collaborator

I think Google is pretty inconsistent with this terminology. I think basically in my view pod and superpod should be interchangeable, and refer to a slice of the maximum size.

1

Write a reply

Kumawat-Akhilesh Feb 14

Although ICI connects the TPUs within a pod (e.g., 8,076 TPUs), it would not be sufficient to directly connect a much larger number of TPUs—say 30,000 or even 100,000—as required for extremely large-scale training. In such scenarios, how are these TPUs interconnected?

Are all the TPUs connected individually to the DCN (e.g., via Ethernet) using NICs, similar to how NVIDIA connects GPUs to a scale-out network despite having an NVLink network? Or do the OCS switches connecting the 8,076-TPU pod also interface with DCN switches, forming a hierarchical structure? This hierarchical approach would mean there isn't a completely separate scale-up (intra-pod) and scale-out (inter-pod) network, as in NVIDIA's systems, but rather a unified network with a layered design. Could you clarify?

↑ 2

3 replies

jacobaustin123 Feb 14 Collaborator

Beyond the ICI network, TPUs are connected to their hosts via PCIe and the hosts are connected to NICs which connect to DCN. So I think similar to NVIDIA GPUs beyond the NVLink layer.

With that said, JAX/XLA can abstract DCN as yet another axis of a multi-TPU network, so from a software standpoint it's a minimal change from scaling over ICI.

1

Kumawat-Akhilesh Feb 14

Thank you for the clarification. Does the each TPU get connected to one NIC and hence one port of the DCN switch or because of ICI, we don't need one DCN switch port or one NIC per GPU?

jacobaustin123 Feb 16 Collaborator

The details differ a bit generation by generation, but generally there is one NIC per host (so per 4 or 8 TPUs). When it's possible to use ICI, you generally prefer to, but then you can do direct host-to-host transfers over DCN between any TPUs.

Write a reply

idnm Feb 24

I have several questions trying to connect a high-level picture (FLOPs, Bandwidths, etc) with the details of how systolic arrays work.

1. I don't think I understand all the details of how systolic arrays perform matmuls. If there are any additional refs with thorough explanations, that would be highly appreciated!

2. In particular, you state that

> When fully saturated the systolic array can perform one bfloat16[8,128] @ bf16[128x128] -> f32[8,128]9 multiplication per 8 clock cycles.

Where does the "8" come from? I like to think that systolic arrays perform a single matrix-vector multiplication per clock cycle, is that a bad mental model?

3. More importantly, is the critical batch size $B \approx 240$ directly related to the MXU geometry, i.e. it's height? Intuitively, it should be, because for smaller B loading weights would take longer than loading activations, and they could not be perfecly overlapped. The number $B \approx 240$ is presented everywhere as the ratio of FLOPs/bandwidth, but I wonder if the bandwidth was adjusted specifically to match the geometry-constrained batch size?

4. Along a similar lines. A general argument shows that for large matmuls compute (N^3) starts to dominate the memory (N^2) and we should be able to get to the compute-bound regime by simply scaling the matrix size. However, systolic arrays effectively perform N operations per clock cycle in parallel, to their actual compute time also scales as N^2 instead of N^3. Then, we generally can't compensate insufficient bandwidth by increasing the chip size. This is also clear from the systolic array dynamics. If the weights/activations can be feed at the same rate as they propagate through a systolic array (around $1.5GHz$) we won't be memory-bound.
   So, can the ideal memory BW be alternatively computed from the frequency and geometry of the chip?

p.s. Thanks for the fantastic guide!

↑ 1 ☺ 2 replies

jacobaustin123 Feb 24 Collaborator

A couple quick answers to these questions:

1. someone linked https://ecelabs.njit.edu/ece459/lab3.php below which is maybe better? You kind of just have to stare at some animations for a while.
2. To some extent it's just a magic number. You can think of the activations flowing through the systolic array at some rate, which is one batch of size 8 every 8 clock cycles. The mental model you suggest isn't really correct since it isn't doing them one at a time, but it's reasonable if it helps you remember.
3. It's related to the size of the MXU in the sense that that determines how many FLOPs/s it can do. If you made the MXU bigger without increasing your HBM bandwidth (or added another MXU), you'd increase this number. I think this ratio is something the designers picked based on their sense of what common ML workloads look like.
4. I think it's rather the opposite. If we reduced the size of the MXU, we'd again reach a compute-bound regime, since we'd reduce the number of FLOPs the machine can do per-byte. That would be wasteful though for many workloads.

Thanks :)

☺ 👍 1

idnm Feb 25

@jacobaustin123 Thanks, that clarifies a lot! Please let me follow up on a particular point, though.

I can't help thinking that batch size $B \approx 240$ being so similar to the size of a TPU $256x256$ is not a coincidence, but a very direct relation.

First, by looking at the pictures with sliding parallelograms, it seems that the width (batch size) of the parallelogram sliding from the left must be at least the size of TPU, otherwise it will be too short and cause bubbles and underutilization. So I'd think that we must have a relation batch size >= TPU width. Is that misguided?

From a slightly different angle. Say, for TPUv5e we have four MXU, which effectively gives a 256x256 systolic array operating at a frequency 1.5 GHz. The HBM bandwidth is 820 GB/s. If we compute the number of bytes per cycle *per input* (assume we only slide activations from the left, so that there are 256 inputs) we get 820 BG/s / 1.5 GHz / 256 = 2.14 bytes, which is very close to 2 bytes per cycle per input that need to be fed to the systolic array. Is that a coincidence?

Or, framing this another way, if we were to increase the HMB bandwidth by 10x, would the critical batch size drop by 10x? I don't think it will, because the systolic array frequency remains fixed and already saturated, so the ability to feed the inputs faster won't change the processing speed.

☺

Write a reply

FL33TW00D Mar 1

I might be missing something obvious here, forgive me.

The link you provided for the v5p lists the Interchip Interconnect BW (ICI) as 4800Gbps (600GB/s): https://cloud.google.com/tpu/docs/v5p#system_architecture

If it has 6 way interconnect, wouldn't that be 100GB/s? Where does 90GB/s come from?

↑ 1 ☺ 1 reply

jacobaustin123 Mar 3 Collaborator

It's less than ideal but all TPU docs report slightly different numbers for these bandwidths. Sometimes they're rounded for simplicity, other times there are bottlenecks that limit the peak bandwidth for different operations. I've added a short note.

☺ 👍 1

Write a reply

tianshub Mar 16

This is really insightful. Minor nit: the equation in the answer of Q4 has lhs max{T_math, T_comm} but rhs max{T_comm, T_math}

↑ 1 ☺ 👍 1 0 replies

Write a reply

mrinal-essential Mar 19

Could you give a brief description of how the Optical Switches (OCS) compare & contrast against the Electrical Switches that are the standard in datacenters today? Is it mostly savings for Google or are there advantages that ML practitioners can benefit from?

↑ 1 ☺ 1 reply

jacobaustin123 Mar 19 Collaborator

https://arxiv.org/pdf/2208.10041 I think provides some good answers. My general sense is that the main advantage is reconfigurability, but I'm not an expert

☺

Write a reply

Kumawat-Akhilesh Apr 10

I see above that ICI bidirectional bandwidth per link for TPU v6e (Trillium) is 180 GB/s. For the newly introduced Ironwood, the ICI bidirectional bandwidth per link is 1.2Tbps or 1200/8= 150 GB/s. But the Ironwood announcement says the ICI for Ironwood is 1.5x of Trillium. How shall I reconcile or I am mis understanding here.
Thank you.

↑ 2 ☺ 4 replies

jacobaustin123 Apr 10 Collaborator

Ironwood is connected in a 3D torus, so we have an addition 3/2 total bandwidth.

☺ 👍 2

Kumawat-Akhilesh Apr 10

But then it means Ironwood is 1.25x better... Ironwood has 150 GBps while Trillum has 180 GBps.. So Ironwood would be (150/180)* (3/2) = 1.25x times more bandwidth and not 1.5x?

Also this assumes there is one link connecting one TPU to another. So Trillium with 2 d torus would be 4 links and Ironwood with 3 d torus would be 6 links.

At the same time, if I visit Google's Trillium page (https://cloud.google.com/tpu/docs/v6e), Trillum has 3584 Gbps or 448 GBps. With 4 links, I get 112GBps for Trillum. This is much lower than above in the table stated 180 GBps...

jacobaustin123 Apr 10 Collaborator

Ironwood also has 180 GBps depending on the workload. As noted in a comment above, these bandwidths differ slightly depending on the workload.

👍 1

Kumawat-Akhilesh Apr 10

Thank you, Jacob!

Write a reply

domluna May 12

I'm a bit confused by the ICI calculation in Q6. Does it assume 15GB is on adjacent TPUs, so 7.5GB can travel on each link in one hop? The way I reasoned about this was that the upper bound would be transferring 1GB from one corner to the other, which would be 6 hops, so 1e9 / 9e10 ~= 11ms * 6 ~= 66ms to get the final GB to the destination TPU. This doesn't factor in latency calculations.

↑ 1 

2 replies

domluna May 12

I can't edit above: 66ms < 166ms (the lower bound in the answer). But would latency of the hops be greater than 100ms?

jacobaustin123 May 13 Collaborator

No, there are many more than one hops. But this is the absolute minimum amount of time from a throughput standpoint. Somehow, 15GB needs to pass through those two links in one direction. Each one has 4.5e10 bytes / s of bandwidth. 15e9 / (4.5e10 * 2) = 166ms. The per-hop reasoning ignores that we're throughput and not latency-bound.

Write a reply

FrankLong1 May 14

This is phenomenal reading thank you so much for writing this!!

↑ 1 

0 replies

Write a reply

vorushin May 19

CS336 (Spring 2025) has a great video about the modern GPU architecture (a nice addendum to the Appendix A): https://www.youtube.com/watch?v=6OBtO9niT00

↑ 1 

0 replies

Write a reply

RissyRan Jun 19

Thanks for the great doc!

I am curious about `For most TPU generations, it performs one bfloat16[8,128] @ bf16[128,128] -> f32[8,128] matrix multiply`. It mentions LHS (activation) use 8x128 and RHS (weight) uses 128x128. Wondering if we should put opposite when sequence length is increasing? i.e. activation is (batch, seq, model_dim) and weight is (model_dim, intermediate_dim). In recent DeepSeek v3 case, seq length could be up to 128K, and intermediate_dim for moe layer is 2k.

If we could opposite the padding size, as a JAX user, how could we update our code accordingly? Thank you!

↑ 1

0 replies

Write a reply

Write Preview Aa

Write a comment