High Performance LLMs From First Principles (2024)

Session 3
https://github.com/rwitten/HighPerfLLMs2024

rwitten@

Goal: learn how to achieve **high performance** for LLMs

This week: understand multiple chip performance

(Last week was single chip)

Program (write code in Jax)

Predict (roofline on napkin or spreadsheet)

Profile (run code, compare to predictions)

My Asks

Please ask lots of questions! Just raise your hand or speak up!

If there are topics you're interested in, message me between sessions.

Join the discord! https://discord.gg/2AWcVatVAw

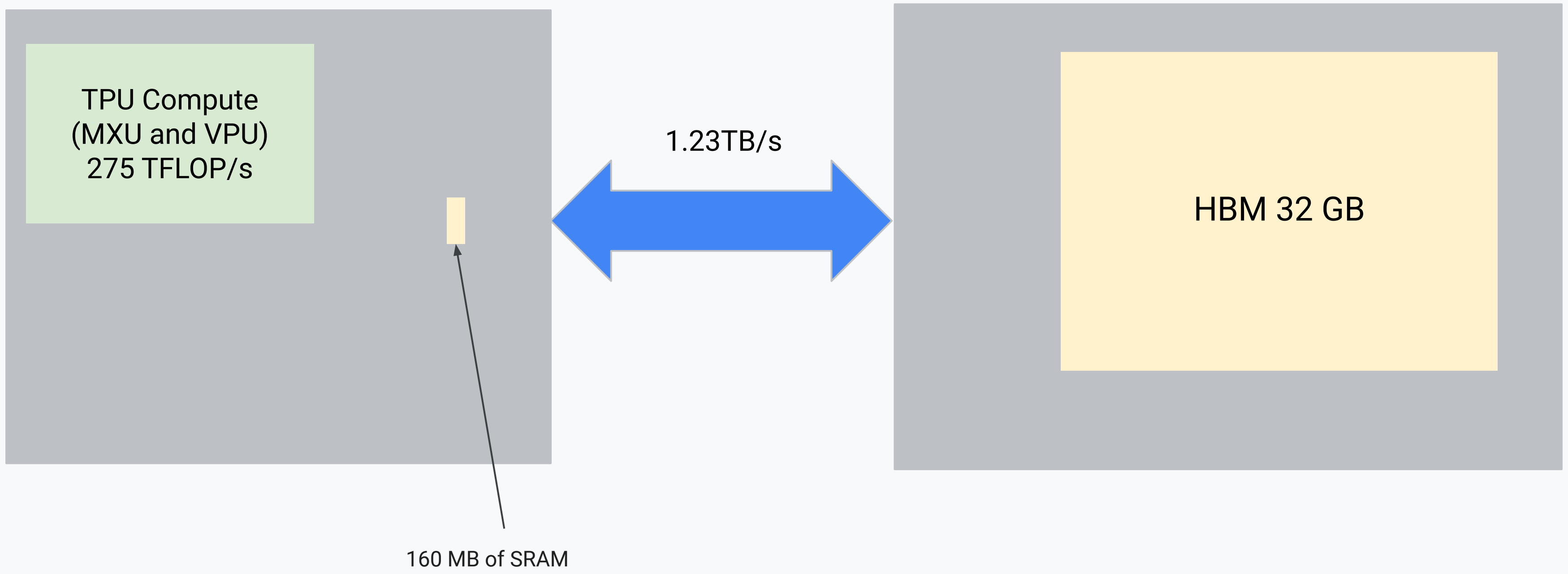
Do the exercises! Give feedback, ask questions!

Website: https://github.com/rwitten/HighPerfLLMs2024

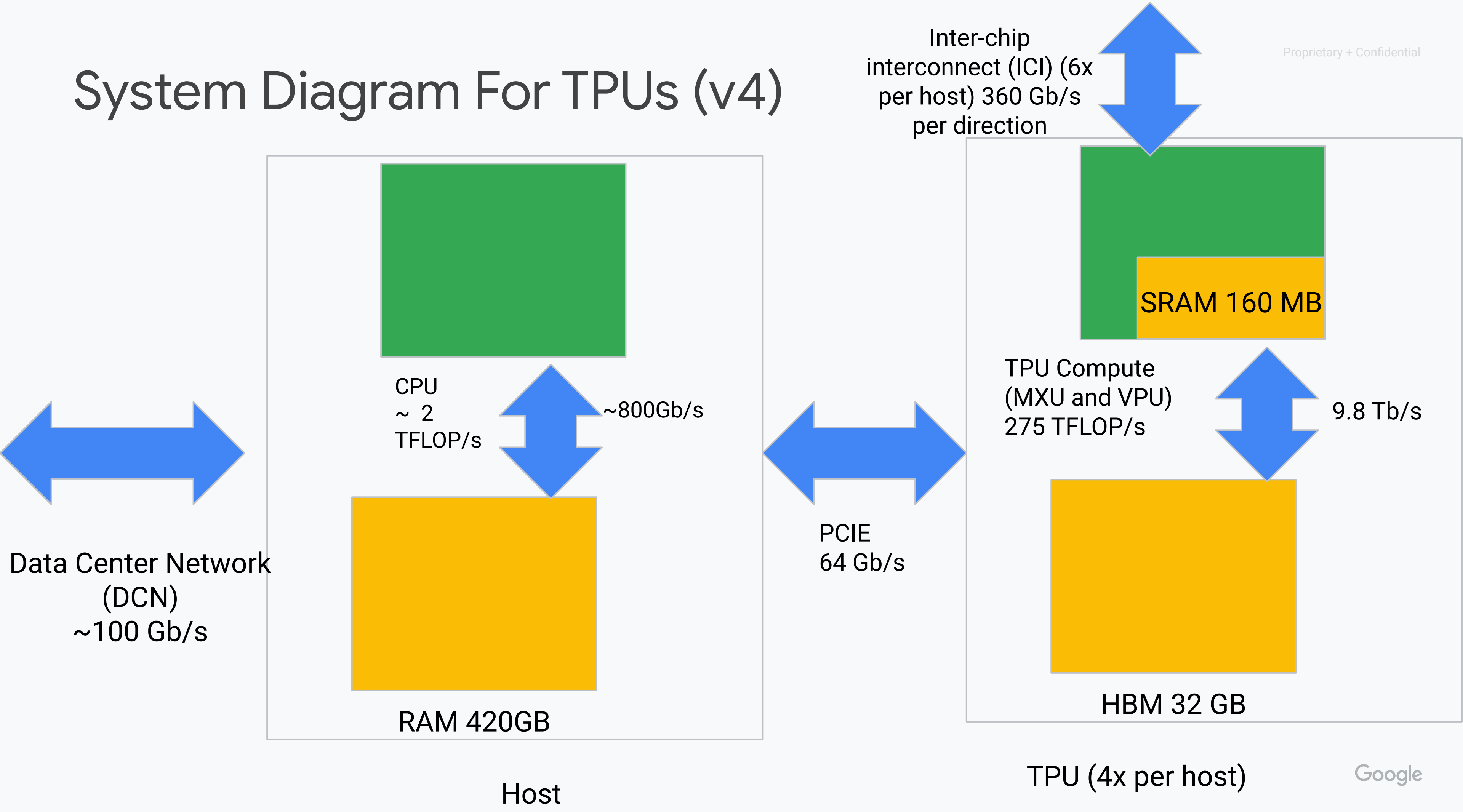
Why Multiple Chips?

- Training a frontier model would take 3170 years on a single chip.
- (Other reasons too.)

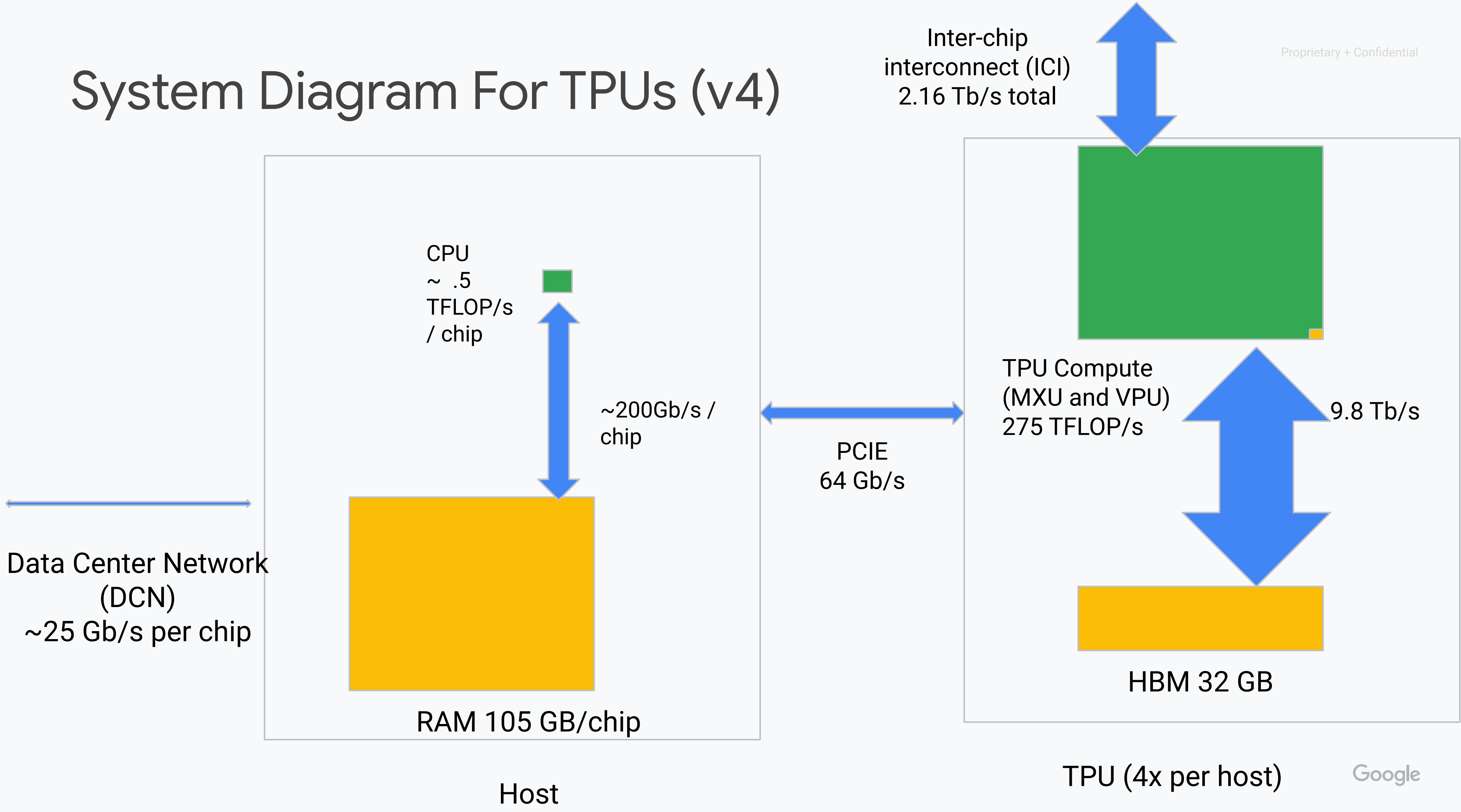
Last Week: System Diagram For TPUs (v4)



System Diagram For TPUs (v4)

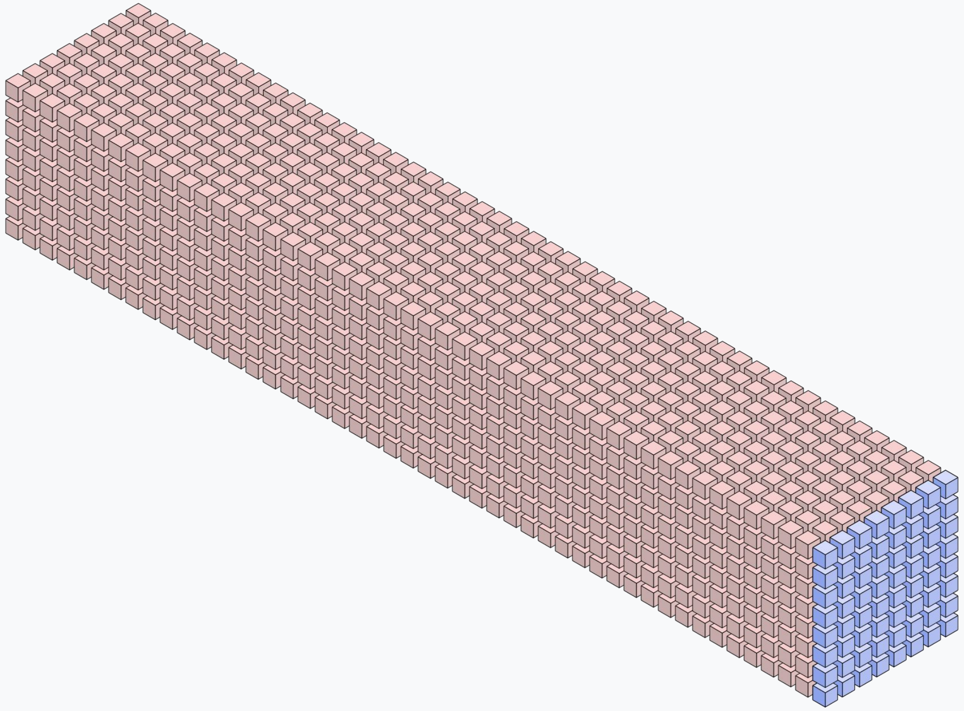

Inter-chip interconnect (ICI) (6x per host) 360 Gb/s per direction
SRAM 160 MB
TPU Compute (MXU and VPU) 275 TFLOP/s
9.8 Tb/s
HBM 32 GB
TPU (4x per host)
CPU ~ 2 TFLOP/s
~800Gb/s
RAM 420GB
Host
PCIE 64 Gb/s
Data Center Network (DCN) ~100 Gb/s

System Diagram For TPUs (v4)

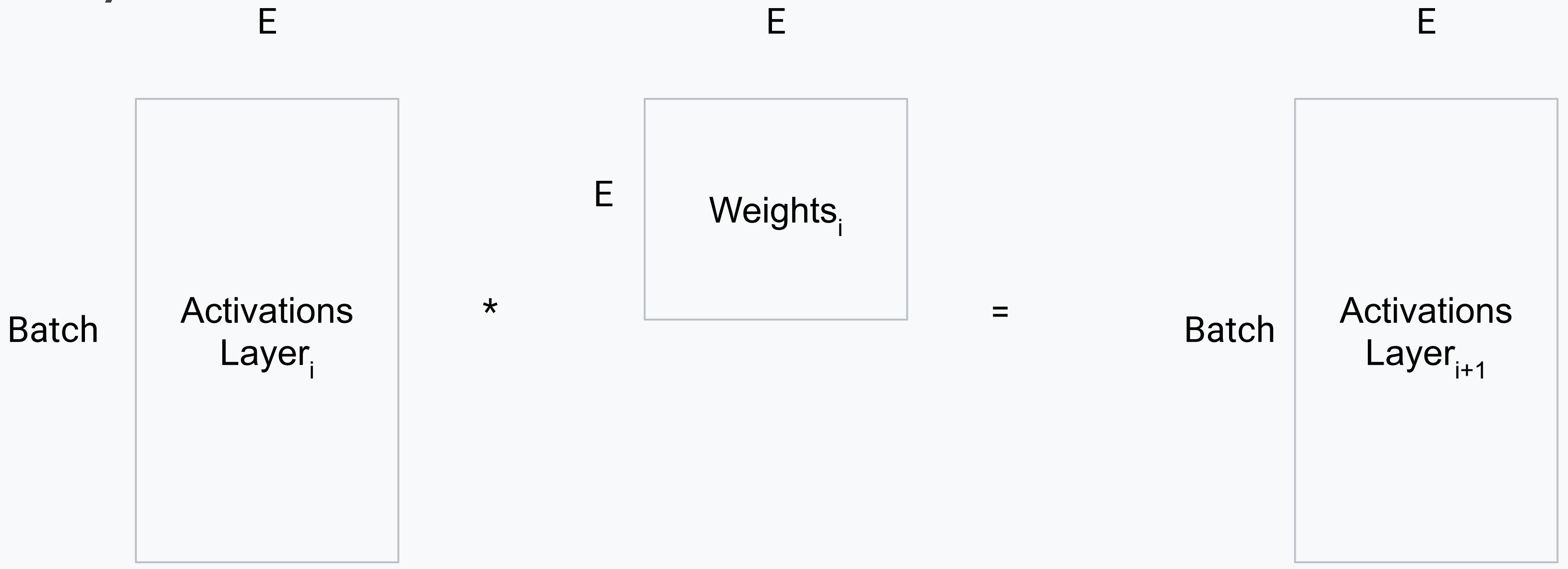


Recall the private network (ICI/NVLink)

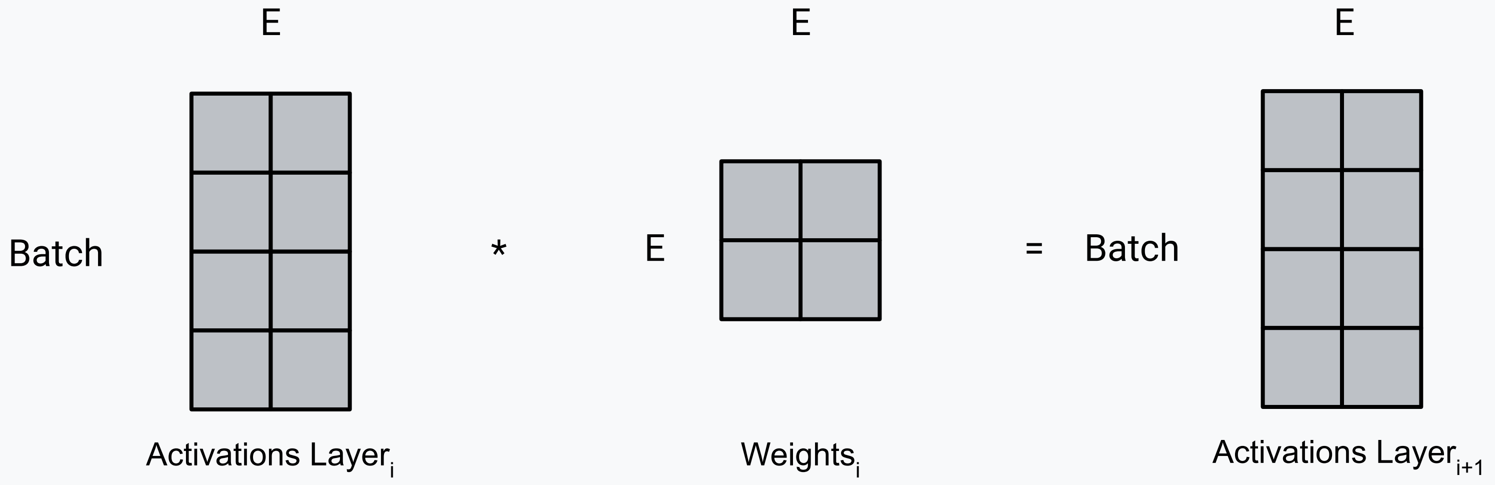
- We have thousands of chips that we want to work together to run a computation.
- Some have a fast interconnect (inter-chip interconnect) to allow them to talk privately together.
- Main focus today!
- (On TPU it is called inter-chip interconnect. On Nvidia GPU it is called NVlink. AMD and Intel have similar technologies.)



Simplified Workload For Analysis: Just 1 matrix multiply per layer



Let's Consider Our Problem

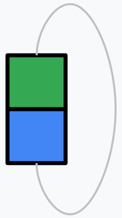


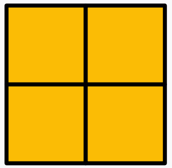
Jax’s Answer: GSPMD

- Tell the compiler to “shard” the matrices over different axes.
- This is a very restrictive set of computations but an amazing simplification.
- Let’s see it in detail…
- Fun fact: GSPMD probably stands for G Single Program Multiple Data. The original paper doesn’t suggest what the G stands for.

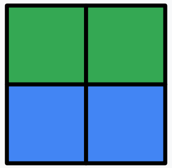
What is sharding?

- Assume we have a 1D circular array of devices.
- Then there are three possible shardings for a matrix:
  - Fully replicated (unsharded)
  - Sharded by rows
  - Sharded by cols

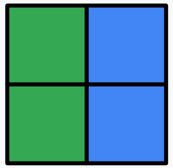




Fully Replicated (All the Data on All Computers)



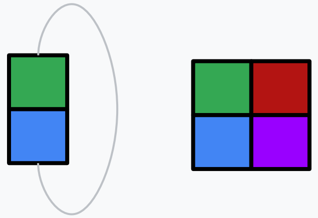
Sharded by rows

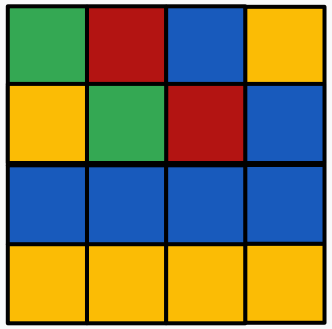


Sharded by cols

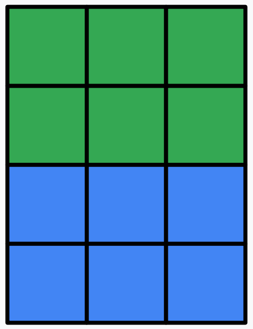
What is sharding? (Continued)

- Matrices don’t need to be square (but they need to be equally divisible)
- This is why we typically use powers of 2 for EVERYTHING.
- (Arguably the XLA team could fix this. But not a big deal overall.)

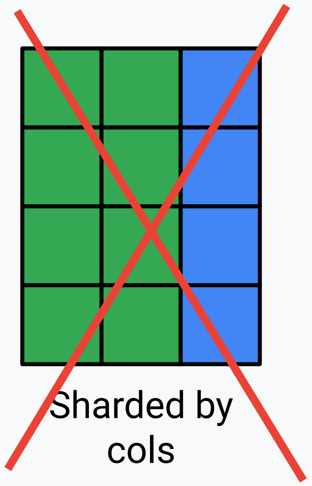




Fully Replicated (All the Data on All Computers)



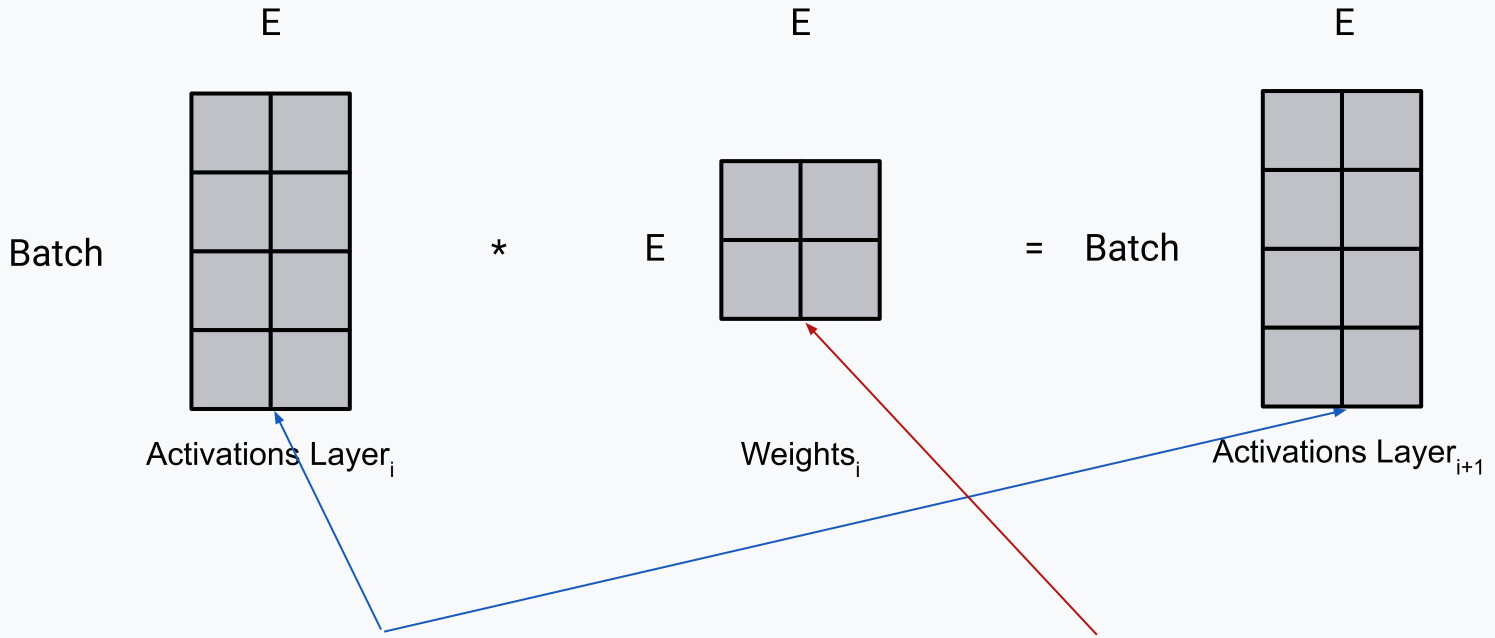
Sharded by rows



~~Sharded by cols~~

Break For Programming Exercise

Let’s Consider How to Shard Our Problem

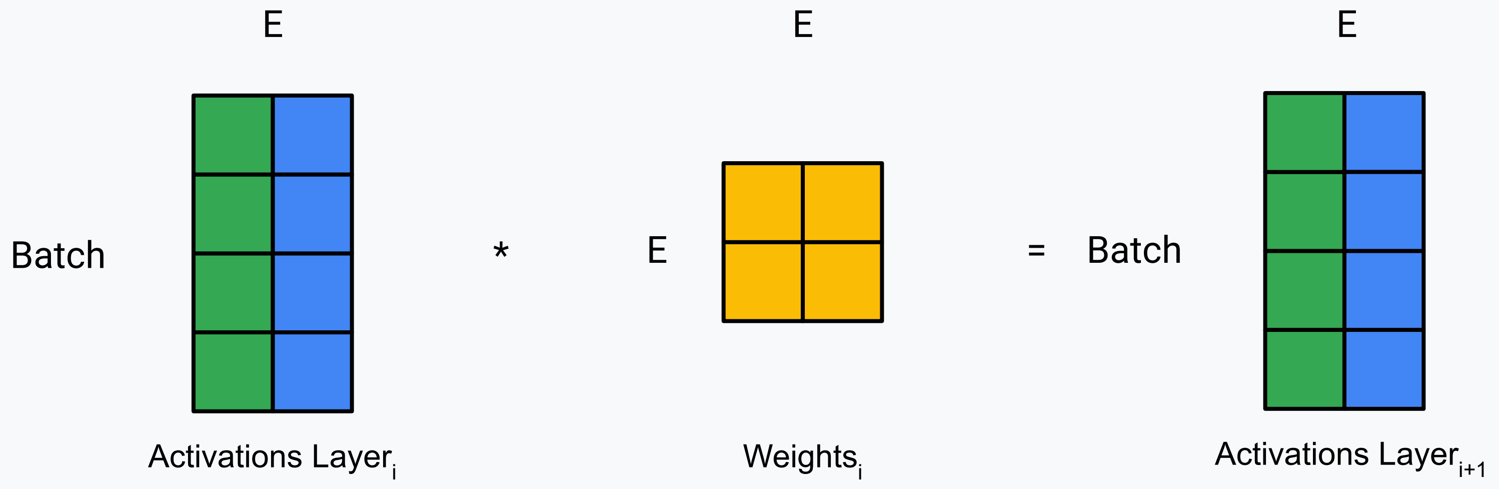


Let’s assume these use the same sharding. (Think of it like a for loop!)

Weights can be sharded differently than activations

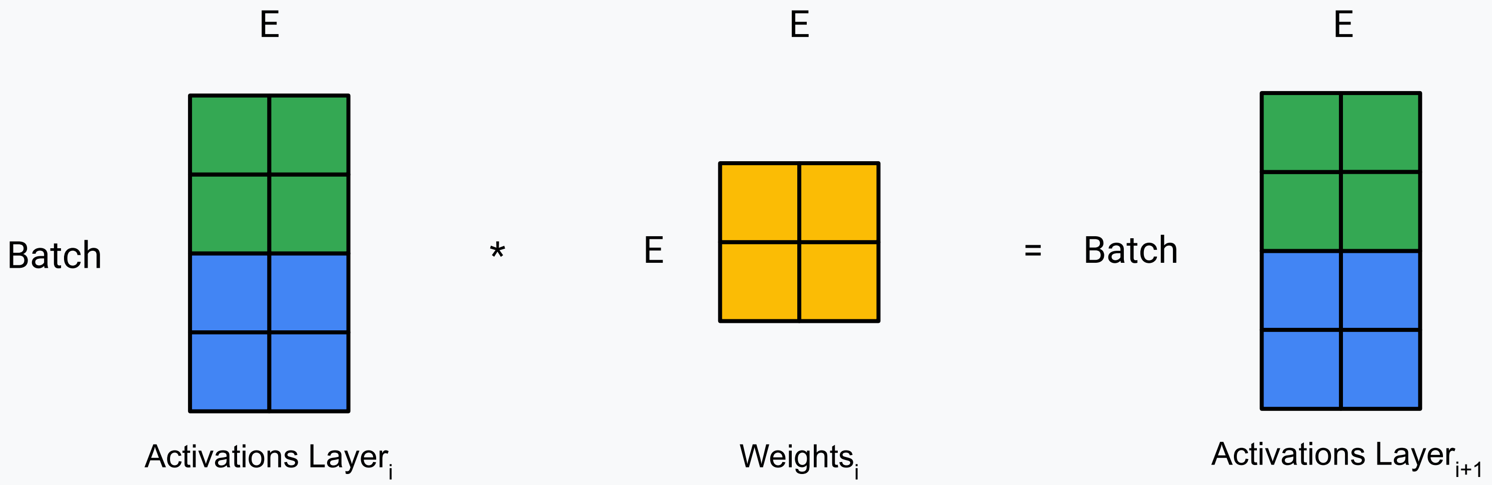
Activation Sharding (Tensor or Model Parallelism)

- Activations NEED to be sharded – they're much bigger that weights. We can either shard them by rows or columns.
- (For now assume weights are fully replicated)



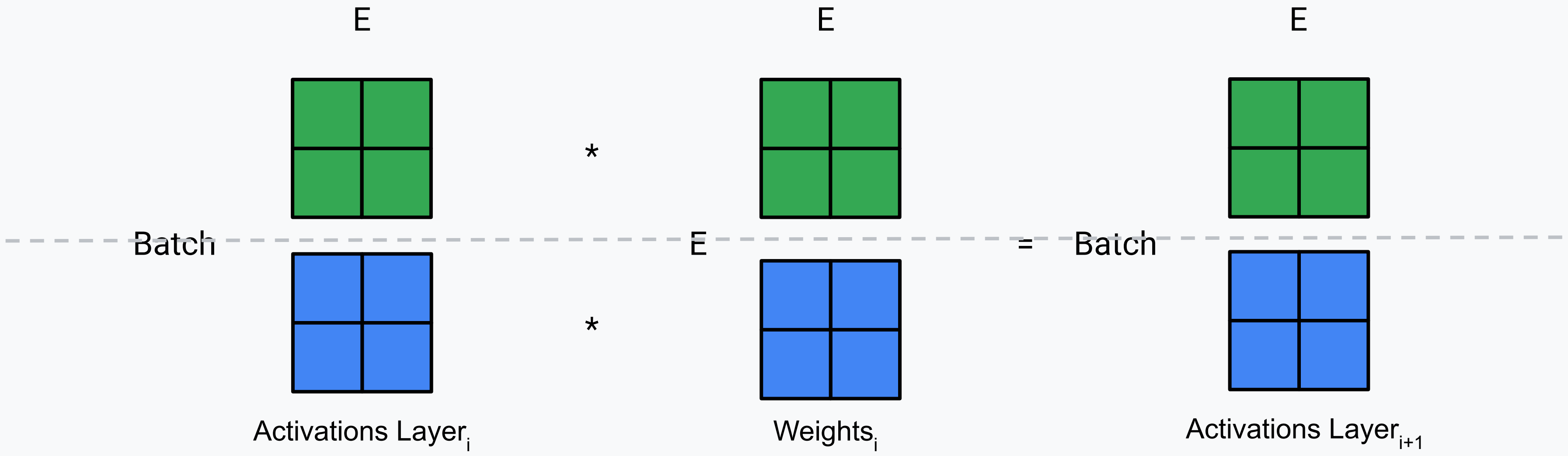
Batch Sharding (Data Parallelism)

- Activations NEED to be sharded – they're much bigger that weights. We can either shard them by rows or columns.
- (For now assume weights are fully replicated)



Batch Sharding (Data Parallelism)

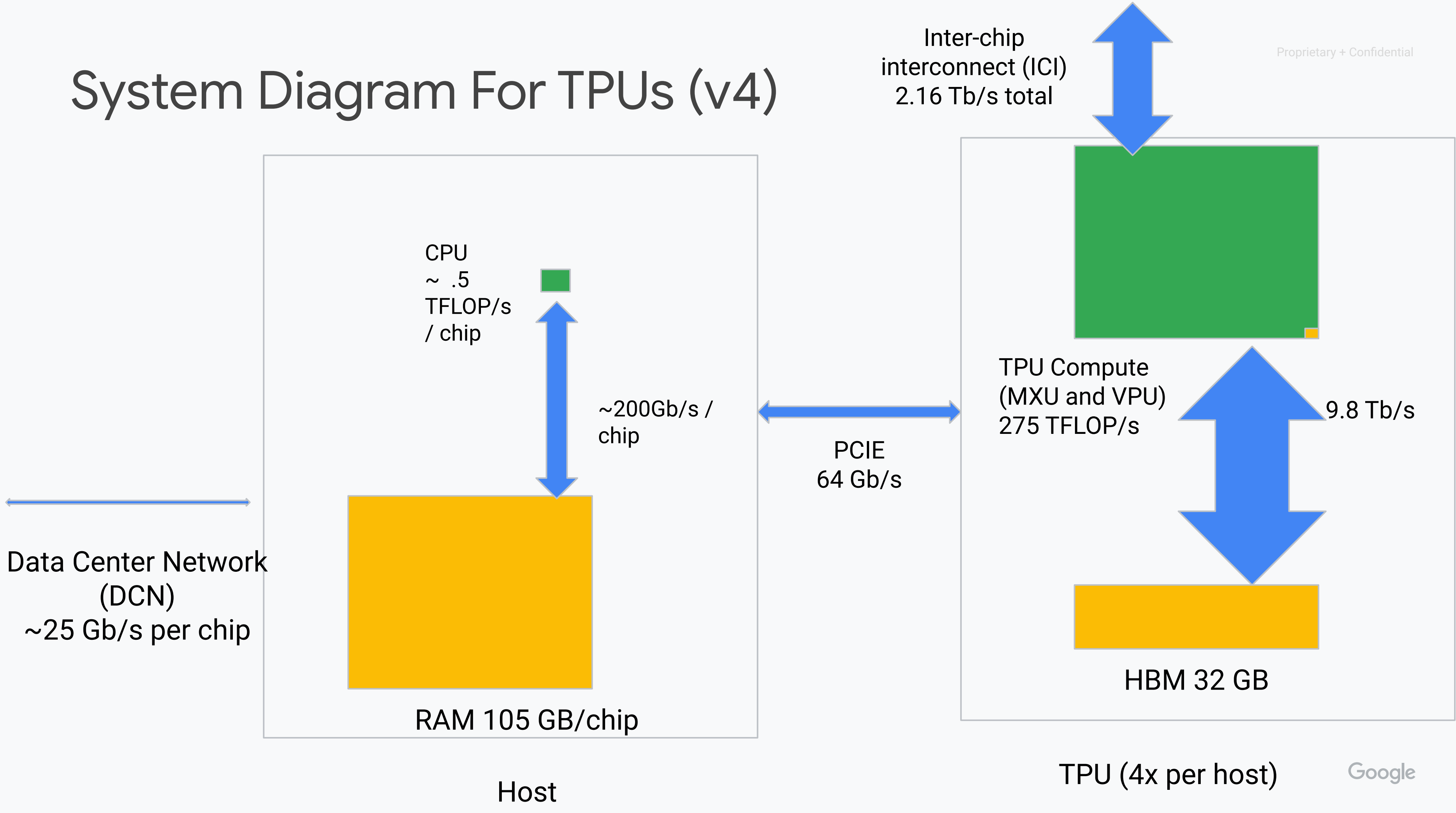
- Batch Sharding is much easier to reason about. Embarrassingly parallelizable.
- It keeps working with a linear speedup as long as you scale batch linearly with number of TPUs!



Problem: But What About the Weights?

- As long as we’re scaling the batch linearly with the number of chips, batch sharding / data parallelism delivers strong performance!
- But what if we grow the size of the model?
- GPT3 is 175B parameters.
- At 4 bytes per parameter (ignoring optimizer state – the real problem is more severe):
  - **700GB** for GPT3
- Problem: each v4 chip only has **32GB**.

System Diagram For TPUs (v4)

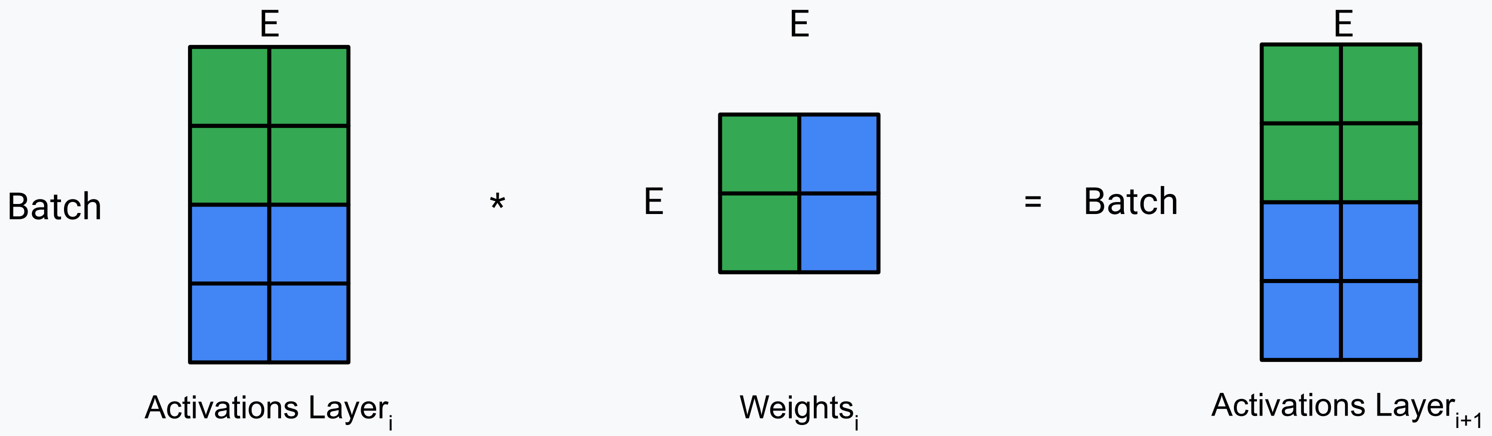

Inter-chip interconnect (ICI) 2.16 Tb/s total
CPU ~ .5 TFLOP/s / chip
~200Gb/s / chip
PCIE 64 Gb/s
TPU Compute (MXU and VPU) 275 TFLOP/s
9.8 Tb/s
Data Center Network (DCN) ~25 Gb/s per chip
RAM 105 GB/chip
HBM 32 GB
Host
TPU (4x per host)

Key Arithmetic Intensities of the Hardware (v4 specific)

- HBM: 275 TFLOP/s / 9.8 Tb/s =   224 FLOPs/byte
  - Fast but there’s too little HBM
- ICI: 275 TFLOP/s / 2.16 Tb/s =   1018 FLOPs/byte
  - **Let’s try it!**
- ~~PCIE: 275 TFLOP/s /   64 Gb/s = 34375 FLOPs/byte~~
- ~~DCN: 275 TFLOP/s / 25Gb/s = 88000 FLOPs/byte~~

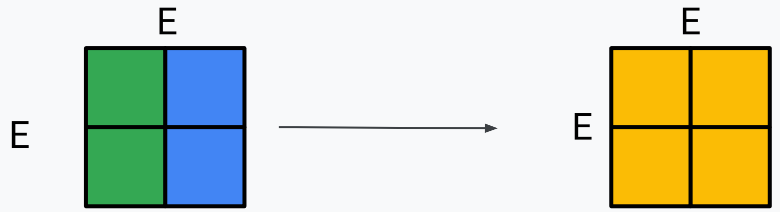
Batch Sharding with Weight Sharding (Fully Sharded Data Parallelism)

- Shard the weights as well.
- Now each TPU can't compute the matrix multiplication given the data it has locally! The next slide explains the solution – all-gathering the next layer while doing the arithmetic on this layer.
- (Requires storing only 1 layer at a time. The big NN's have much more than 200 matmuls so this is a big savings!)

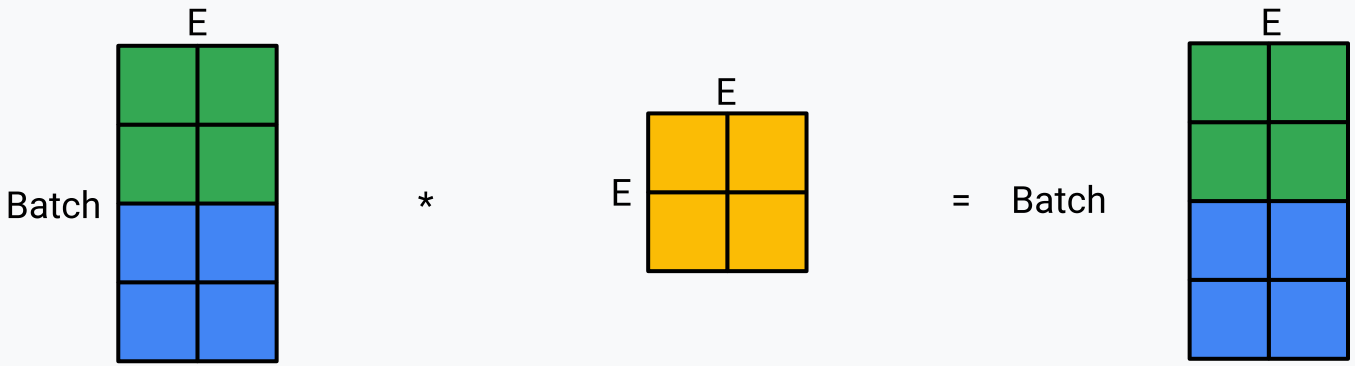


Batch Sharding with Weight Sharding (FSDP)

Layer$_i$: All-gather



Layer$_{i-1}$: Matrix Multiply (same as data parallelism)



When Does FSDP Work?

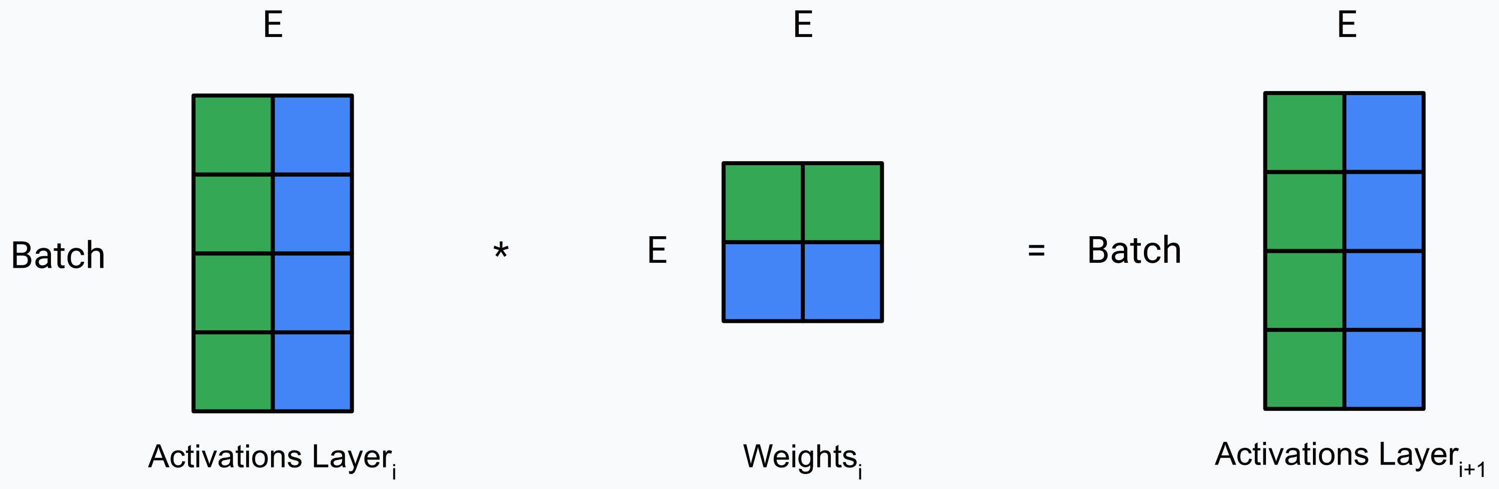
- The matrix multiply takes:
    - FLOPs = 2 * BATCH_PER_CHIP * E^2
- The all gather requires reading ~2 * E^2 bytes (assuming bfloat16)
- Arithmetic intensity (FLOPs/bytes) = 2 * BATCH_PER_CHIP * E^2 / (2 * E^2 bytes) = BATCH_PER_CHIP FLOPs / byte.
- ICI arithmetic intensity is 1018 FLOPs/byte (assumes all 3 axes assigned) so we need our BATCH_PER_CHIP to be comfortably larger than 1018.
- Depending on the hardware generation, we might not be able to overlap the ALL GATHER with the arithmetic (if they use the same hardware) so on those generations we want it WAY bigger.

When Specifically Is FSDP Not Sufficient?

- Very Big LLMs (due to batch size constraints).
  - Due to memory constraints, a single chip can’t fit a global batch big enough.
    - Imagine sequence length 65K.
- Inference – FSDP doesn’t help with latency. For many applications, we need lower latency. Example:

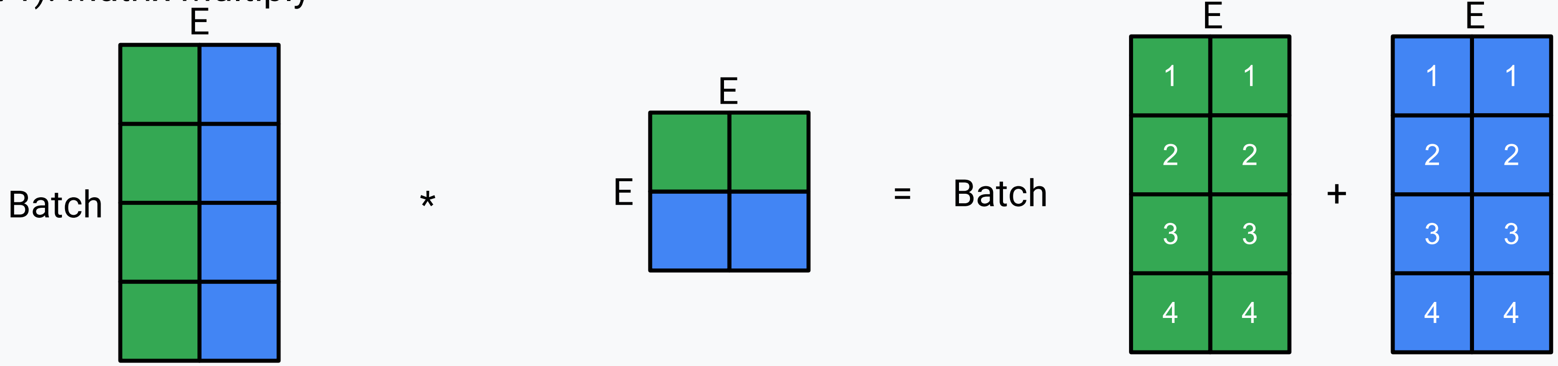
Activation Sharding (Tensor or Model Parallelism)

- Shard the activation dimension.

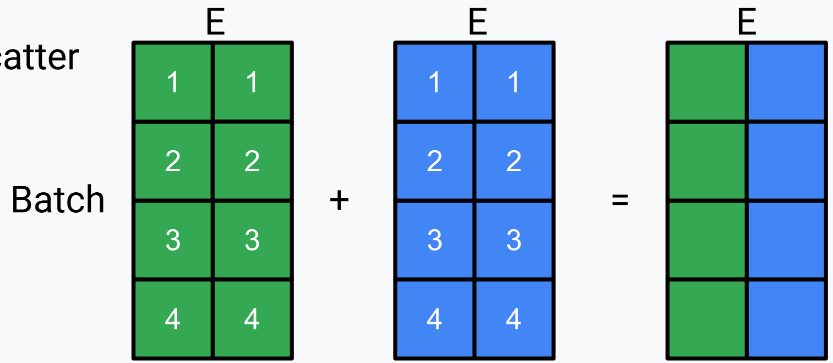


Activation Sharding (Tensor or Model Parallelism)

Layer$_i$ (part 1): Matrix Multiply



Layer$_i$ (part 2): Reduce-scatter

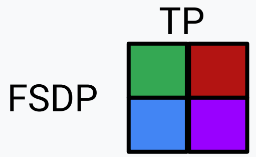


Google

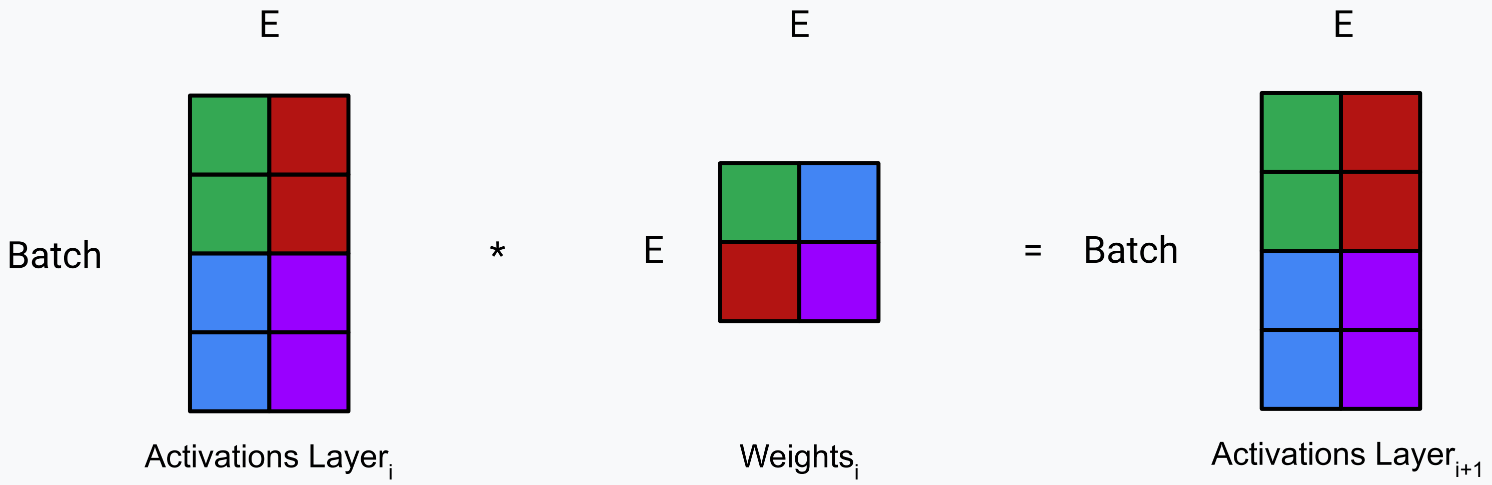
When Does Tensor Parallelism Work?

- The matrix multiply takes (per chip):
    - FLOPs = 2 * BATCH * E^2 / (NUM_TP_SHARDS)
- The reduce scatter requires ~2 * BATCH * E bytes (assuming bfloat16)
- Arithmetic intensity (FLOPs/bytes) = E / NUM_TP_SHARDS.
- ICI arithmetic intensity is 1018 FLOPs/byte (assumes all 3 axes assigned) so we need E to be comfortably larger 1018 * NUM_TP_SHARDS.
- Overlapping the operations is possible with a pipelined implementation. It is the White Whale of compiler optimizations, the **collective matmul.** (Because it can't just be "prefetched" it is WAY harder than FSDP.)
- Our latency will go down linearly with our number of TP shards.

FSDP with TP

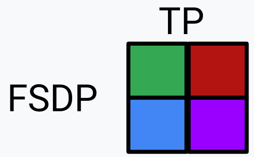


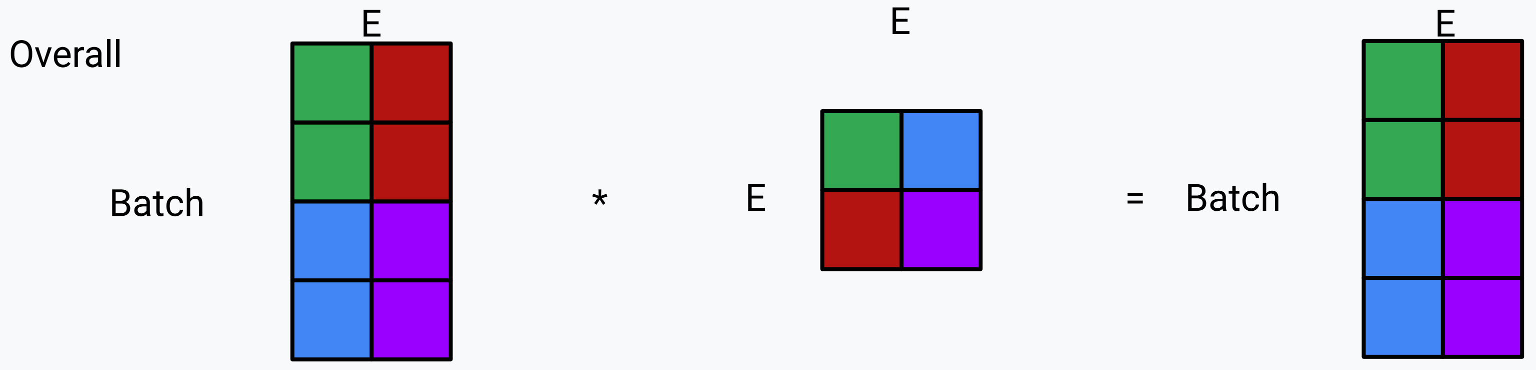
- Assume a 2D mesh.
- While processing layer$_i$, all-gather layer weights for layer$_{i+1}$ for FSDP.
- Then it just becomes TP.

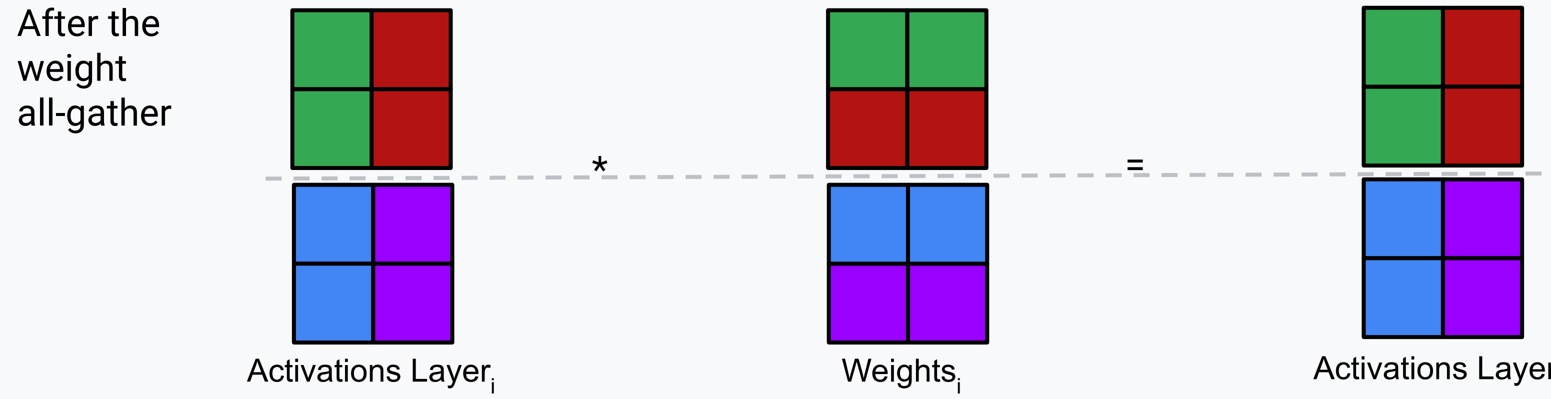


FSDP with TP

- (After all gathering the weights it becomes just TP)







Activations Layer_i Weights$_i$ Activations Layer_{i+1}

When Does FSDP with TP work?

- Given NUM_TP_SHARDS and NUM_FSDP_SHARD
- When both:
  - E > 2 * 1018 * NUM_TP_SHARDS
  - **NUM_TP_SHARD * BATCH_PER_CHIP** > 2 * 1018
- Key is that all the members of a Tensor Parallel share their batch!
- So if we have a VLP (16 by 16), we can assign one axis to TP and one to FSDP.
  - E > ~32k
  - BATCH_PER_CHIP > 125
- In our examples, we've had E,F of about 52k and batch per chip of about 1k.

Break For Analysis (Shardings.py)

Thanks!

Ping me (rwitten@google.com) with feedback, suggested topics, etc!