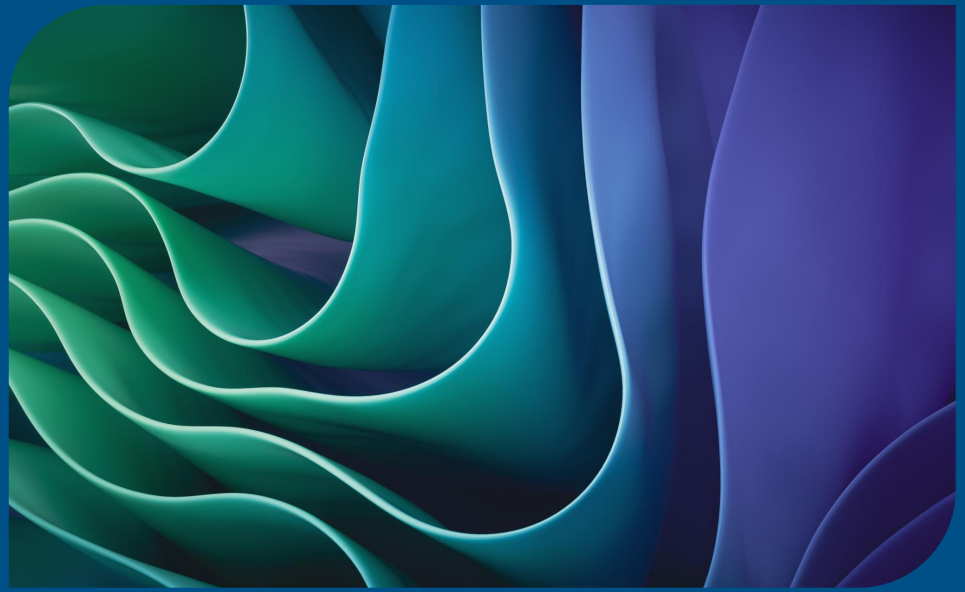
2025

Scaling systems for LLMs

Systems (hardware & software) used to train large models



Alban Rrustemi

Introduction

Overview

Background and motivation

Scaling laws for transformers

A Systems View

Intro to accelerators for LLMs + some basics in computer architectures

Supercomputers

How do we scale compute for training LLMs?

Training transformers

Batch parallelism

Data parallelism

FSDP (Fully sharded data parallelism)

Tensor parallelism (aka Megatron)

Pipeline parallelism

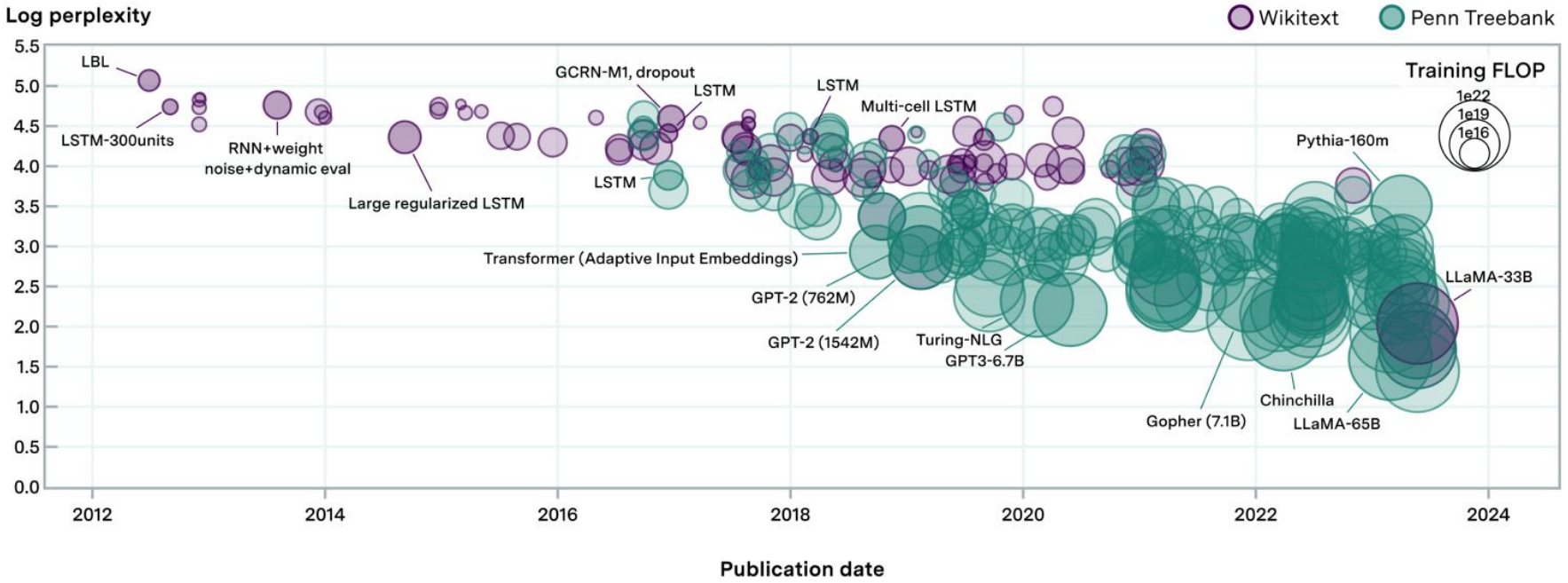
Credit: Scaling book (aka: How to Scale Your Model)

Notable AI models

EPOCH AI

Training compute (FLOP)

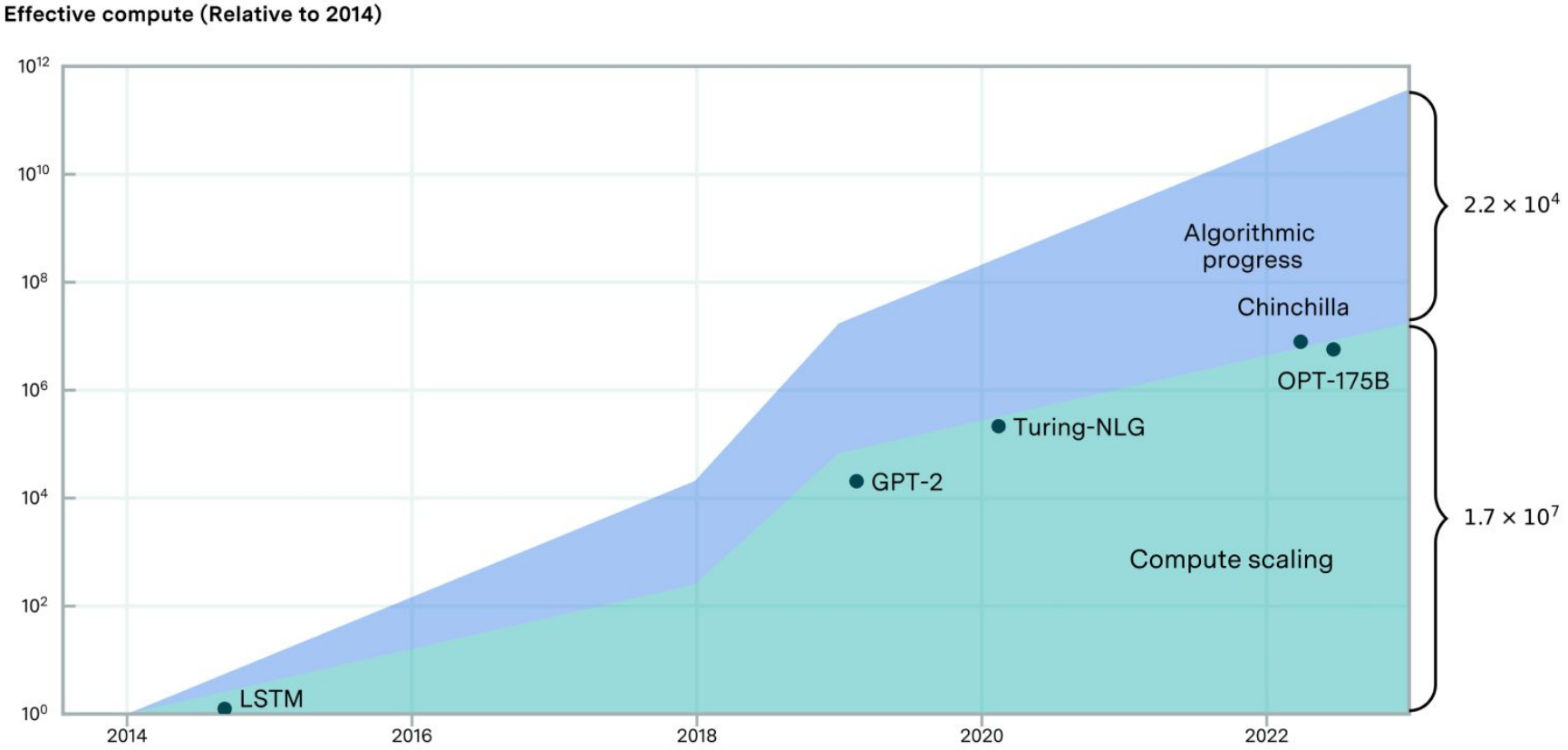
Performance and scale of language models over time



For more details, see https://epoch.ai/blog/algorithmic-progress-in-language-models

Relative contribution of compute scaling and algorithmic progress

EPOCH AI

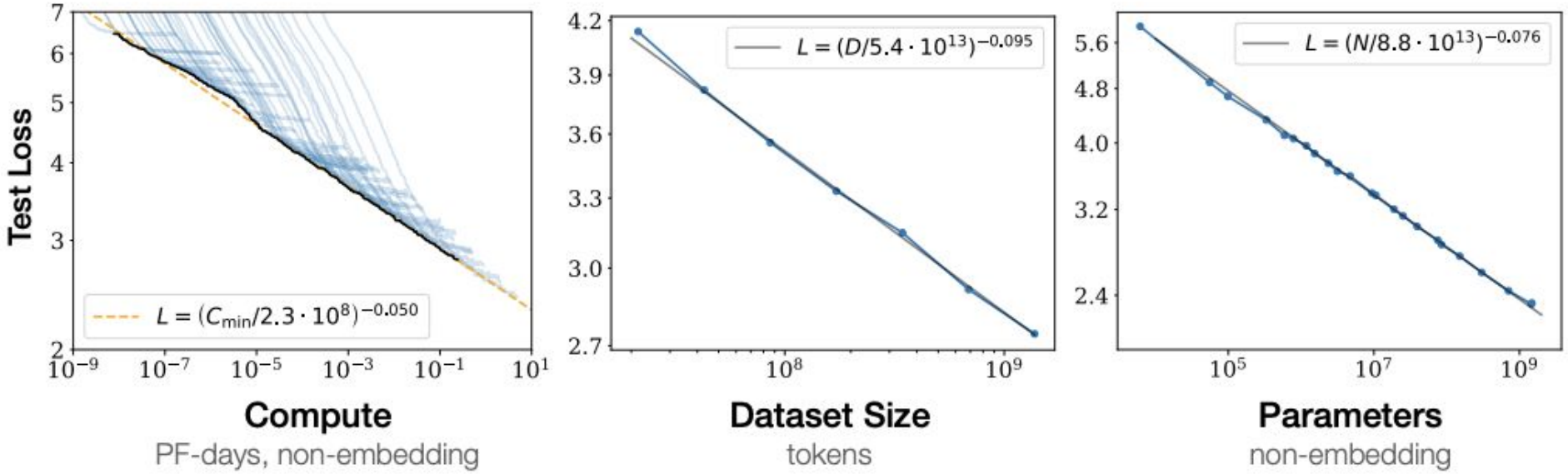


https://epoch.ai/blog/algorithmic-progress-in-language-models

Scaling laws for transformers

Scaling laws for transformers

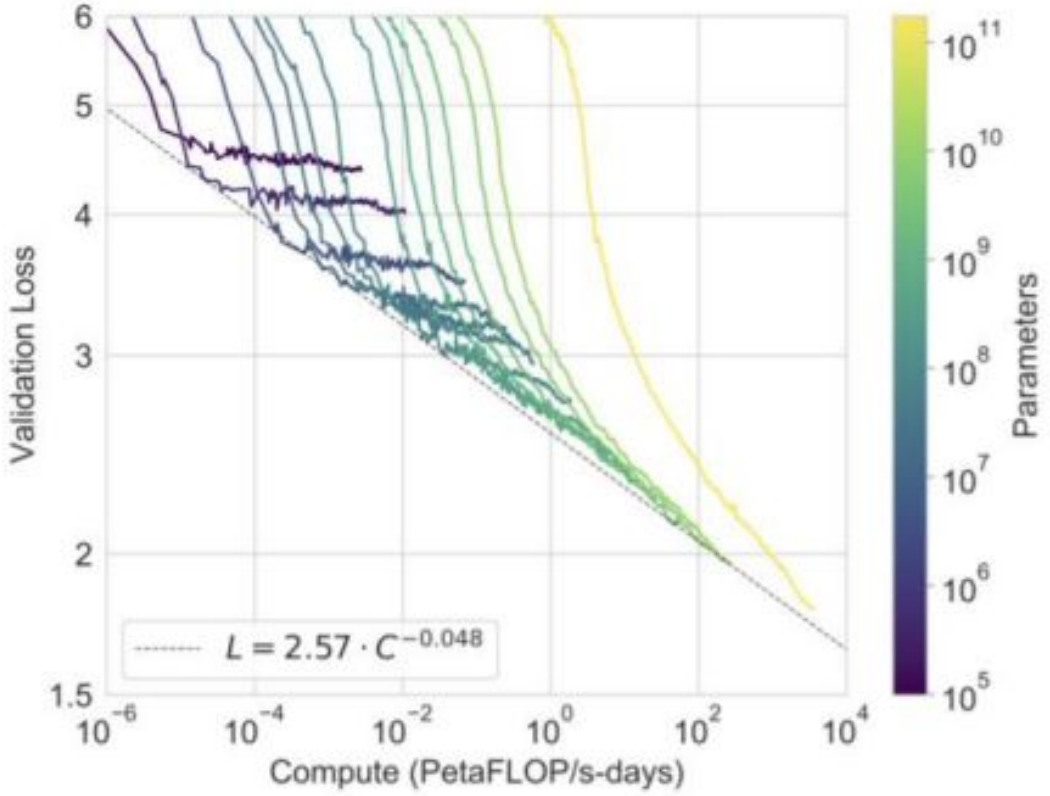
Simple, predictive behaviors ('laws') for some properties of transformer-based models



More accurate estimates in: Chinchilla scaling laws

Scaling laws for transformers

Focus: model size & compute



Model	GPT-3 (2020)	Llama-3 (2024)
# parameters (B)	175	405
PetaFLOPs	5.24E+08	3.79E+10
PetaFLOP/s-Days	6,064	438,750

Wait... what is a PetaFLOP?

Key takeaway: more FLOPs + more data => better models!

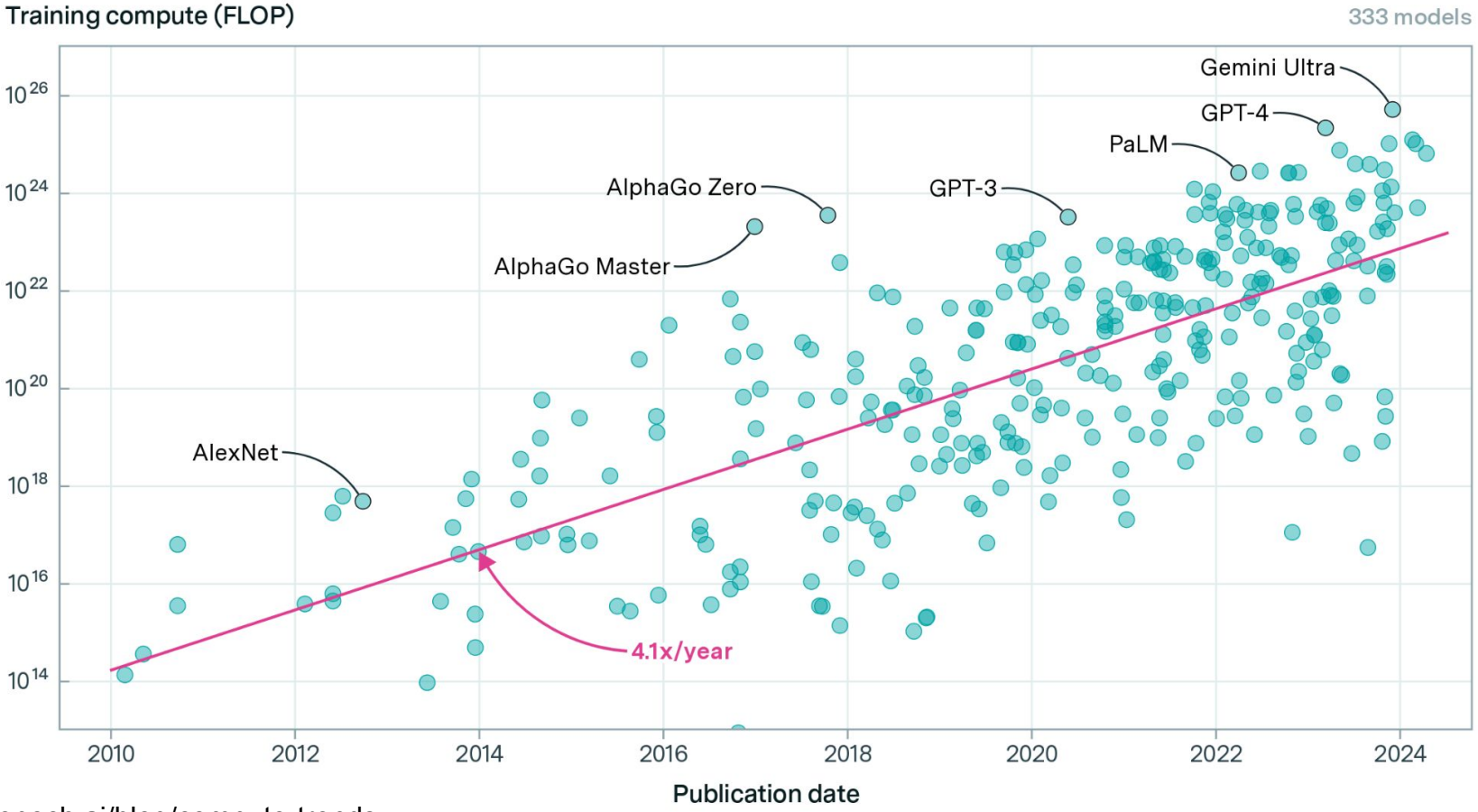
Side channel topics

- The bitter lesson by Rich Sutton
- Recent breakthroughs in AI: From Attention to GPT-4: Aravind Srinivas and Lex Fridman

Next section: a very brief intro on the compute systems that provide the FLOPs we need!

Training compute of notable models

EPOCH AI

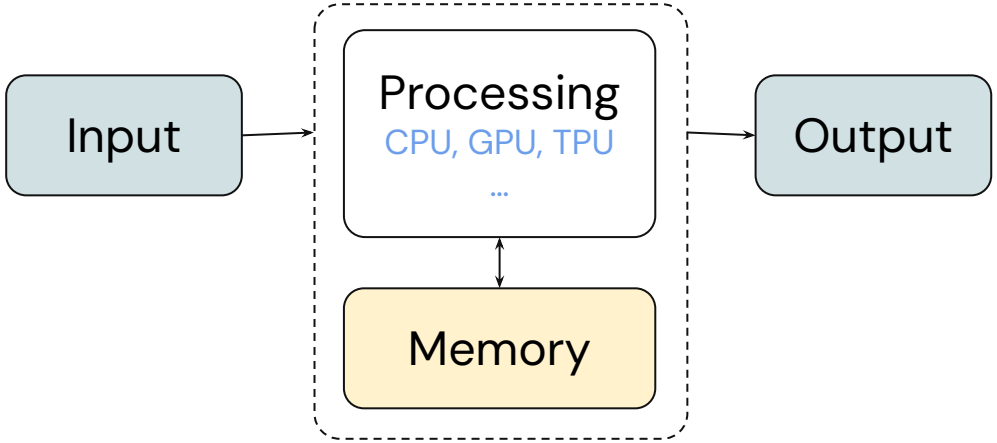


https://epoch.ai/blog/compute-trends

A brief introduction on accelerators for training LLMs (and other models)

Basics of computer architectures

The Von Neumann architecture



Example instructions

From Risc-V for adding two numbers:

```
add rd, rs1, rs2
```

What types of numbers are input and output values?

Data/number representation:

Key criteria:
1. type of number (integer vs rational)
2. number of bits to represent it

Integer:
- Traditional: int32, int64
- ML making popular int4, int8, int16

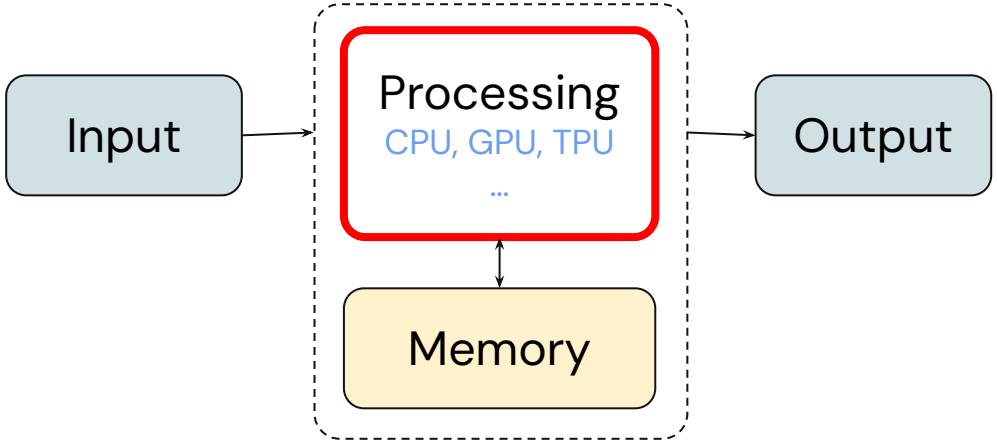
Rational (floating point):
- Traditional: fp32, fp64 (IEEE 754)
- ML making popular fp4, fp8, fp16
- ML specific: Brain float 16-bit (bf16)

FLOP/s: Floating-point operations per second

PetaFLOP: 10^15 FLOP
PetaFLOP-day (or PF-day): 86.4k PetaFLOP(s)

Basics of computer architectures

Let's look again at the Risc-V example




```
lw rs1, 0(t1)  # load 4 bytes from memory address in t1
lw rs2, 4(t1)  # load 4 bytes from address in t1+4
# add rd, rs1, rs2 # rd = rs1 + rs2
mul rd, rs1, rs2 # rd = (rs1 * rs2)[31:0]
sw rd, 0(t2)   # store rd value onto address t2
```

Execution time

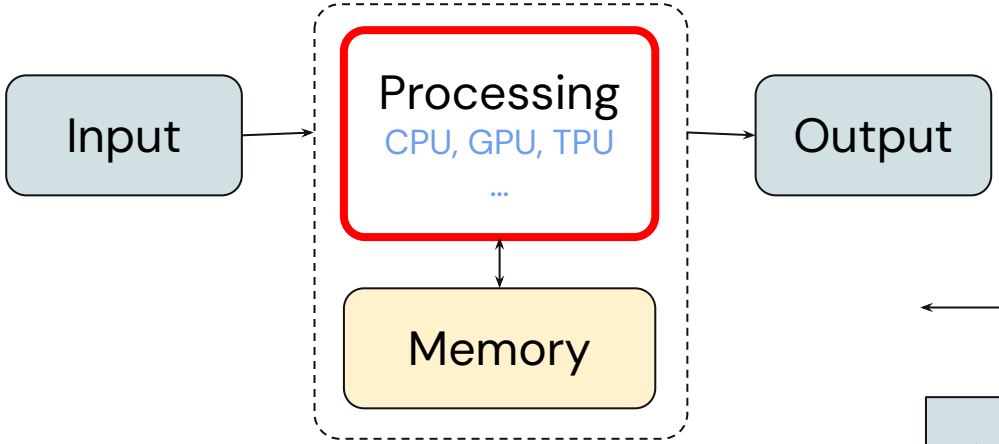
`add/mul` take one instruction cycle to complete.

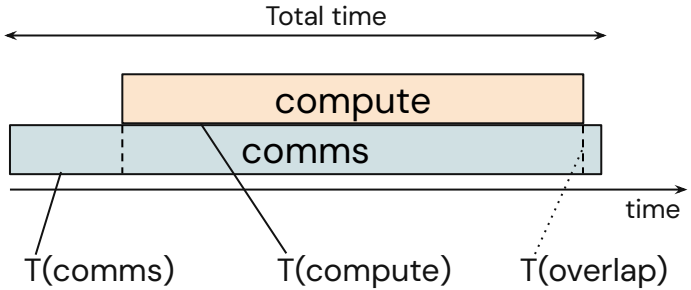
That's typically done in ~**1ns**!

`lw/sw` take a variable time, depending on the memory architecture. If the data is in DRAM, it's in the **~10ns** range (see CAS latency for more)

Basics of computer architectures

Computation vs communication





Computing dot-product

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

Home exercise: write the assembly instructions for dot-product

Total >= T(comms)
Total >= T(compute)
Total = T(comms) + T(compute) – T(overlap)

Communication bound?
Compute bound?

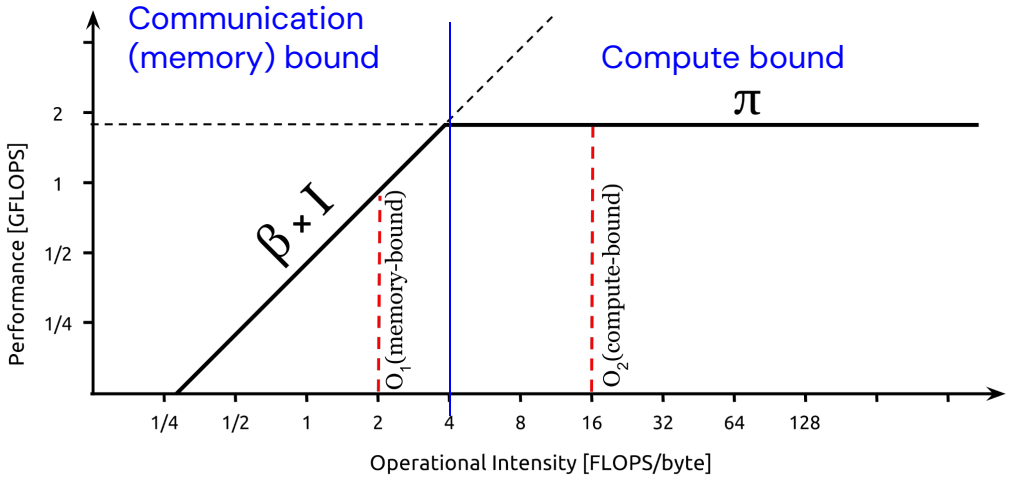
Basics of computer architectures

Understanding performance: the roofline model

I - Operational or Arithmetic intensity, measured in FLOPS/byte

$$I = \frac{W}{Q}$$

- no of *arithmetic* operations performed by a given computation
- no of bytes of memory transfers incurred during the execution of the computation



Arithmetic intensity of dot-product

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

W = n [multiplications] + (n − 1) [additions] = 2n−1
Q = n*2*bytes_per_scalar

Assuming bf16: bytes_per_scalar = 2; Q=4*n; **I** = (2n−1)/4*n ~ **1/2**

π - peak performance (FLOPS)
β - peak communication/memory bandwidth
For peak (FLOPS) performance: I >= π/β

For good utilisation/efficiency, we need the arithmetic intensity of our computations to be high!

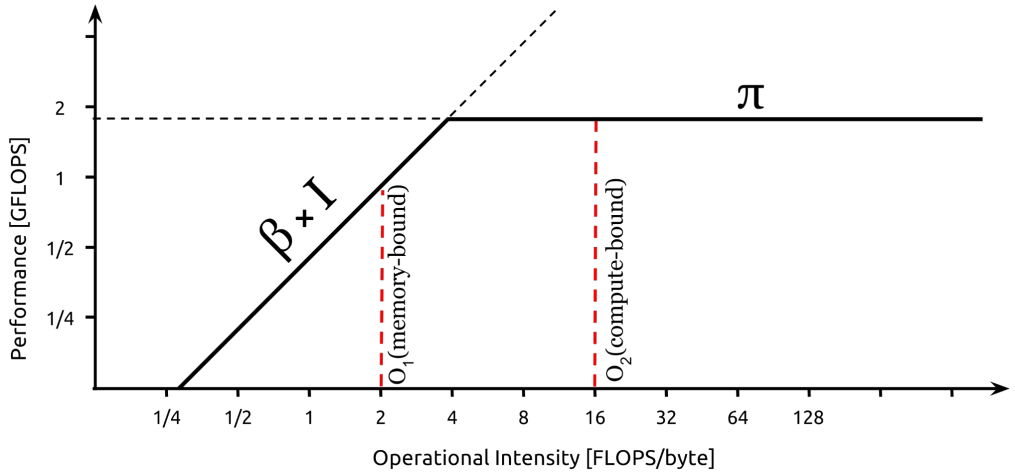
Basics of computer architectures

Arithmetic intensity of matrix multiplication

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj},$$



I = W / Q

W = 2 * m * n * p
Q = 2 * (m * p + n * p + m * n)

I = (m * n * p) / (m * p + n * p + m * n)

Larger matrices generally have higher arithmetic intensity!

So,... how do we make π bigger?

...much, much bigger?

Practicals

1. Compute the arithmetic intensity

2. Matrix multiplication
   - different sizes relationship to the roofline model
   - batch size and FLOPs vs communication bound

Summary of what we covered

- Basics of the Von Neumann architecture (CPU, Mem, input, output)
- Data/number representations (e.g.: int32, fp32, bf16, etc)
- Cost of communication vs computation
- Arithmetic/operational intensity
- Large matrices have good operational intensity!

Key takeaways

- Need lots of FLOP/s but that's not the only bottleneck.
- Communication/memory performance are very important.
- *Good news:* matmuls can have very high arithmetic intensity.

Side channel topics

- Systolic arrays

Next section: how to multiply very large matrices very fast 😎

Computer architectures
Why accelerators?

	TFLOP/s (peak)	Time to train model (in years)*	
		GPT-3	Llama-3 (405B)
Apple (MBA M3)	18	923.02	66,780.82
Intel (Core ultra)	48	346.13	25,042.81
Google TPU (Trillium)	918	18.10	1,309.43
NVIDIA GPU (H100)	2,000	8.31	601.03

We need to bring this down to a few weeks!

Specialization:
Accelerators enable faster execution of certain computations

How?

Earlier example for adding two numbers:

```
add rd, rs1, rs2
```

NVIDIA H100 (PTX) for matrix multiply(-accumulate) - i.e.: `D = AxB + C`

```
mma.sync.aligned.m8n8k4.alayout.blayout.dtype.f16.f16.ctype  d, a, b, c;
```

Computer architectures
Why accelerators?

	TFLOP/s (peak)
Apple (MBA M3)	18
Intel (Core ultra)	48

We need to bring this down to a few weeks!

How?

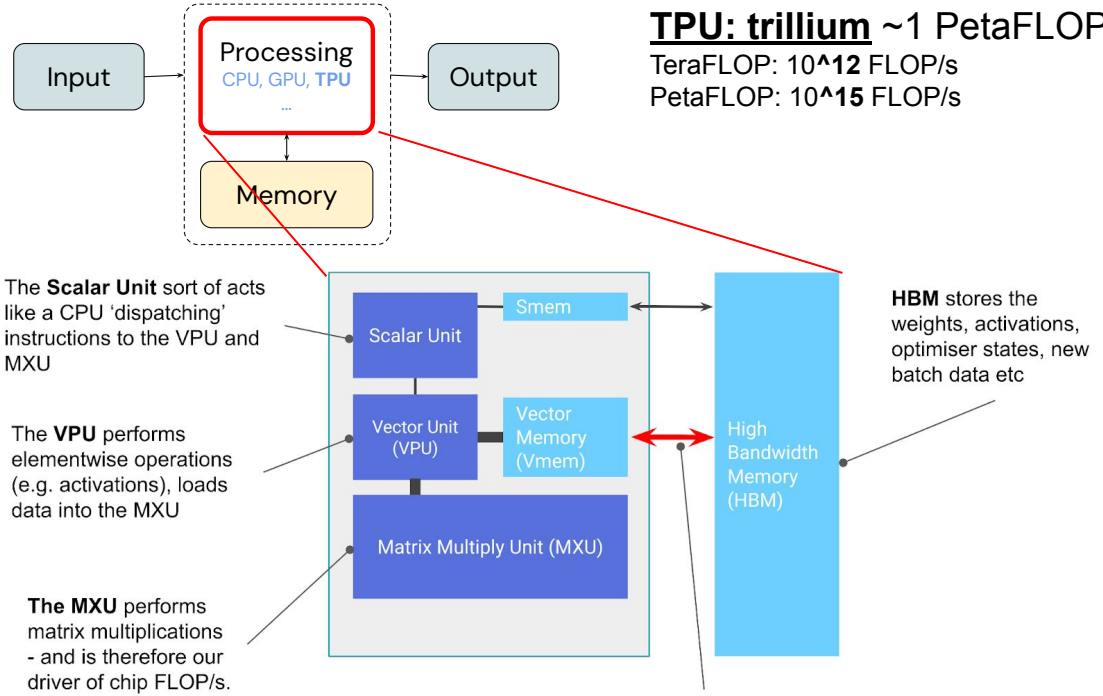
Specialization:
Accelerators enable faster execution of certain computations

Earlier example for adding two numbers:
```
add rd, rs1, rs2
```

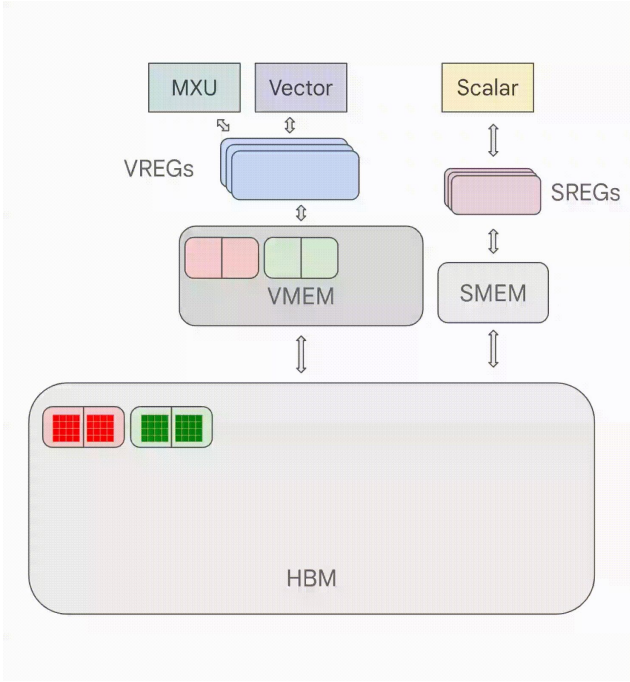
NVIDIA H100 (PTX) for matrix multiply(-accumulate) - i.e.: `D = AxB + C`
```
mma.sync.aligned.m8n8k4.alayout.blayout.dtype.f16.f16.ctype  d, a, b, c;
```

Accelerators: TPUs

How do we make π bigger? Use accelerators



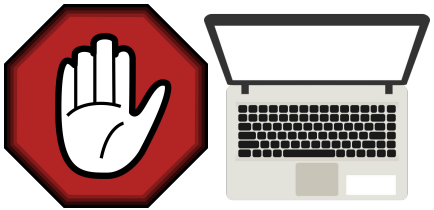
TPU: trillium ~1 PetaFLOP/s/chip
TeraFLOP: 10^**12** FLOP/s
PetaFLOP: 10^**15** FLOP/s



For most TPU generations, **MXU** performs one
`bfloat16[8,128] @ bf16[128,128] -> f32[8,128]`
matrix multiply every 8 cycles

Computer architectures

Training LLMs in datacentres



Computer architectures
Training LLMs in datacentres

LLaMA

"Fun" stats on training Llama-3

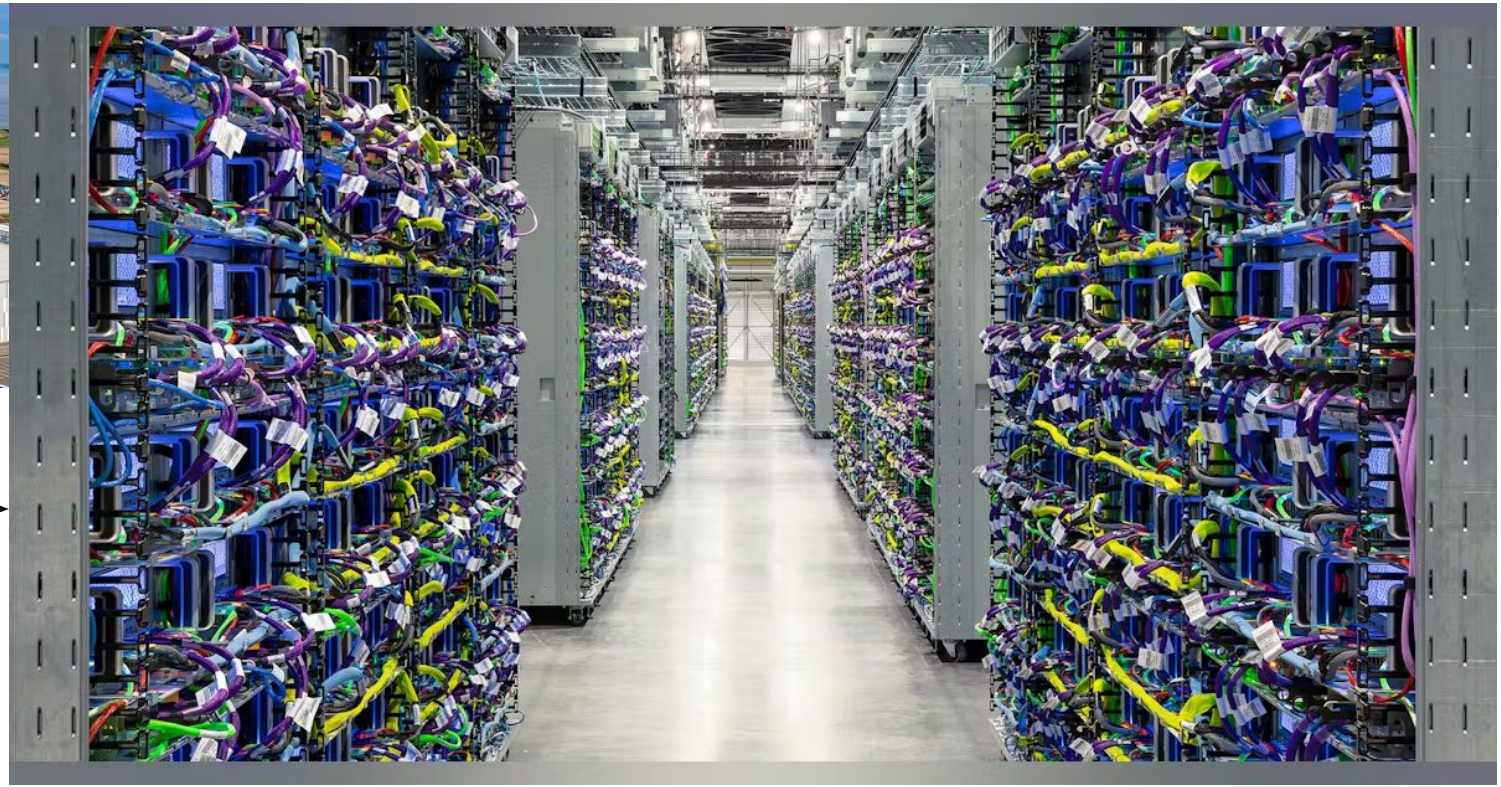
Trained using a cluster with **16k H100 GPUs**

Only GPUs in the cluster consume up to 11.2 MWatts
*enough to power homes/needs of **~30k people** in S. Africa.*

Accommodating 16k H100s requires building space with ~**500 m2**
larger than a basketball court

Supercomputers

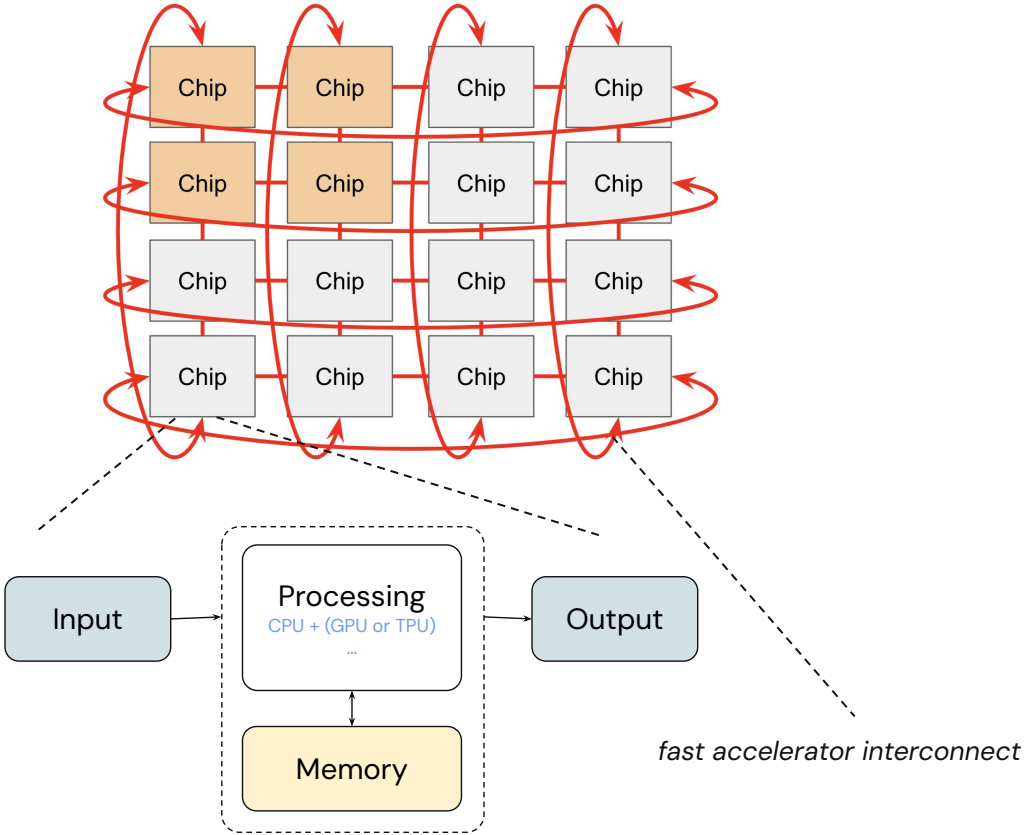
A quick look inside these buildings

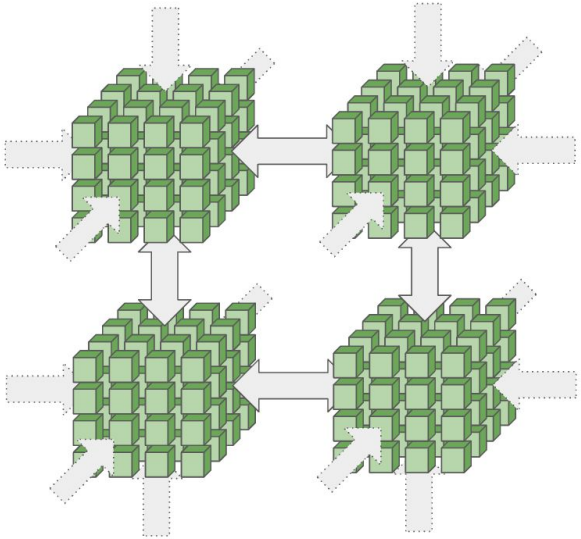


TPU systems inside a Google data centre

Computer architectures

Connecting lots (thousands) of accelerators to train LLMs



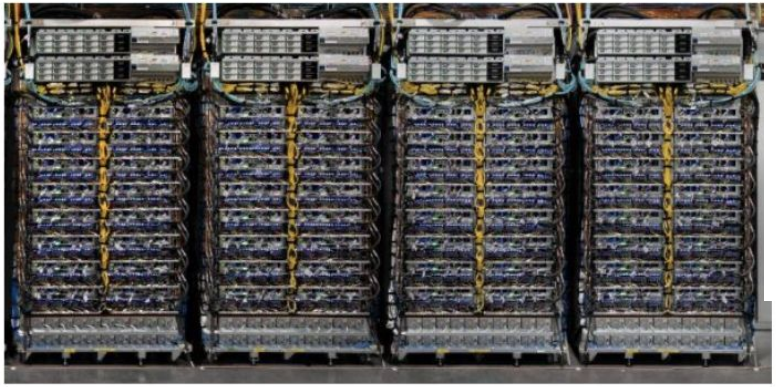


Key bottlenecks

1. Processing (speed)
2. Memory (capacity and speed)
3. Network interconnect

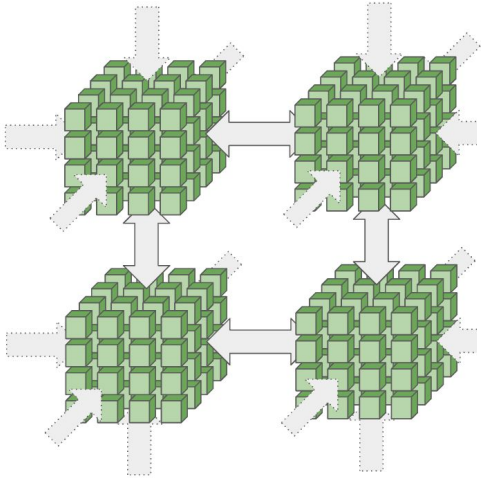
TPU supercomputers

*More details in the **Scaling book***

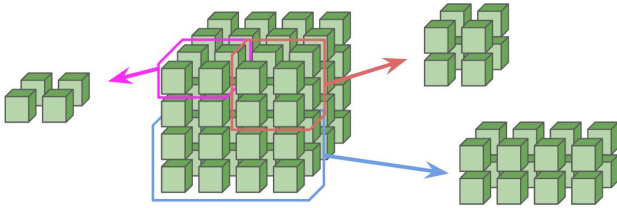


Each rack corresponds to 4x4x4 TPUv4s

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

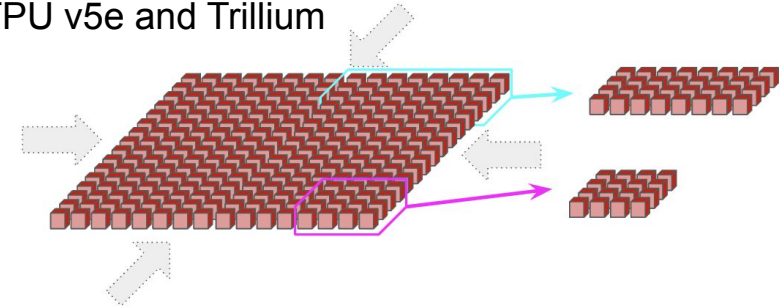


TPU v4p and v5p

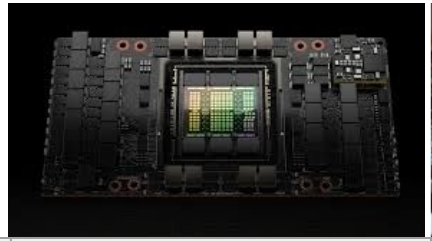


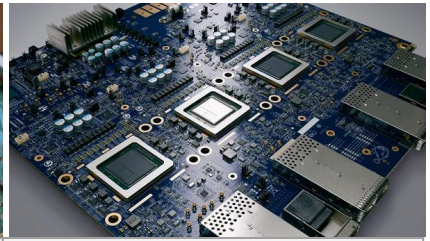
Smaller slices can also be requested.

TPU v5e and Trillium



Case studies

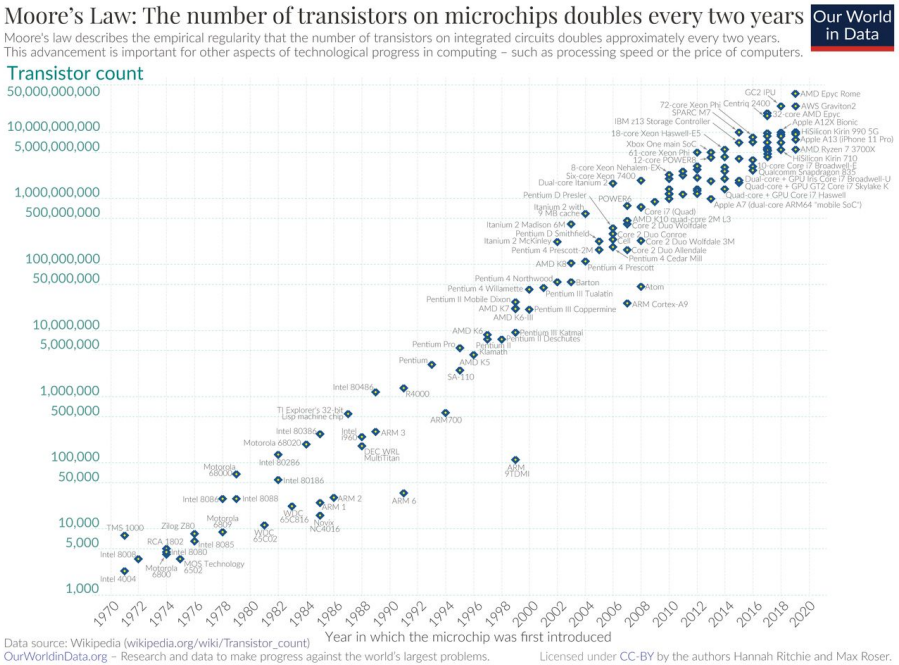




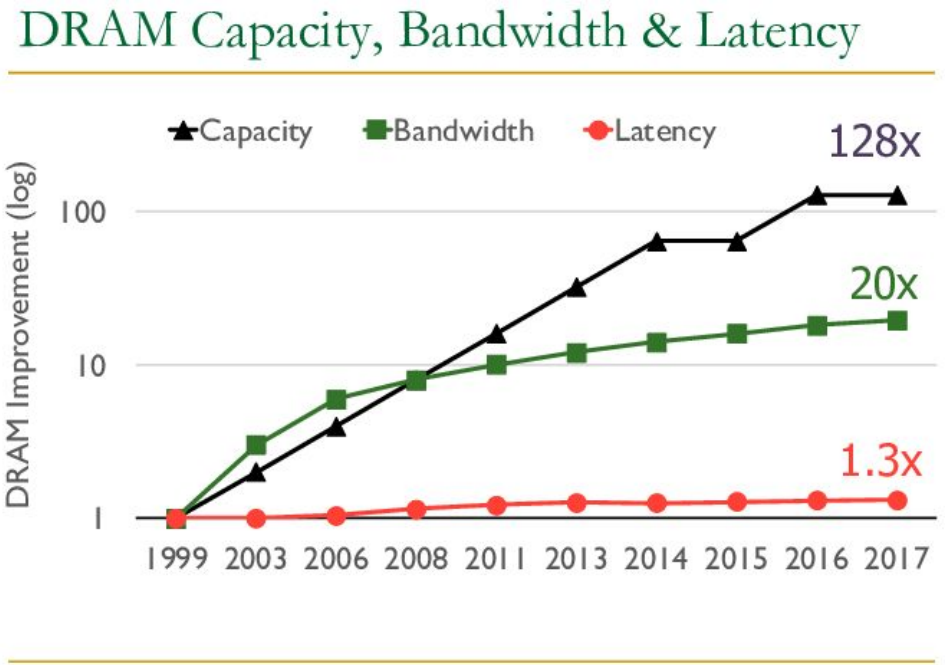
	NVIDIA H100 GPU (source)	Google TPU v5p (source)	Google Trillium TPU (source)
TeraFLOPS (fp16/bf16)	1,979	459	926
Accelerator memory (HBM3)	80GB	~95GiB	32GB
Memory bandwidth	3.35TB/s	2.77TB/s	1.64TB/s
Interconnect	8 fully connected GPUs with NVIDIA NVLink™: 900GB/s (can connect more with NVLINK switch)	16x16x32 4,800 Gbps (can connect even more via ethernet)	16x16 2D torus 3,200 Gbps (can connect more via ethernet)

Side channel topics

Trends that have been great for AI & technology



Other trends are not as great…

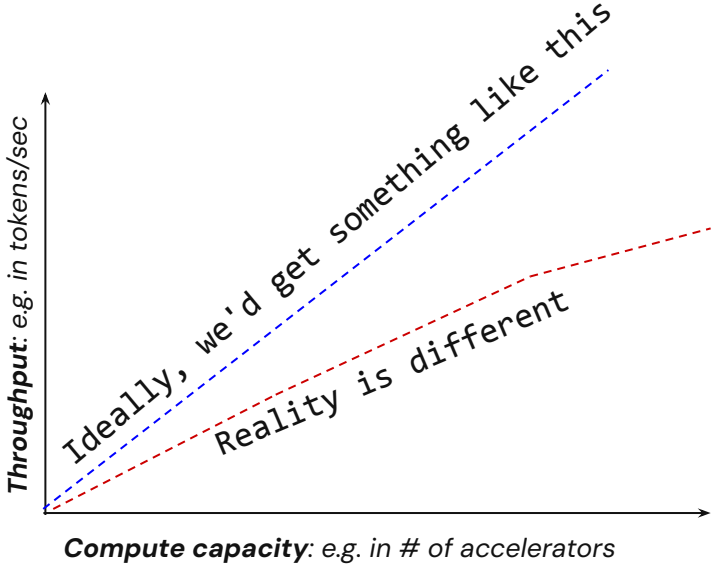


How do we scale compute for training LLMs?

What does that mean?

Strong scaling
The property we want:
training speed (throughput) increases linearly with the number of accelerators used for training.

Throughput: tokens/sec (vs steps/sec)



Key bottlenecks (again)

1. Processing (speed)
2. Memory (capacity and speed)
3. Network interconnect

More bottlenecks (that we'll ignore here)

- Accelerator <-> CPU interconnect
- Storage speed
- …

- 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.

Scaling training

1. **Our baseline:** Batch parallelism (& single chip training)
2. Data parallelism
3. FSDP (Fully sharded Data parallelism)
4. Advanced topics: megatron and pipeline parallelism

Training transformers + batch parallelism

The basics of transformers

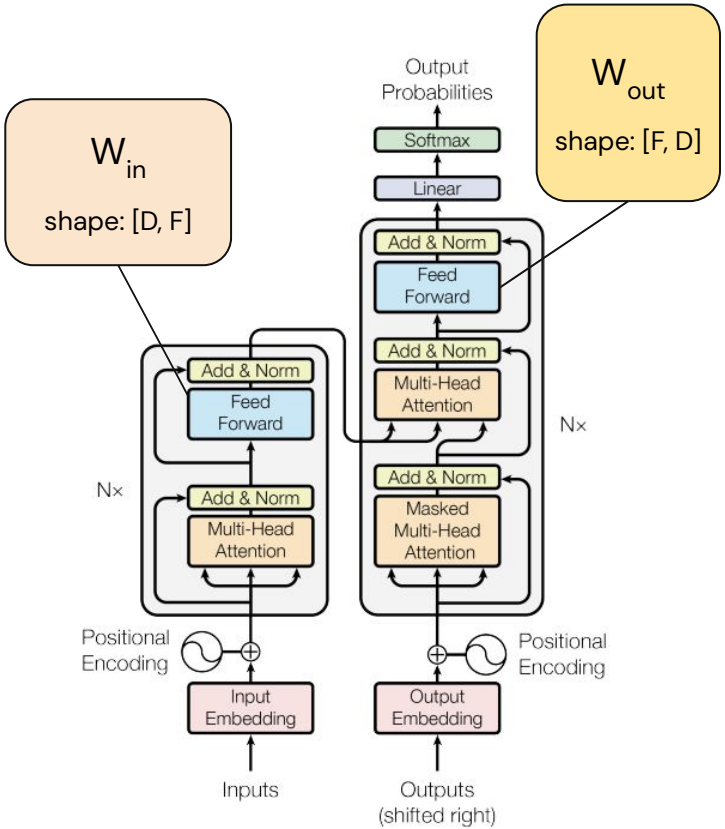


Figure 1: The Transformer - model architecture.

The key approximation to remember

Out[D] = In[D] * W_{in}[D, F] * W_{out}[F, D]

The basics of transformers

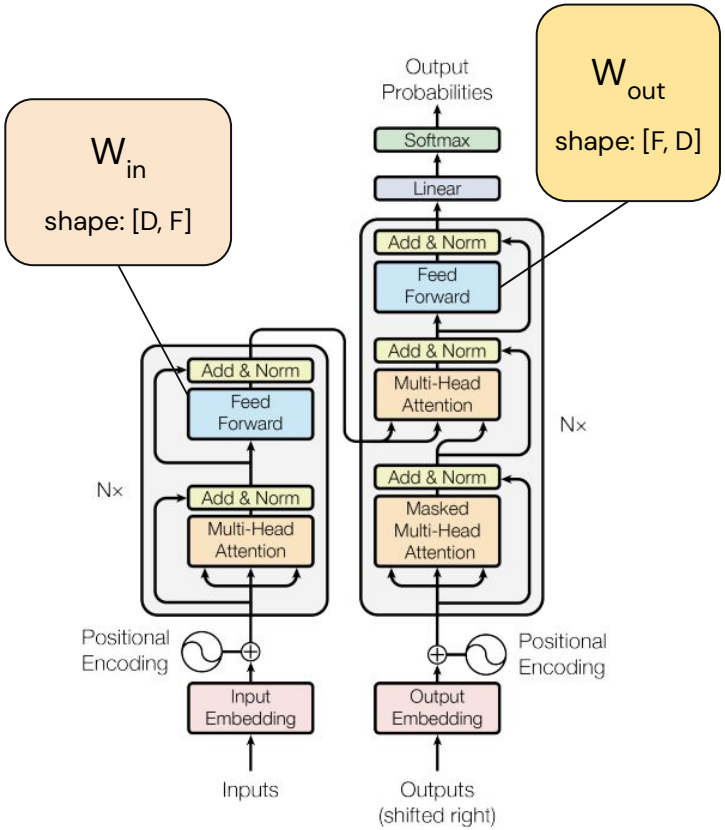


Figure 1: The Transformer - model architecture.

The key approximation to remember

Out[D] = In[D] * W_{in}[D, F] * W_{out}[F, D]

For more details on transformers watch this

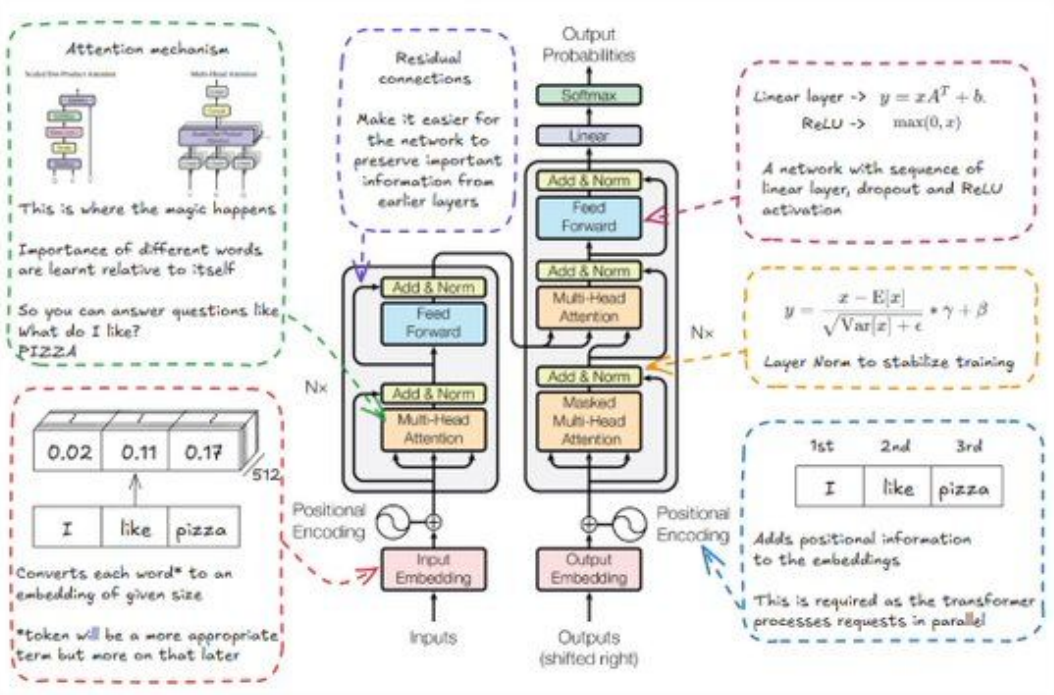
Transformers

Model Architecture

The section all of us had been waiting for. I will divert a bit from the paper here. Because I find it easier to follow the data. Also if you read the paper, each word of it should make sense to you.

We will first start with the Multi-Head Attention, then the feed forward network, followed by the positional encoding, Using these we will finish the Encoder Layer, subsequently we will move to the Decoder Layer, After which we will write the Encoder & Decoder block, and finally end it with writing the training loop for an entire Transformer on real world data.

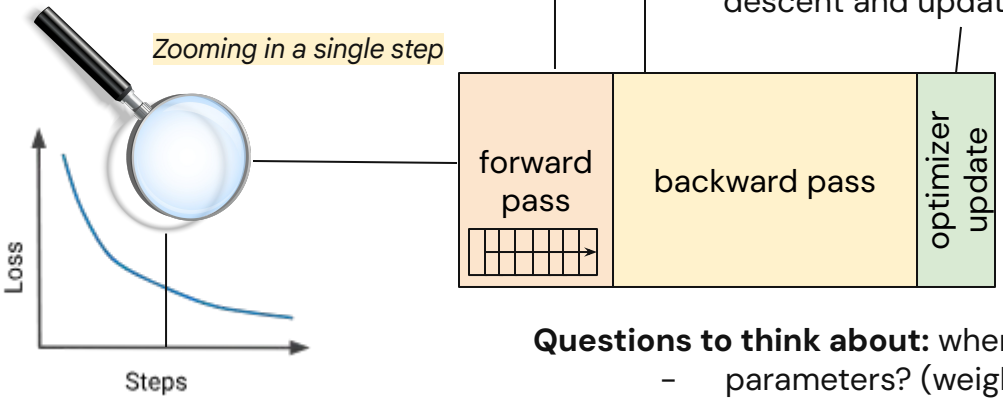
The full notebook can be found here



The basics of training a model

Assumes you know what forward/backward/optimizer updates are.
If you're looking for a refresher, see e.g.: https://xnought.github.io/backprop-explainer/

For any training step
For given input data and network weights/biases, calculate output and loss
Backward propagation using the chain rule to compute gradients
[If using SGD] Descend the loss by performing gradient descent and update weights/biases.



Questions to think about: where and for how long do we store
- parameters? (weights, biases)
- activations?
- gradients?
- optimizer state? (in case of e.g. Adam)

General rule of thumb for calculating FLOPs when training Transformers

```
FLOPs ≈ 6 * num_tokens * param_count
```

(NB: ignores the attention FLOPs used during training; ok for large models)

Batch parallelism

Where are the matmuls?

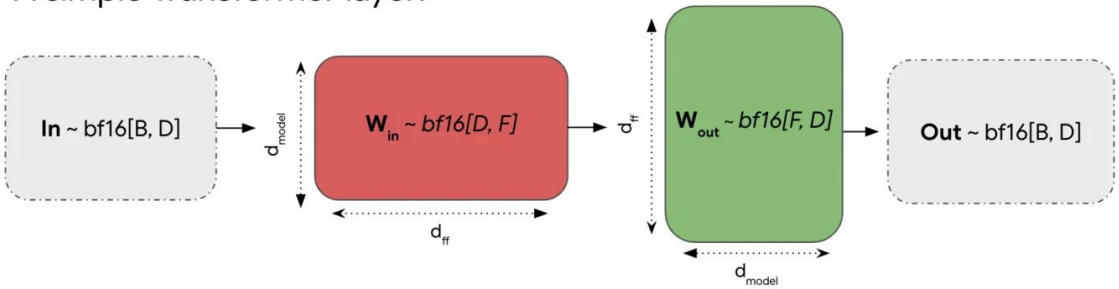
Out[D] = In[D] * W_{in}[D, F] * W_{out}[F, D]

How do we make ***Input * W_{in}*** have high arithmetic intensity?

Batch parallelism

Out[B, D] = In[B, D] * W_{in}[D, F] * W_{out}[F, D]

A simple Transformer layer:



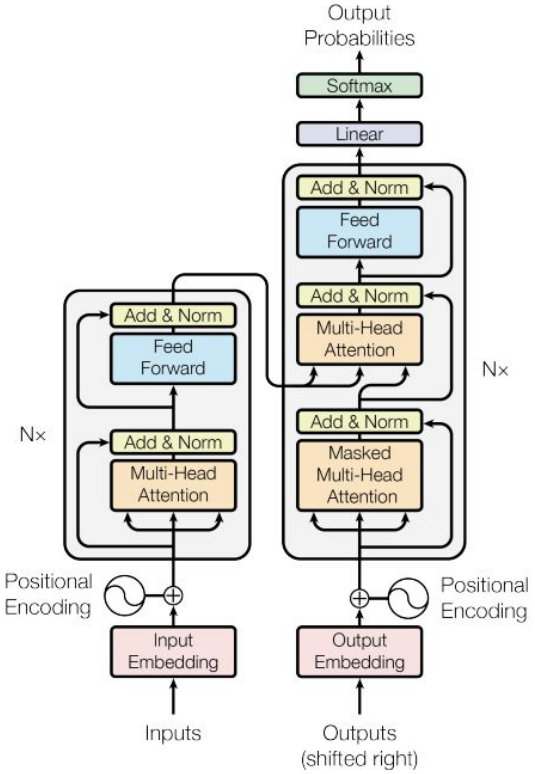


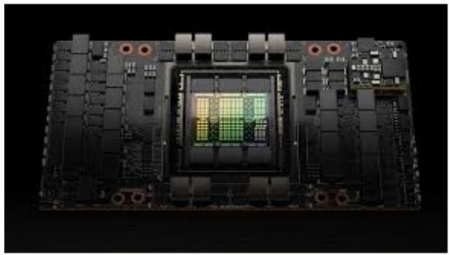
Figure 1: The Transformer - model architecture.

Min batch size that maximizes utilization (in a single chip)

Reminder 1: Arithmetic intensity of matrix multiplication!

Reminder 2: Min intensity to reach peak performance

Reminder 3: Accelerator spec (e.g. NVIDIA H100)



Arithmetic intensity for ***In[B, D] * W$_{in}$[D, F]****:*

I = (B*D*F) / (B*D + B*FF + D*F)

Assuming B << D and B << F, then **I ≈ B**

$I > \pi/\beta$ => **B >** π/β

H100 performance spec:
- Peak FP16 FLOPs (π): 1979 TFLOPs
- HBM Bandwidth(β): 3.35 TB/s

B > π/β => **B >** 1979/3.35 = 590.746 (~600)

	NVIDIA H100 GPU (source)
TeraFLOPS (fp16/bf16)	1,979
Accelerator memory (HBM3)	80GB
Memory bandwidth	3.35TB/s

I = (m * n * p) / (m * p + n * p + m * n)

Larger matrices generally have higher arithmetic intensity!

Exercise: *calculate the min batch size that maximizes utilization of Google Trillium TPU!*

Key takeaways:

- Batch parallelism is one of the most essential methods that enables us to increase accelerator utilization
- Transformers are typically always trained with batch parallelism
- The minimum batch size required to maximize utilization of a given accelerator can be calculated as **B > peak_FLOPS (π)/ HBM_BW (β)**

Side channel topics

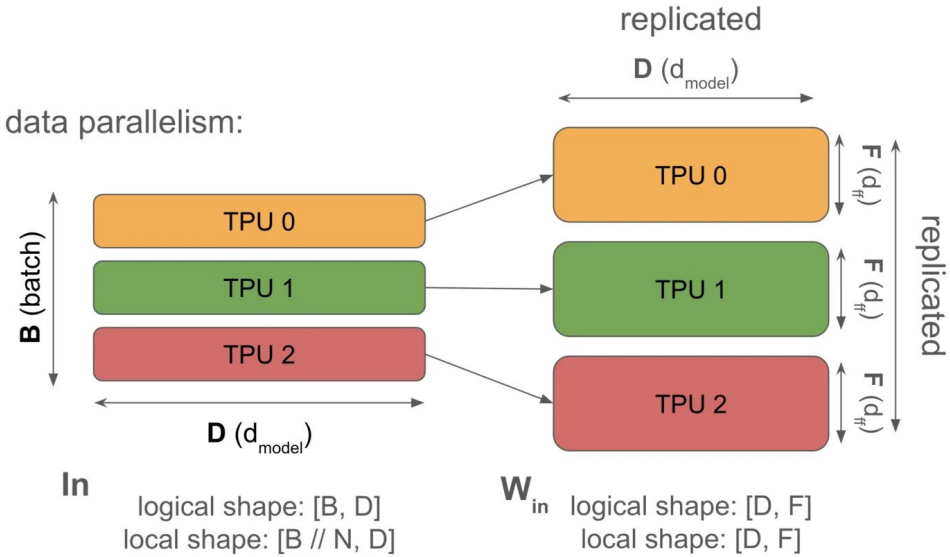
- See https://huggingface.co/blog/lsayoften/optimization-rush for some fun details.

Next section: scaling batch parallelism beyond a single chip

Data parallelism

and adventures in multiplying sharded matrices

Overview



In[B, D] *$_D$ W_{in}[D, F] *$_F$ W_{out}[F, D] -> Out[B, D]

In[Bx, D] *$_D$ W_{in}[D, F] *$_F$ W_{out}[F, D] -> Out[Bx, D]

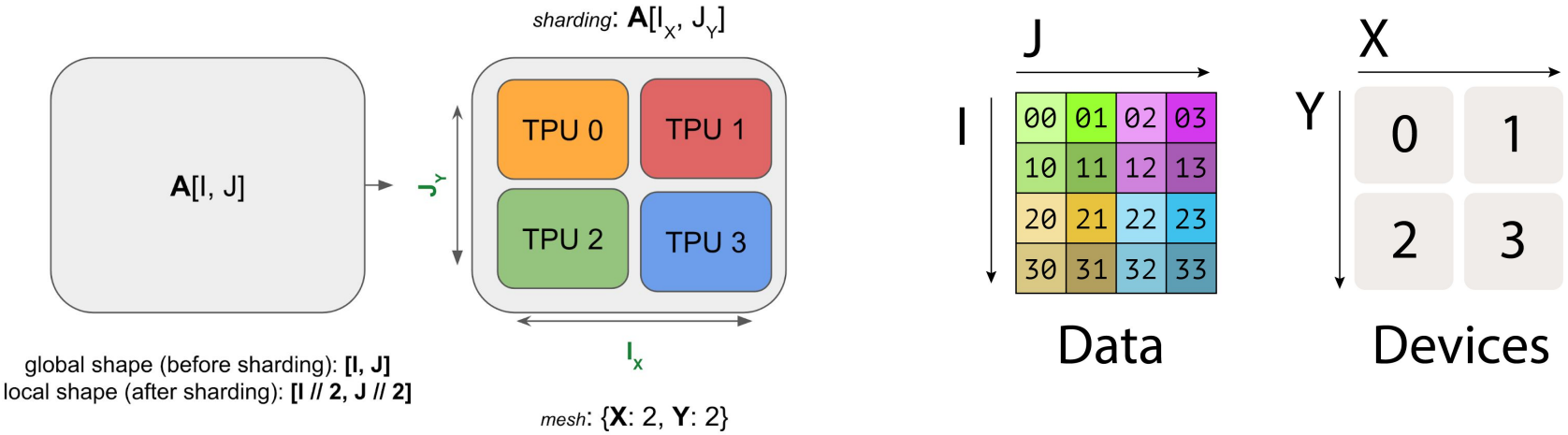
What happens to activations?

They're sharded(?) across the batch dimension, reducing memory pressure.

When to use: as long as your model fits in a single chip with a batch size that keeps training compute-bound

Note: Sequence/context parallelism: just another form of DP, where we shard batch over both the batch and sequence dimensions.

Sharded matrices



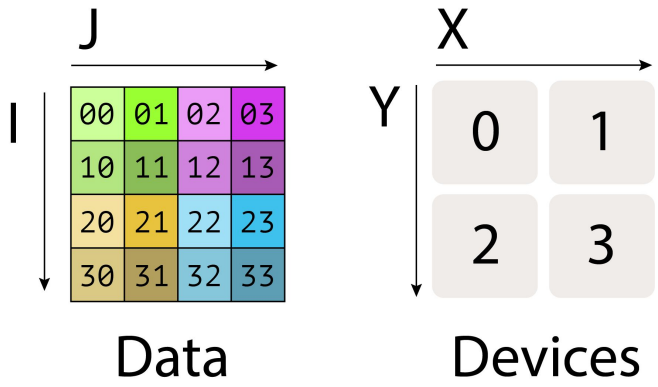
Sharding: A[I_X,J_Y] which tells us to shard the first axis, I, along the mesh axis X and the second axis, J, along the mesh axis Y.

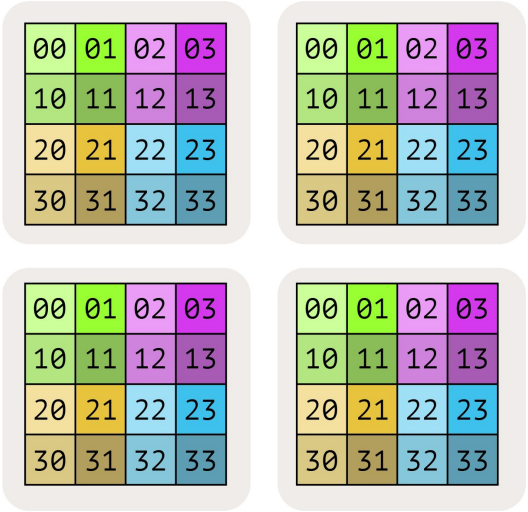
This sharding tells us that each shard holds 1/(|X| * |Y|) of the array.

Mesh: the device mesh above Mesh(devices=((0, 1), (2, 3)), axis_names=('X', 'Y')), which tells us we have 4 TPUs in a 2x2 grid, with axis names X and Y.

Sharded matrices

Visualizing shardings

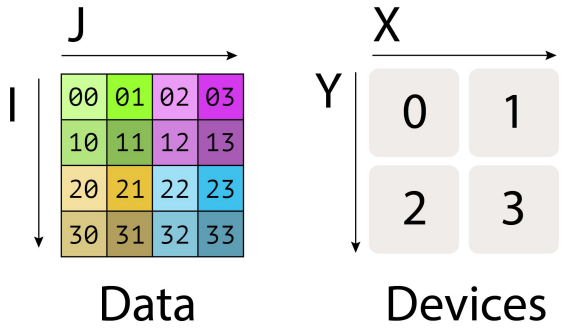


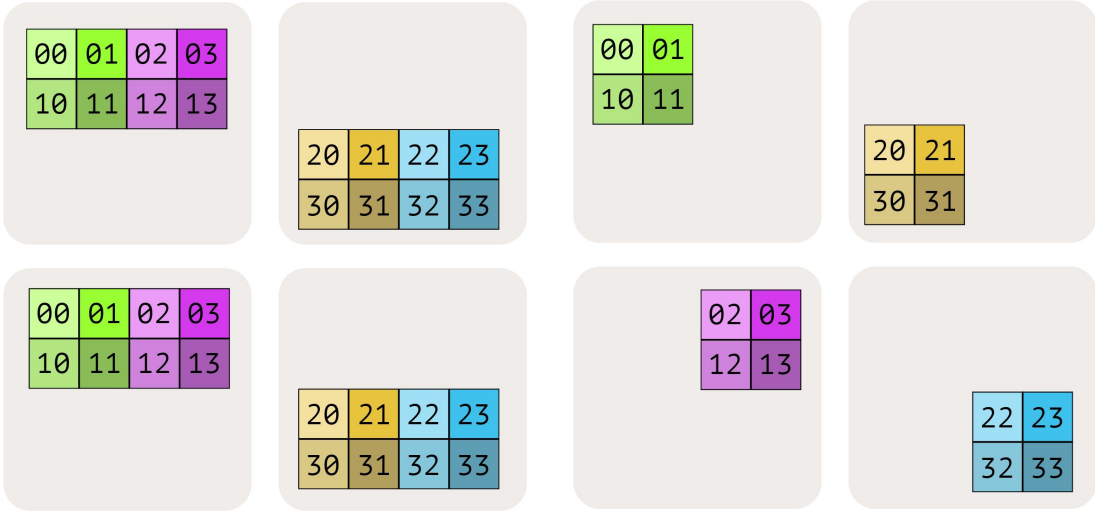


Fully-replicated
Notation: simply as **A[I,J]** with no sharding assignment

Sharded matrices

Visualizing shardings



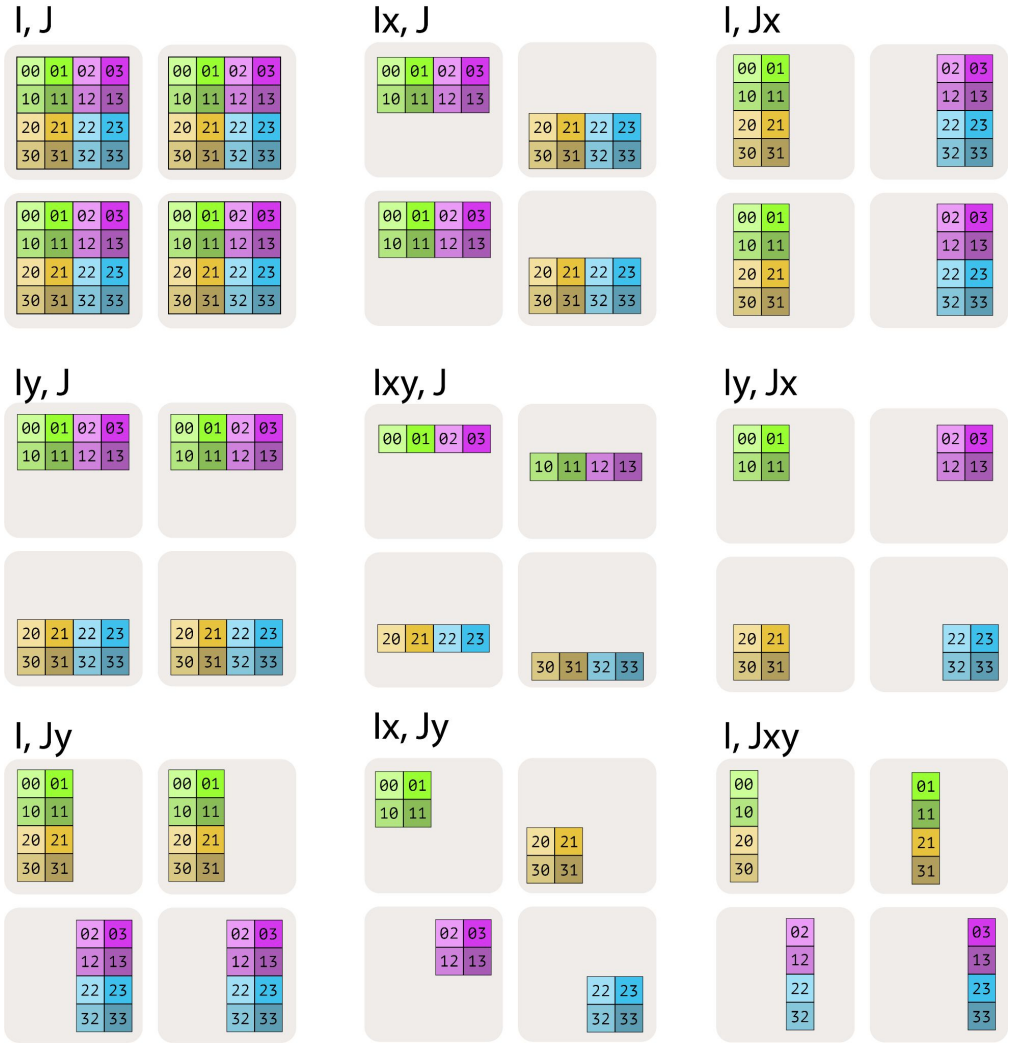


Partially-replicated
Notation: **A[I_X,J]**

Fully-partitioned
Notation: **A[I_X,J_Y]**

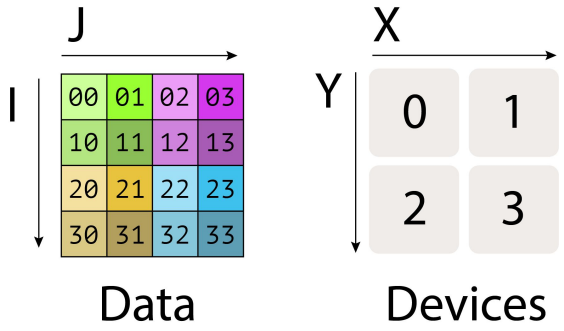
Sharded matrices

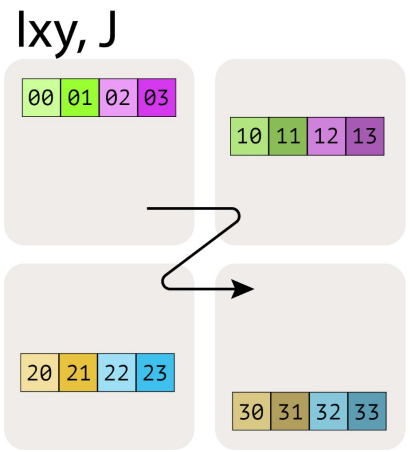
Visualizing shardings

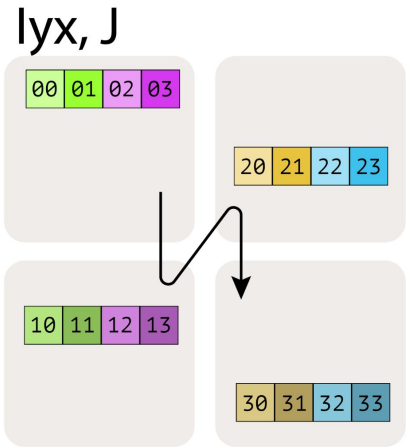


Sharded matrices

Visualizing shardings







Quick exercise: Let A be an array with shape `bf16[256, 2048]`, sharding $A[I_{XY}, J]$, and mesh `Mesh({'X': 4, 'Y': 8, 'Z': 2})`(so 64 devices total).
1. How much memory does A use per device?
2. How much total memory does A use across all devices?

Sharding matrix multiplications

$$A = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1q} \\ A_{21} & A_{22} & \cdots & A_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ A_{p1} & A_{p2} & \cdots & A_{pq} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} & \cdots & B_{1s} \\ B_{21} & B_{22} & \cdots & B_{2s} \\ \vdots & \vdots & \ddots & \vdots \\ B_{r1} & B_{r2} & \cdots & B_{rs} \end{bmatrix}$$

$$C = AB = \begin{bmatrix} \sum_{i=1}^{q} A_{1i}B_{i1} & \sum_{i=1}^{q} A_{1i}B_{i2} & \cdots & \sum_{i=1}^{q} A_{1i}B_{is} \\ \sum_{i=1}^{q} A_{2i}B_{i1} & \sum_{i=1}^{q} A_{2i}B_{i2} & \cdots & \sum_{i=1}^{q} A_{2i}B_{is} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^{q} A_{pi}B_{i1} & \sum_{i=1}^{q} A_{pi}B_{i2} & \cdots & \sum_{i=1}^{q} A_{pi}B_{is} \end{bmatrix}$$

$$\begin{bmatrix} A_1 \\ A_2 \\ \cdots \\ A_n \end{bmatrix} [B] = \begin{bmatrix} A_1 B \\ A_2 B \\ \cdots \\ A_n B \end{bmatrix}$$

In[B_X, D] *$_D$ W1[D, F] *$_F$ W2[F, D] -> Out[B_X, D]

$$\begin{bmatrix} IN_1 \\ IN_2 \\ \cdots \\ IN_x \end{bmatrix} * W_1 * W_2 = \begin{bmatrix} IN_1 * W_1 * W_2 \\ IN_2 * W_1 * W_2 \\ \cdots \\ IN_x * W_1 * W_2 \end{bmatrix}$$

Sharding matrix multiplications

1: $A[I_X,J] \cdot B[J,K_Y] \rightarrow C[I_X,K_Y]$

2: $A[I,J_X] \cdot B[J,K] \rightarrow C[I,K]$

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

4: $A[I_X,J] \cdot B[J,K_X] \rightarrow \{C[I_X,K] \text{ or } C[I,K_X]\}$

Sharding matrix multiplications

AllGather

1: $A[I_X,J] \cdot B[J,K_Y] \rightarrow C[I_X,K_Y]$

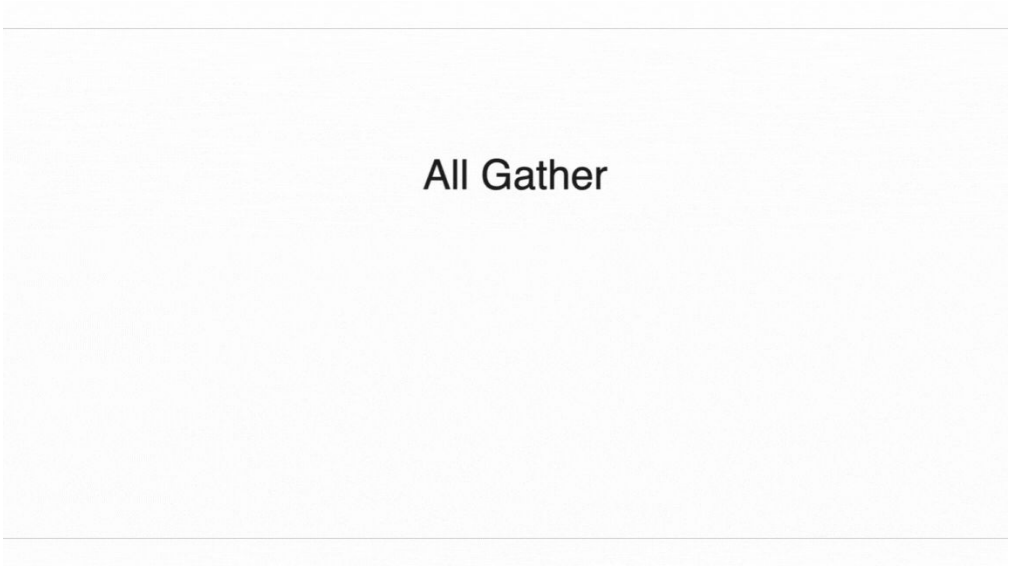
TLDR: neither input is sharded along the contracting dimension. We can multiply local shards without any communication.

2: $A[I,J_X] \cdot B[J,K] \rightarrow C[I,K]$

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

$AllGather_X A[I,J_X] \rightarrow A[I,J]$;
$A[I,J] \cdot B[J,K] \rightarrow C[I,K]$

How long does AllGather take?
total_time = matrix_size / nw_bw



Sharding matrix multiplications

AllReduce

1: $A[I_X,J] \cdot B[J,K_Y] \rightarrow C[I_X,K_Y]$

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

2: $A[I,J_X] \cdot B[J,K] \rightarrow C[I,K]$

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

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

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

{Ux} means that C[I,K] is **unreduced** along mesh X axis (i.e. *incomplete*).

LOCAL means that only local computations have been done

$\text{AllReduce}_X C[I,K]$ {Ux} -> C[I,K]

AllReduce removes partial sums whereby every device sends its shard to its neighbors, and sums up all the shards that it receives. Typically expressed as composition of ReduceScatter and AllGather.

Sharding matrix multiplications

AllReduce, ReduceScatter

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

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

{Ux} means that C[I,K] is **unreduced** along mesh X axis (i.e. *incomplete*).

LOCAL means that only local computations have been done

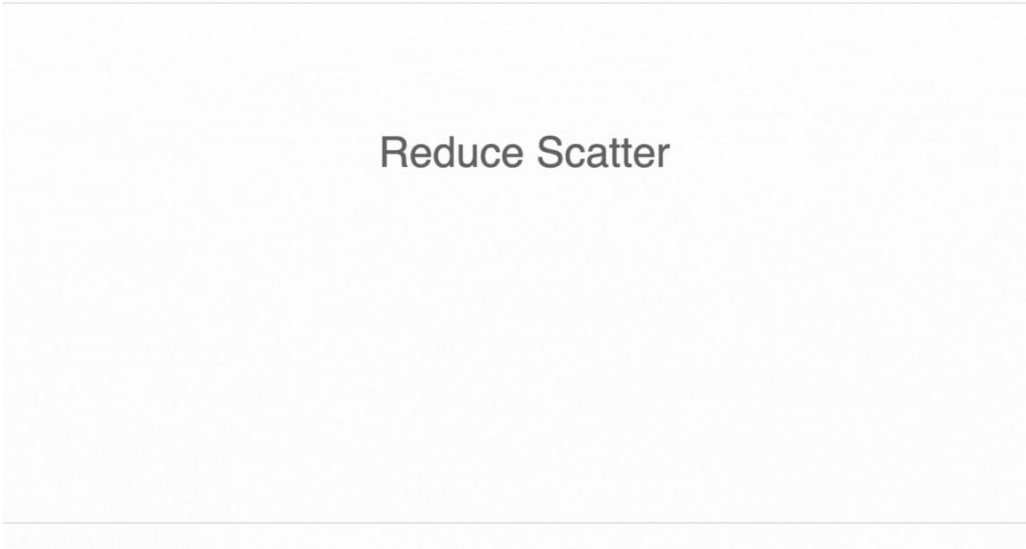
$AllReduce_X C[I,K]$ {Ux} -> C[I,K]

AllReduce removes partial sums whereby every device sends its shard to its neighbors, and sums up all the shards that it receives.
Typically expressed as composition of ReduceScatter and AllGather.
ReduceScatter sums an unreduced/partially summed array and then scatters (shards) a different logical axis along the same mesh axis.

$ReduceScatter_{Y,J} A[I_X,J]$ {U_Y} -> $A[I_X,J_Y]$

$AllGather_Y$: $A[I_X,J_Y] \rightarrow A[I_X,J]$

How long does AllReduce take?
total_time = 2 * matrix_size / nw_bw



Sharding matrix multiplications

1: $A[I_X,J] \cdot B[J,K_Y] \rightarrow C[I_X,K_Y]$

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

2: $A[I,J_X] \cdot B[J,K] \rightarrow C[I,K]$

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

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

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

{Ux} means that C[I,K] is unreduced along mesh X axis (i.e. *incomplete*).

LOCAL means that only local computations have been done

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

4: $A[I_X,J] \cdot B[J,K_X] \rightarrow \{C[I_X,K]$ or $C[I,K_X]\}$

$AllGather_X A[I_X,J] \rightarrow A[I,J];\ A[I,J] \cdot B[J,K_X] \rightarrow C[I,K_X]$ **or**

$AllGather_X B[J,K_X] \rightarrow B[J,K];\ A[I_X,J] \cdot B[J,K] \rightarrow C[I_X,K]$

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

Practical

Sharded Matrix Multiplication and Collectives in JAX

The colab covers sharded array and matrix multiplication with a performance analysis

How to make data parallelism work

$In[B_X, D] *_D W_{in}[D, F] *_F W_{out}[F, D] \rightarrow Out[B_X, D]$

Forward pass calculations [simple]

Need to compute $Loss[B_X]$

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

TLDR: *each node/accelerator performs computation in its own shard of batch (i.e. no communication)*

Backward pass calculations:

Need to compute $dW_{out}[F, D]$, $dW_{in}[D, F]$

How to make data parallelism work

$In[B_X, D] *_D W_{in}[D, F] *_F W_{out}[F, D] \rightarrow Out[B_X, D]$

Forward pass calculations [simple]

Need to compute $Loss[B_X]$

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

TLDR: *each node/accelerator performs computation in its own shard of batch (i.e. no communication)*

Backward pass calculations:

Need to compute $dW_{out}[F, D]$, $dW_{in}[D, F]$

Let: $Tmp = W_{in} \cdot In$

1. $dOut[B_X, D] = \ldots$
2. $dW_{out}[F, D]\ \{U_X\} = Tmp[B_X, F]^T *_B dOut[B_X, D]$
3. $dW_{out}[F, D] = \text{AllReduce}(dW_{out}[F, D]\ \{U_X\})$
4. $dTmp[B_X, F] = dOut[B_X, D] *_D W_{out}[F, D]^T$
5. $dW_{in}[D, F]\ \{U_X\} = In[B_X, D]^T *_B dTmp[B_X, F]$
6. $dW_{in}[D, F] = \text{AllReduce}(dW_{in}[D, F]\ \{U_X\})$
7. $dIn[B_X, D] = dTmp[B_X, F] *_F W_{in}[D, F]^T$
   (needed for previous layers)

TLDR: *requires all gradients to be present in all nodes/accelerators*

Assuming:

- B a matrix in a larger network (i.e. B -> B' -> B" -> ….);
- A are our input activations;
- **C = A * B**

The derivative of the loss **L** with respect to B and A are given by the chain rule:

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

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

Note: 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).

Key point: we need to make available parameters that we have previously sharded to complete the backwards pass!

Data parallelism

$In[B_X, D] *_D W_{in}[D, F] *_F W_{out}[F, D] \rightarrow Out[B_X, D]$

Forward pass calculations [simple]

To compute $Loss[B_X]$

1. $Tmp[B_X, F] = In[B_X, D] *_D Win[D, F]$
2. $Out[B_X, D] = Tmp[B_X, F] *_F Wout[F, D]$
3. $Loss[B_X] = \ldots$

Backward pass calculations:

Need to compute dWout[F, D], dWin[D, F]

Let: $Tmp = W_{in} \cdot In$

1. $dOut[B_X, D] = \ldots$
2. $dW_{out}[F, D]$ **{U_X}** $= Tmp[B_X, F]^T *_B dOut[B_X, D]$
3. $dW_{out}[F, D] =$ **AllReduce**$(dWout[F, D]$ **{U_X}**$)$
   (not on critical path, can be done async)
4. $dTmp[B_X, F] = dOut[B_X, D] *_D Wout[F, D]^T$
5. $dW_{in}[D, F]$ **{U_X}** $= In[B_X, D]^T *_B dTmp[B_X, F]$
6. $dW_{in}[D, F] =$ **AllReduce**$(dWin[D, F]$ **{U_X}**$)$
   (not on critical path, can be done async)
7. $dIn[B_X, D] = dTmp[B_X, F] *_F Win[D, F]^T$
   (needed for previous layers)

How to stay compute bound with data-parallelism?

I.e. how to avoid being communication bound? Let

C = chip flops; **W** = bidirectional network bandwidth; **X** = number of shards across which the batch is partitioned; **B** = batch size

T = sequence length; **L** = no of layers, **D** = embedding dimension, **F** = MLP hidden dimension

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

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

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

We are compute bound when (B * T) / (X * C) > 1 / W

I.e.: to stay compute bound, chose X <= (W * B * T) / C

Limitations of data parallelism

- Duplicates a lot of work

  Each accelerator AllReduces the full gradient, then updates the full optimizer state (identical work on all accelerators), then updates the parameters (again, fully replicated).

- Requires keeping weights and optimizer state in HBM
  - Weights: model_size in bf16 - i.e. 2 Bytes/parameter
  - Optimizer state (Adam): first and second order accumulators in fp32 - (4+4) bytes/parameter
  - Total size per parameter: 10Bytes - implies limits to model size being ~size_of(HBM)/10

- Relies on large batch sizes
  - There are limits to how large one can make the batch size!
    (Model training suffers from very large batch sizes)

Practicals

Data parallel training of transformers

The colab takes the transformer architecture we implemented earlier in the class and trains it with data parallel training using the jax pmap API, with (jax.lax.pmean for gradients communication).

Key takeaways

Pros

- DP: the most *simple* and common form of scaling training compute

Cons

- Memory: the largest model we can train with data parallelism:

  `num_params = HBM per device / 10` *(assuming optimizers like Adam)*

  (TPU v5p: 9B params; H100: 8B params)

- Work duplication, large batch sizes

Side channel topics

- A Deeper Dive into TPU Communicaton Primitives

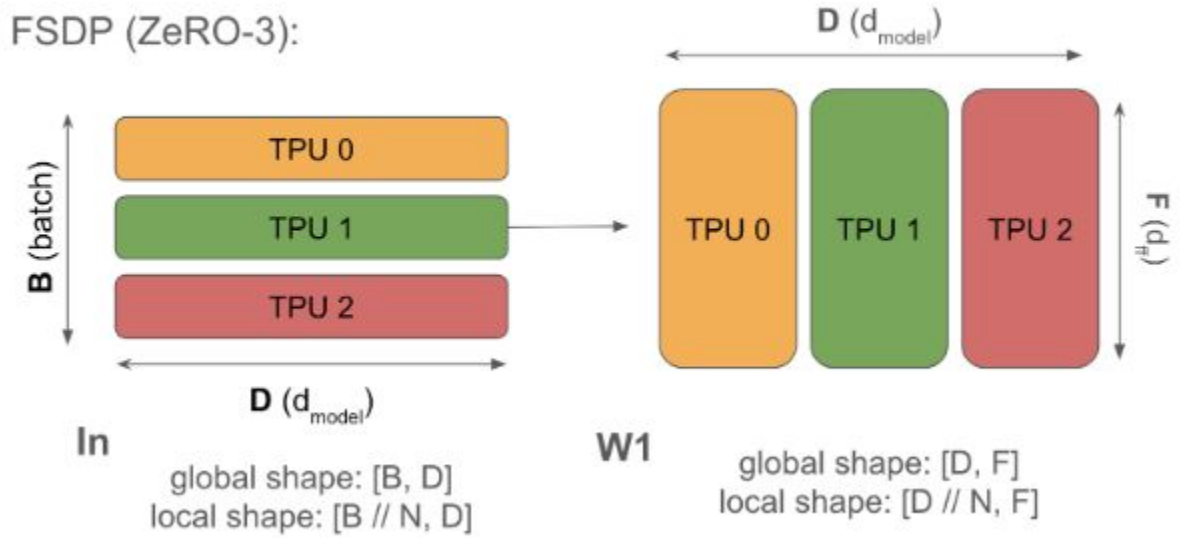
Next section: Fully sharded data parallelism

Fully Sharded Data Parallelism

Overview

Approach: Shard the weights and model optimizer state across the same axis as batch (data parallel) and efficiently gather and scatter these as needed.

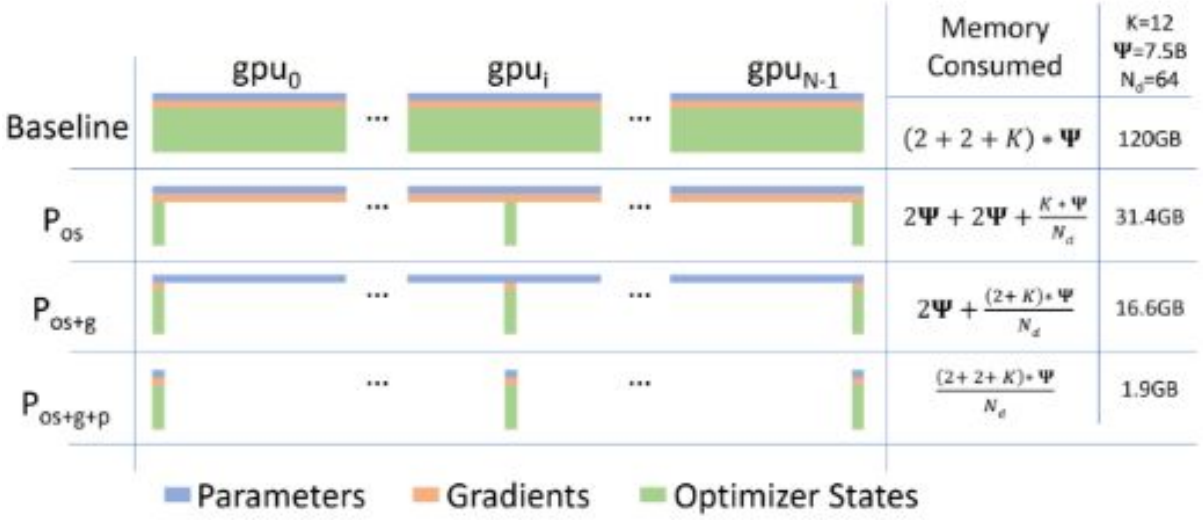
In[B_x, D] *$_D$ **W**$_{in}$[D_X, F] *$_F$ **W**$_{out}$[F, D_X] -> **Out**[B_x, D]



Also called: ZeRO sharding, ZeRO Overhead sharding
ZeRO-{1,2,3} are used to refer to sharding the optimizer states, gradients, and weights in this way, respectively.

FSDP efficiency improvements

Compared to pure data-parallelism, FSDP *drastically reduces per-device memory usage and saves on backward pass FLOPs, with very minimal overhead.*



Also called: ZeRO sharding, ZeRO Overhead sharding
ZeRO-{1,2,3} are used to refer to sharding the optimizer states, gradients, and weights in this way, respectively.

How to make FSDP work

In[B_x, D] *$_D$ **W**in[D_X, F] *$_F$ **W**out[F, D_X] -> **Out**[B_x, D]

Forward pass calculations

To compute Loss[B_X]

1. $\mathbf{W}_{in}$[D, F] = AllGather($\mathbf{W}_{in}$[D_X, F])
   (not on critical path, can do it during previous layer)
2. Tmp[B_X, F] = In[B_X, D] *$_D$ W_{in}[D, F]
   (can free up $\boldsymbol{W}_{in}$[D, F] now - no longer needed)
3. $\mathbf{W}_{out}$[F, D] = AllGather($\mathbf{W}_{out}$[F, D_X])
   (not on critical path, can do it during previous layer)
4. Out[B_X, D] = Tmp[B_X, F] *$_F$ $\mathbf{W}_{out}$[F, D]
5. Loss[B_X] = …

How to make FSDP work

In[B_x, D] *$_D$ **W**in[D_X, F] *$_F$ **W**out[F, D_X] -> **Out**[B_x, D]

Forward pass calculations

To compute Loss[B_X]

1. $\mathbf{W}_{in}$[D, F] = AllGather($\mathbf{W}_{in}$[D_X, F])
   (not on critical path, can do it during previous layer)
2. Tmp[B_X, F] = In[B_X, D] *$_D$ W_{in}[D, F]
   (can free up $\boldsymbol{W}_{in}$[D, F] now - no longer needed)
3. $\mathbf{W}_{out}$[F, D] = AllGather($\mathbf{W}_{out}$[F, D_X])
   (not on critical path, can do it during previous layer)
4. Out[B_X, D] = Tmp[B_X, F] *$_F$ $\mathbf{W}_{out}$[F, D]
5. Loss[B_X] = …

Backward pass calculations

To compute d**W**out[F, D_X], d**W**in[D_X, F]

1. dOut[BX, D] = …
2. d$\mathbf{W}_{out}$[F, D] {U_X} = Tmp[B_X, F]^T *$_B$ dOut[B_X, D]
3. d$\mathbf{W}_{out}$[F, D_X] = ReduceScatter(d$\mathbf{W}_{out}$[F, D] {UX})
   (not on the critical path, can be done async)
4. $\mathbf{W}_{out}$[F, D] = AllGather($\mathbf{W}_{out}$[F, D_X])
   (can be done ahead of time)
5. dTmp[B_X, F] = dOut[B_X, D] *$_D$ $\mathbf{W}_{out}$[F, D]^T
   (can free up Wout[F, D] here)
6. d$\mathbf{W}_{in}$[D,F] {U_X} = In[B_X, D]^T *$_B$ dTmp[B_X, F]
7. d$\mathbf{W}_{in}$[D_X, F] = ReduceScatter(d$\mathbf{W}_{in}$[D, F] {U_X})
   (not on critical path, can be done async)
8. $\mathbf{W}_{in}$[D, F] = AllGather($\mathbf{W}_{in}$[D_X, F])
   (can be done ahead of time)
9. dIn[B_X, D] = dTmp[B_X, F] *$_F$ $\mathbf{W}_{in}$[D, F]^T
   (needed for previous layers, can free up $\boldsymbol{W}_{in}$[D, F] here)

FSDP performance

How to stay compute bound with FSDP?
I.e. how to avoid being communication bound? Let
C = chip flops; **W** = bidirectional network bandwidth; **X** = number of shards across which the batch is partitioned; **B** = batch size
T = sequence length; **L** = no of layers, **D** = embedding dimension, **F** = MLP hidden dimension

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

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

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

$$T \approx 4 \cdot D \cdot F \cdot \max\left(\frac{B}{X \cdot C}, \frac{1}{W_{\text{ici}}}\right)$$

Similar to data parallelism, compute bound when
(B * T) / (X * C) > 1 / W, or (B * T) / X > C / W

Examples

TPU ICI operational intensity
TPU v5p: **C** = 4.59×10^{14} FLOPs (~0.5PetaFLOPs)
W (ICI_BW) = 1.8e11 Bytes/sec
To stay compute bound, per-TPU **B*T > 2550**
(B*T is often assumed to simply be batch size)
Note: TPU v5p has three different ICI axes that could be used for FSDP. I.e.: **B*T > 2550 / n(axes used)**

DeepSeek V2: used a batch size of ~40M tokens
I.e.: one could use up to 40M/2550/3 = ~47k TPU v5p chips to train with FSDP on 3 axis. That's approximately 5 TPUv5 superpods.

LLaMA-3 70B: 6.3×10^{24} FLOPs, trained with batch size of 16M
Similar to above: No of TPU v5p chips: 16e6 / (2550 / 3) = 18,823
Assuming 50%MFU:
Training time = 6.3×10^{24} FLOPs / (18,823 * 4.59×10^{14} FLOP/s * .5) = ~1.46M seconds or ~17 days

Key takeaways

Pros

- FSDP allows us to scale to >10,000 chips before we hit bandwidth limits.
- Both FSDP and pure data parallelism become bandwidth bound when the batch size per device is lower than C/W

Cons

- While FSDP is more memory efficient than Data-Parallelism, larger models won't fit.

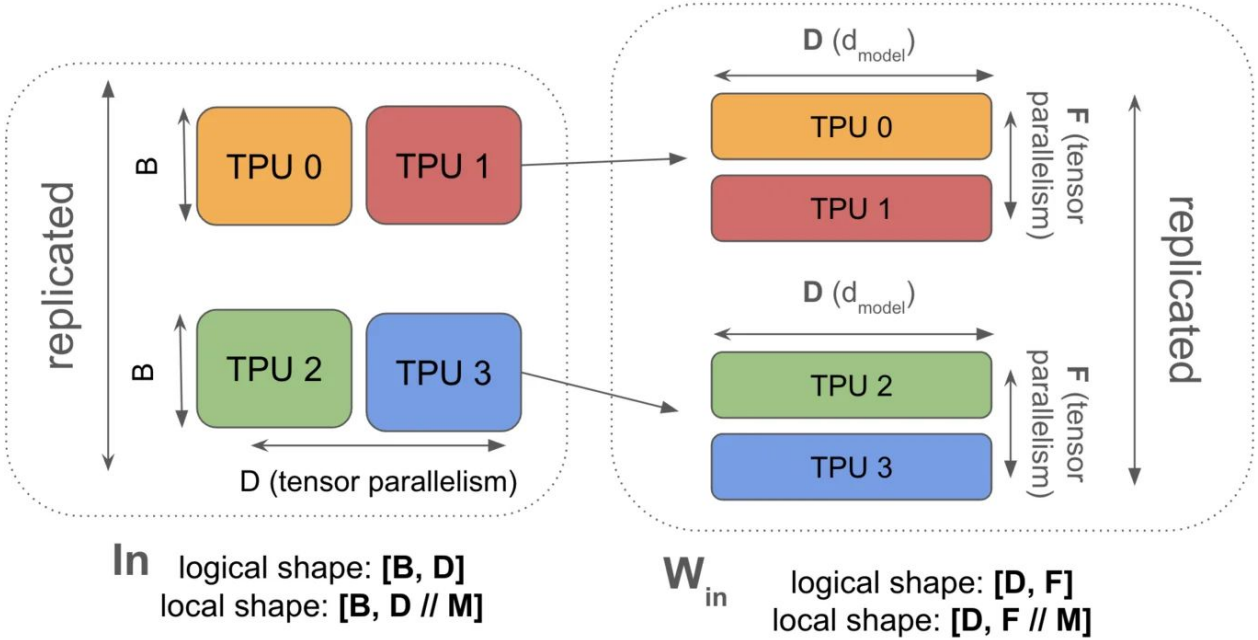
Next section: What if even this is not enough?

Tensor Parallelism

Tensor Parallelism

Aka: Megatron sharding or MP

Approach: Shard the feedforward dimension of the model and move the activations during the layer.
In[B, D_Y] *$_D$ **W**in[D, F_Y] *$_F$ **W**out[F_Y, D] -> **Out**[B, D_Y]
Note: we could also shard across X, but we use Y here to eventually combine with FSDP.



Staying compute bound

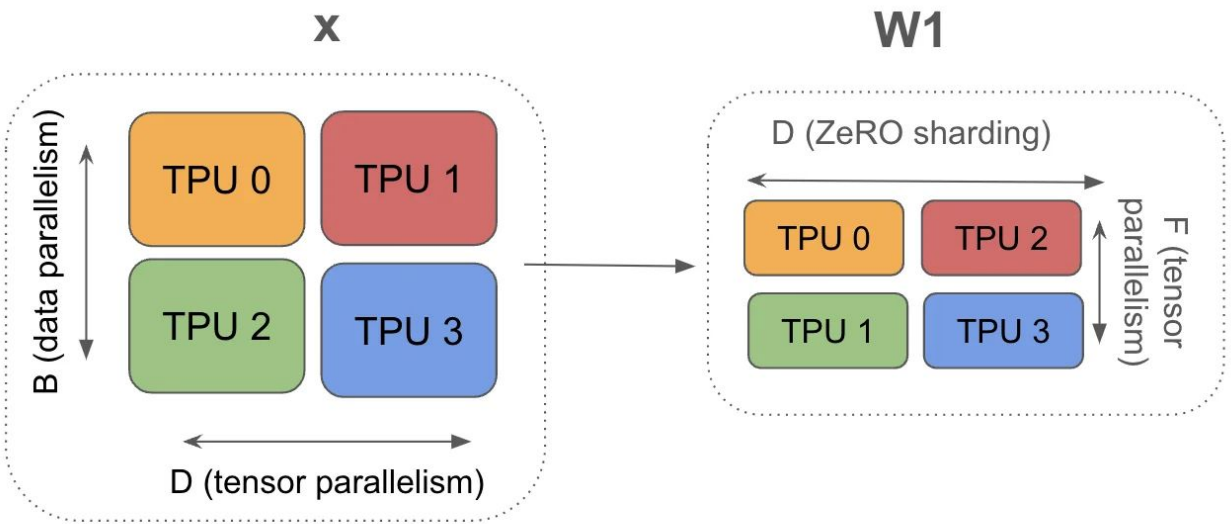
F > Y * C / W
For TPU v5p, C / W is 2550 (bf16)
$Y <= n_{used\_axes} * F / 2550$

which is typically 8 or 16.

Algorithm in the Scaling book
This can unlock a smaller efficient batch size per pod.

Combined FSDP and Tensor Parallelism

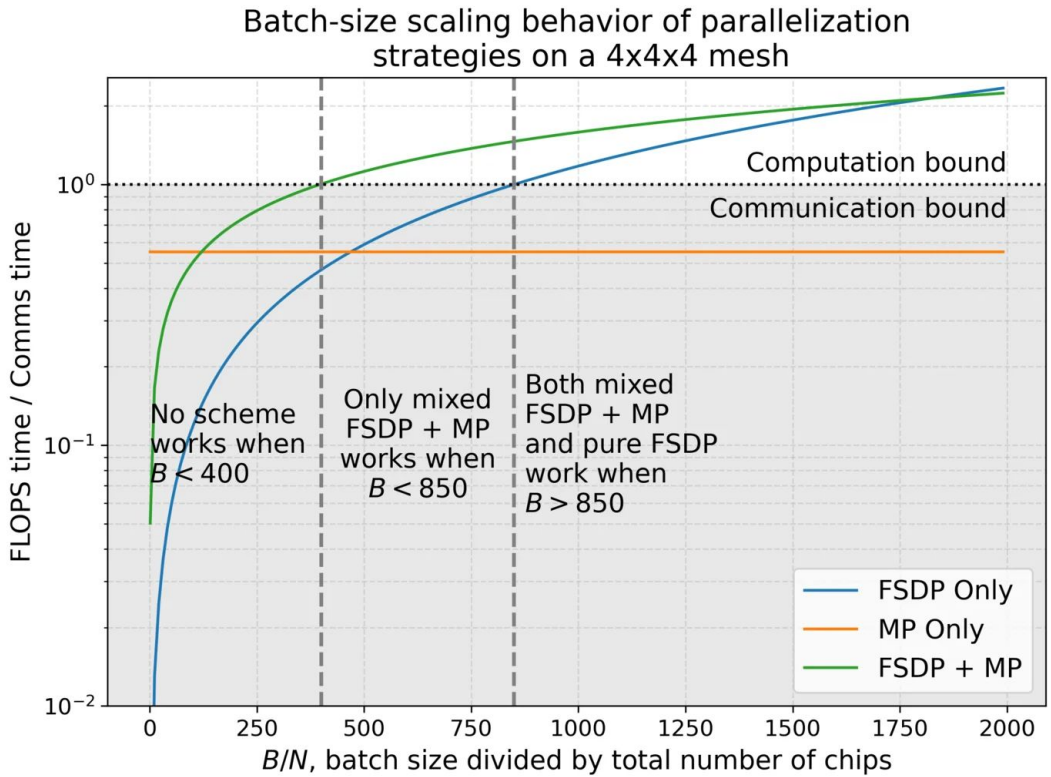
In[B_X, D_Y] *$_D$ **W**in[D_X, F_Y] *$_F$ **W**out[F_Y, D_X] -> **Out**[B_X, D_Y]



global shape: **[B, D]**
local shape: **[B // N, D // M]**

global shape: **[D, F]**
local shape: **[D // N, F // M]**

Combined FSDP and Tensor Parallelism



Ratio of FLOPs to comms time for optimal mixed FSDP/MP on a TPUv5p 4x4x4 slice with F=30k.

Summarizing all the approaches we've learned so far

Strategy	Description	
Data Parallelism	Activations are sharded along the batch dimension. Everything else is fully-replicated Need to apply AllReduce to gradients during the backward pass.	**B > X * C / W_ICI** TPU: v5f: B > 2550/n_axes
FSDP	Activations, weights, and optimizer state are sharded on the batch dimension. Need to gather weights just before they're used. Gradients are reduce-scattered.	**B > X * C / W_ICI** TPU: v5f: B > 2550/n_axes
Model Parallelism (aka Megatron, or Tensor Parallelism)	Activations are sharded along the d_model dimension, weights are sharded along the feedforward dimension, activations are gathered before Win, the result reduce-scattered after **W**out.	**$Y > n_{axes} F / 2550; F$**
Mixed FSDP + Model Parallelism	Both of the above, where FSDP gathers the model sharded weights.	

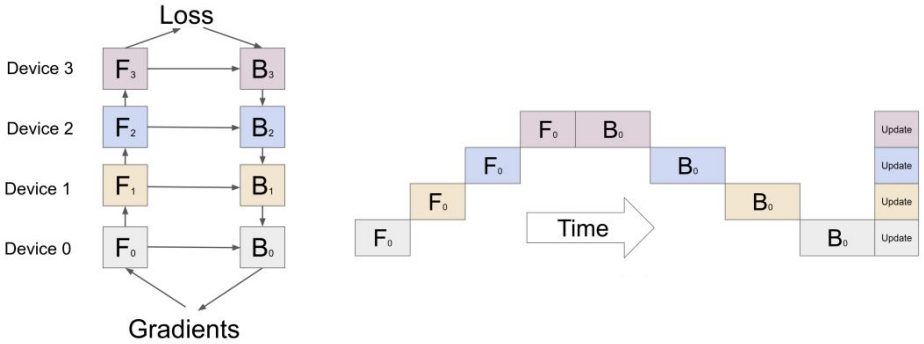
Table adopted from the Scaling Book

Pipeline parallelism

Pipeline parallelism

Approach: Split the model layers across multiple devices (pipeline stages) and pass activations between pipeline stages during the forward and backward pass.

1. Initialize data on Pipeline stage 0 with weights sharded across the layer dimension.
2. Perform the computations of the first layer on pipeline stage 0, then copy the resulting activations to stage 1, and repeat until you get to the last pipeline stage.
3. Compute the loss function and its derivative.
4. For the last pipeline stage, compute the gradients with respect to weights and activations and copy the activations gradients to the previous pipeline stage and repeat until you reach stage 0..



GPipe pipelined parallelism

Pros

- enables training of very large models
- low communication cost between pipeline stages (especially useful for GPUs)

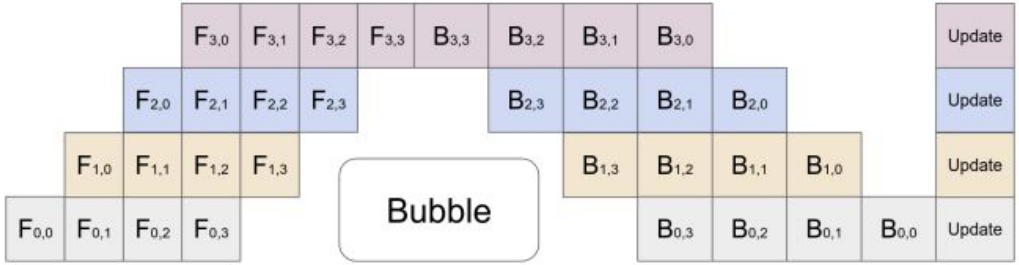
Cons

- Overheads from the **pipeline bubble**

Pipeline parallelism

Reducing the pipeline overheads/bubble

Approach 1: Increase microbatches



K - number of pipeline stages
M - microbatch size

Bubble size: $O(\frac{K-1}{M+K-1})$

Pipeline parallelism

Reducing the pipeline overheads/bubble

Approach 2: carefully schedule/overlap operations.

The DeepSeek v3 pipeline schedule (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, it avoids keeping GPUs/accelerators idle.

Take home exercise

Study: How to scale your model

Specifically, look into: Training LLaMA 3 on TPUs

That's it… we've covered:

Background and motivation

Scaling laws for transformers

A Systems View

Intro to accelerators for LLMs + some basics in computer architectures

Supercomputers

How do we scale compute for training LLMs?

Training transformers

Batch parallelism

Data parallelism

FSDP (Fully sharded data parallelism)

Tensor parallelism (aka Megatron)

Pipeline parallelism

Credit: Scaling book (aka: How to Scale Your Model)

Concluding remarks

Resources/links

Scaling book (again)

Risc V: https://projectf.io/posts/riscv-cheat-sheet/
GPU MMA details: https://arxiv.org/html/2402.13499v1 and here
H100 datasheet: https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet
IEE754 - https://en.wikipedia.org/wiki/IEEE_754
BF16 format: https://en.wikipedia.org/wiki/Bfloat16_floating-point_format
Roofline model: https://en.wikipedia.org/wiki/Roofline_model
Attention is all you need: https://arxiv.org/pdf/1706.03762
Optimization Techniques in Deep Learning: https://huggingface.co/blog/lsayoften/optimization-rush
Transformers made simple diagram here
Backprop explainer- https://xnought.github.io/backprop-explainer/
Tensor parallelism in Jax - link