INDIA-AI IMPACT SUMMIT 2026

MINISTRY OF ELECTRONICS & INFORMATION TECHNOLOGY GOVERNMENT OF INDIA

INDIAai

Workshop on Systems for LLMs

Systems for Large Language Models (LLMs) Workshop with Hands-On
Exploring the Infrastructure Behind Intelligent Systems

Sessions Overview:
LLM building blocks
GPUs for LLM
Inference serving with vLLM
Distributed Training

Tools: Hugging Face, PyTorch

Details
Date: 6th,7th September
Time: 9:30 am-5pm
Register @
https://forms.gle/a8rQQ835jJG42i3eA

Supported by: IBM-IITB Academic Research Partnership

IBM

CSE IIT Bombay

Distributed Training

Recap: The Train Loop

```
For e in epochs:
  For b in batches:
      loss = model.forward(b)
      loss.backward()

      optimizer.step()
```

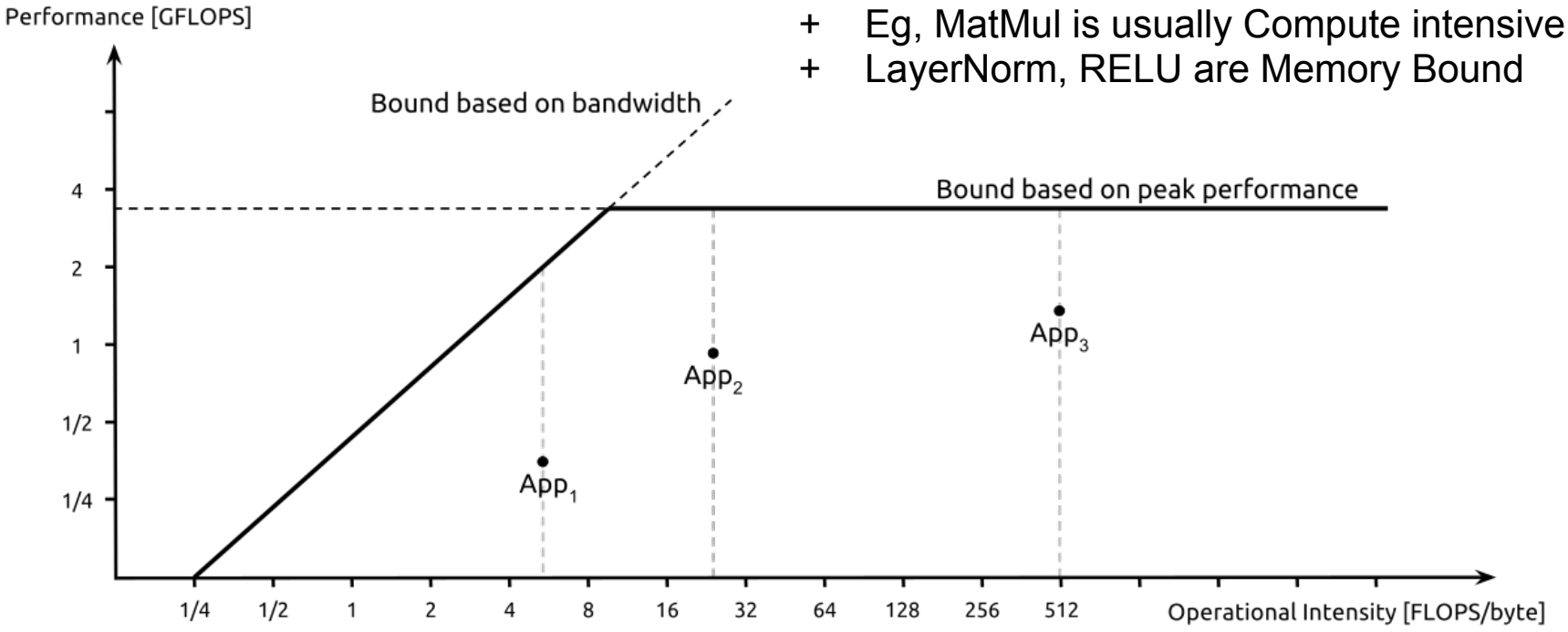
GPU Resources

- Compute
  - The number of SMs in the GPU, eg. 128 SMs on an A100
  - Capacity of the GPU, totally represented as TFLOPs
- Memory
  - HBM capacity: eg. 80 GB on an A100
  - To a lesser extent: Caches and per SM local memory capacity

Compute Bound vs Memory Bound

Each Op can be classified into Compute vs Memory intensive

+ Eg, MatMul is usually Compute intensive
+ LayerNorm, RELU are Memory Bound



Memory requirements for training

Simple envelope calculation: assuming 8B model in fp32 datatype with AdamW optimizer

Model weights: 8B * 4 bytes per param = ~30GB

Gradients: 4 bytes per param = ~30GB

Optimizer state: 12 bytes per param = ~90GB

\+ Activations = f(batch size, seq len)

Flow of memory

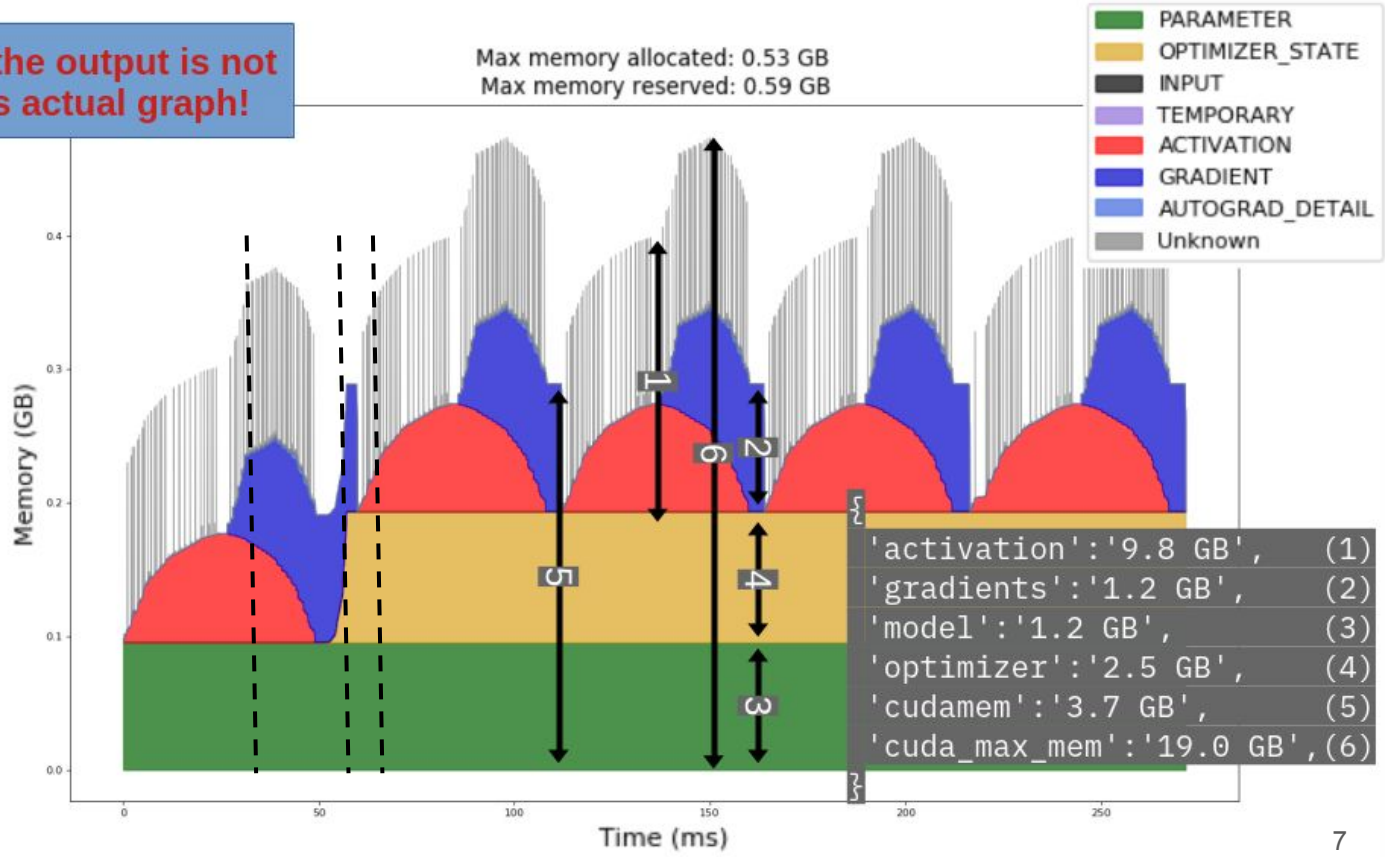
Model and Optimizer are flat.

Forward:
Activations increase

Backward:
Activations are freed,
gradients are created

Optimizer Step:
Gradients are freed.

Note: the output is not for this actual graph!



Memory requirements for training various model sizes

Model	Weights	Gradients	Optimizer	Sum (w/o activations)
Granite 2b	7.5 GB	7.5 GB	22 GB	37 GB
Llama 8b	30 GB	30 GB	90 GB	150 GB
Qwen 32B	120 GB	120 GB	360 GB	600 GB

https://huggingface.co/docs/accelerate/main/en/usage_guides/model_size_estimator

Memory Availability of commonly used GPUs

Vendor	Model	Memory
Nvidia	A100	80 GB
Nvidia	V100	32GB
Nvidia	A6000	23 GB
Nvidia	L40s	48 GB
AMD	Radeon RX 7900 XTX	24GB

Extras: Impact of Optimizer choice

Instead of using Adam, earlier models used to use SGD optimizer,

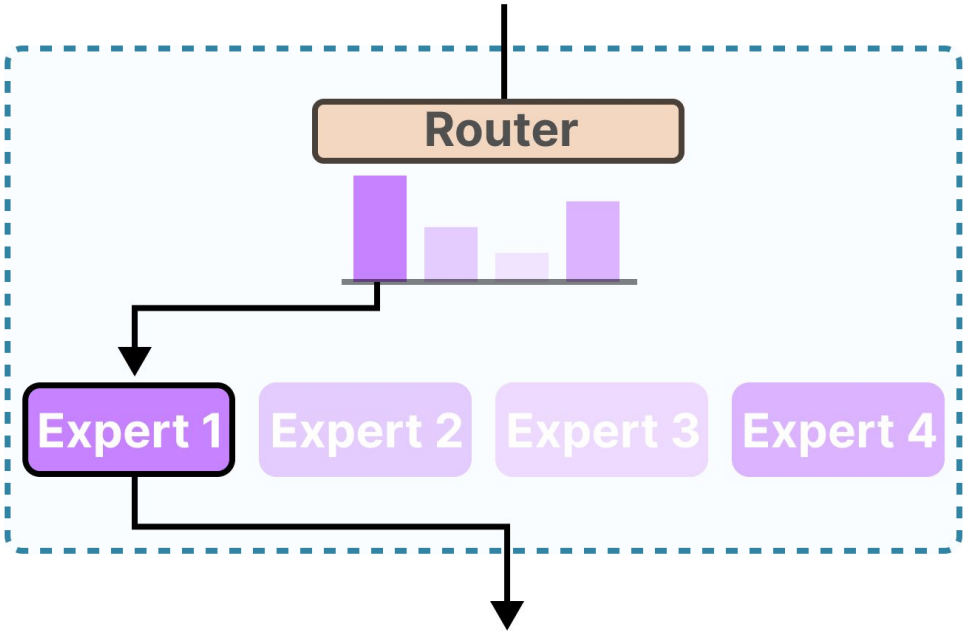
Which has: 8 bytes per param

But Adam/AdamW is better from an algorithmic point of view

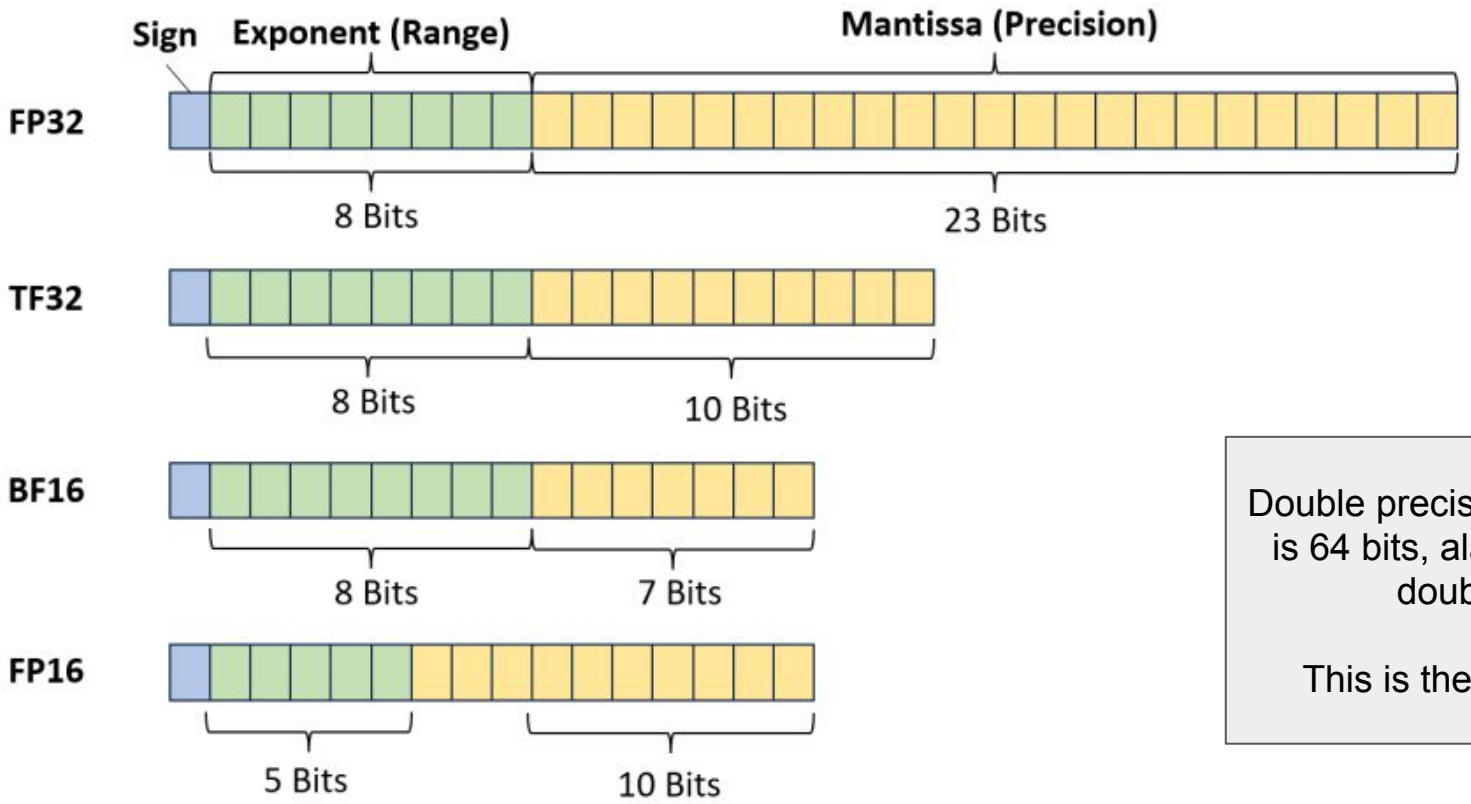
Extras: Impact of MoE

MoE architecture activates a fewer set of params during inferences, but the full model has to be loaded anyway

In case of training, it makes no difference



Other precision Types: half precision



Double precision typically is 64 bits, ala float and double.

This is the inverse.

Image Courtesy of: https://blog.eleuther.ai/transformer-math/

TF32

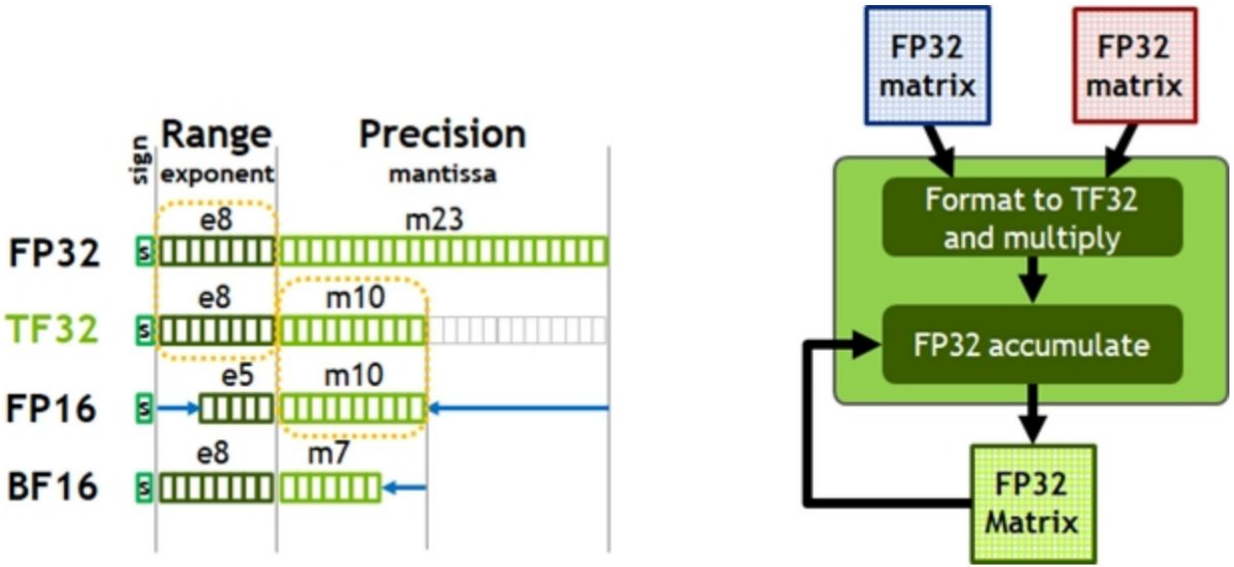


Figure 7. TensorFloat-32 (TF32) provides the range of FP32 with the precision of FP16 (left). A100 accelerates tensor math with TF32 while supporting FP32 input and output data (right), enabling easy integration into DL and HPC programs and automatic acceleration of DL frameworks.

Hardware support for other precisions

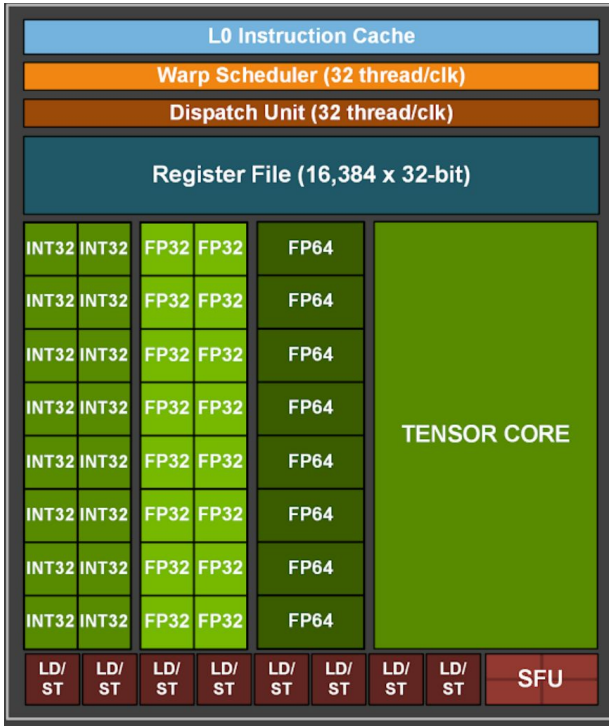
	A100 80GB PCIe	A100 80GB SXM
FP64	9.7 TFLOPS	
FP64 Tensor Core	19.5 TFLOPS	
FP32	19.5 TFLOPS	
Tensor Float 32 (TF32)	156 TFLOPS | 312 TFLOPS*	
BFLOAT16 Tensor Core	312 TFLOPS | 624 TFLOPS*	
FP16 Tensor Core	312 TFLOPS | 624 TFLOPS*	
INT8 Tensor Core	624 TOPS | 1248 TOPS*	
GPU Memory	80GB HBM2e	80GB HBM2e
GPU Memory Bandwidth	1,935 GB/s	2,039 GB/s

https://www.nvidia.com/en-us/data-center/a100/

H100 specs

Technical Specifications		
	H100 SXM	**H100 NVL**
FP64	34 teraFLOPS	30 teraFLOPS
FP64 Tensor Core	67 teraFLOPS	60 teraFLOPS
FP32	67 teraFLOPS	60 teraFLOPS
TF32 Tensor Core*	989 teraFLOPS	835 teraFLOPS
BFLOAT16 Tensor Core*	1,979 teraFLOPS	1,671 teraFLOPS
FP16 Tensor Core*	1,979 teraFLOPS	1,671 teraFLOPS
FP8 Tensor Core*	3,958 teraFLOPS	3,341 teraFLOPS
INT8 Tensor Core*	3,958 TOPS	3,341 TOPS
GPU Memory	80GB	94GB
GPU Memory Bandwidth	3.35TB/s	3.9TB/s

1 SM in the A100



Automatic Mixed Precision

Model weights: half - 16 bits

Gradients: half - 16 bits

Optimizer: full - 32 bits (4 bytes) * 12

Actually faster to run and takes lesser memory

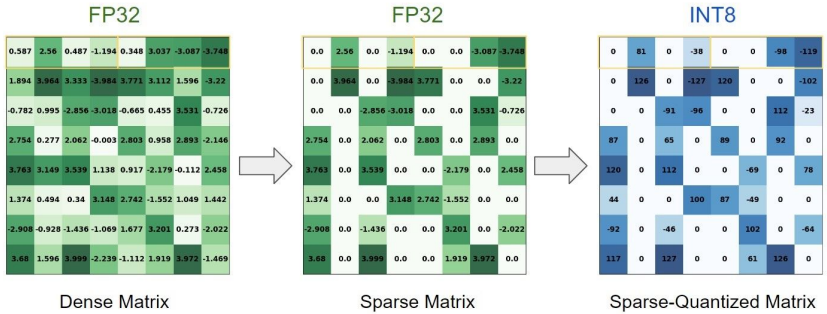
What more from here?

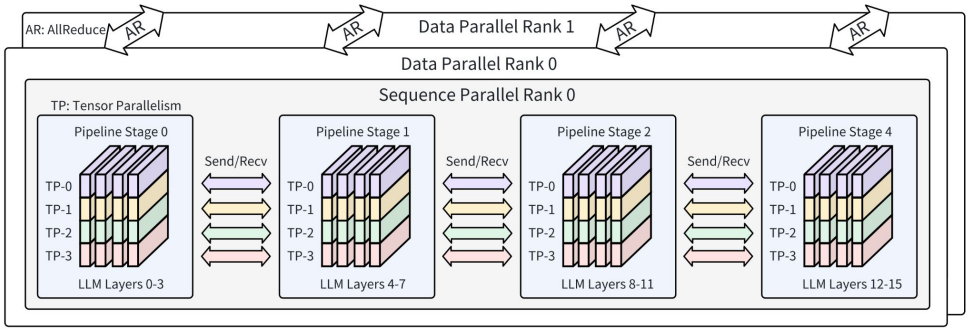
Fit model within this GPU

+ Quantization
+ Sparsity

Throw more resources at this problem

Distributed Training!





https://arxiv.org/pdf/2407.20018
https://developer.nvidia.com/blog/sparsity-in-int8-training-workflow-and-best-practices-for-tensorrt-acceleration/

Distributed Training Algorithms vs Systems

+ Data Parallelism (DDP)
+ Tensor Parallelism (TP)
+ Pipeline Parallelism (PP)

+ Full Sharded Data Parallelism (FSDP)
  + With variants - like Full/Hybrid Shard

+ 3D parallelism
+ Context/Sequence Parallelism
+ Expert Parallelism (EP)

+ Pytorch in-built DDP
+ Hugging Face Accelerate
+ Nvidia Megatron
+ Pytorch in-built FSDP
+ Microsoft DeepSpeed ZeRO family

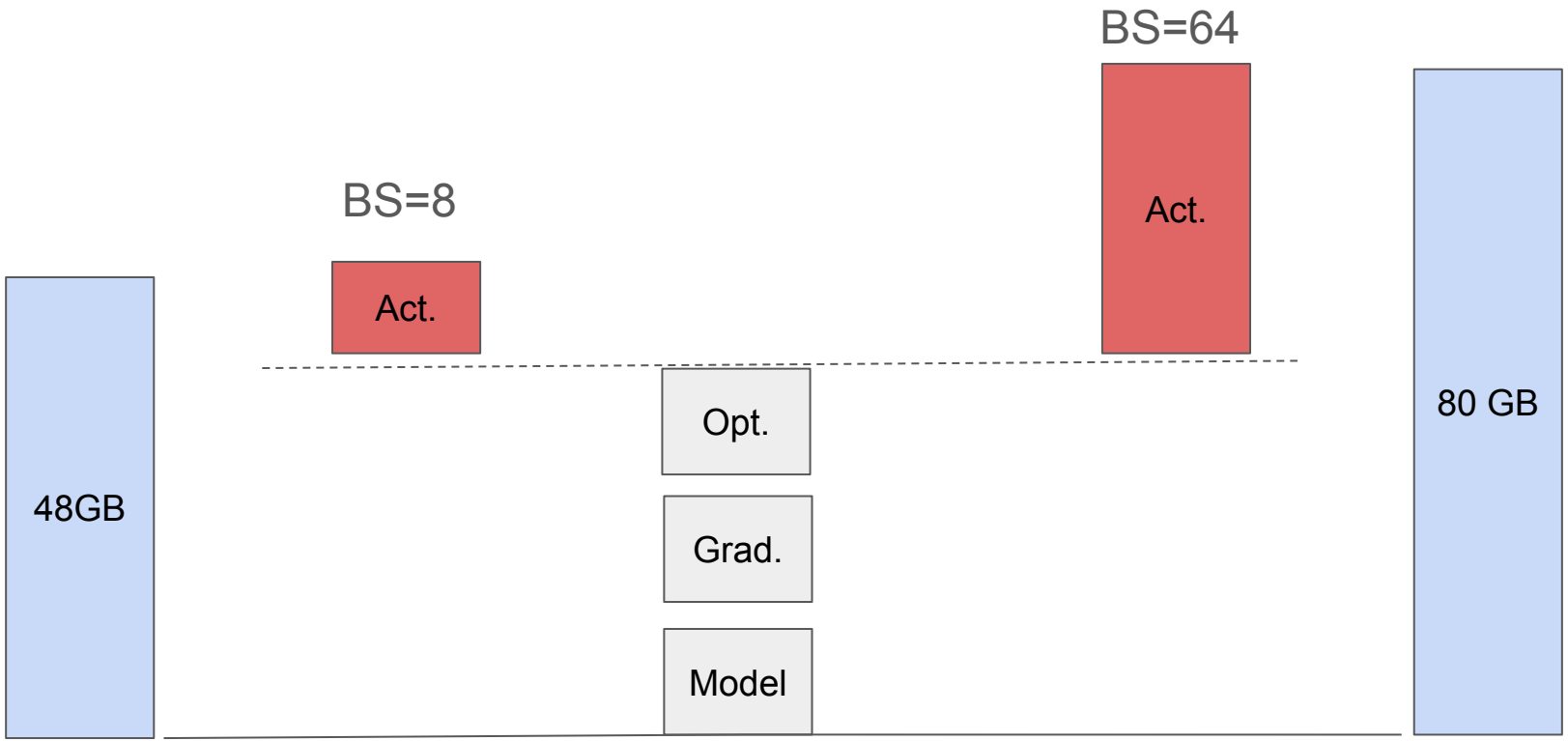
Basic Tech

Does your model fit into on a single GPU?

1. Yes - we go with DDP.
2. No - consider other options.

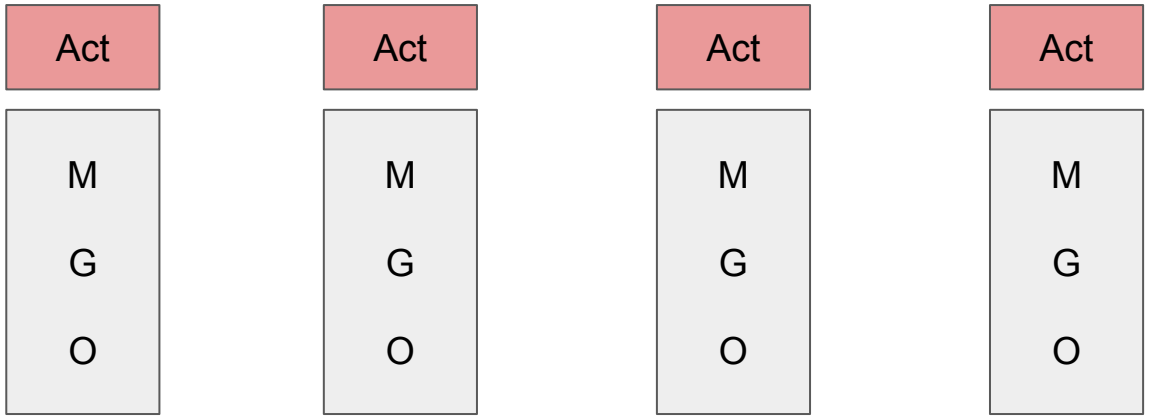
Here: model => (model + optimizer + gradients), the remaining goes to activations

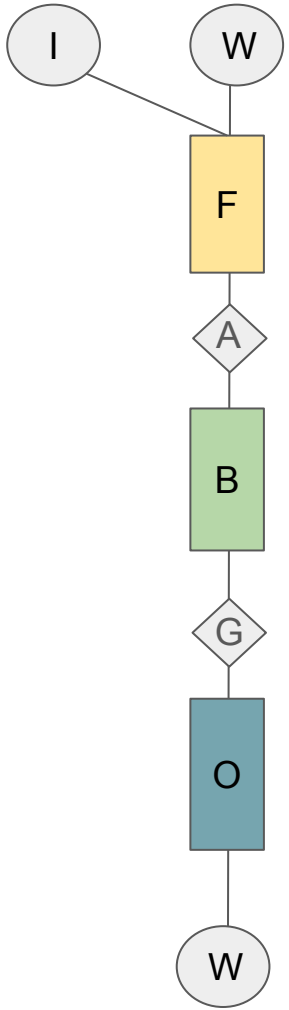
Limited Scope of Vertical Scaling



DDP: Data Parallelism

Can we have multiple gpus loading the same model and run training on different subsets of the data?

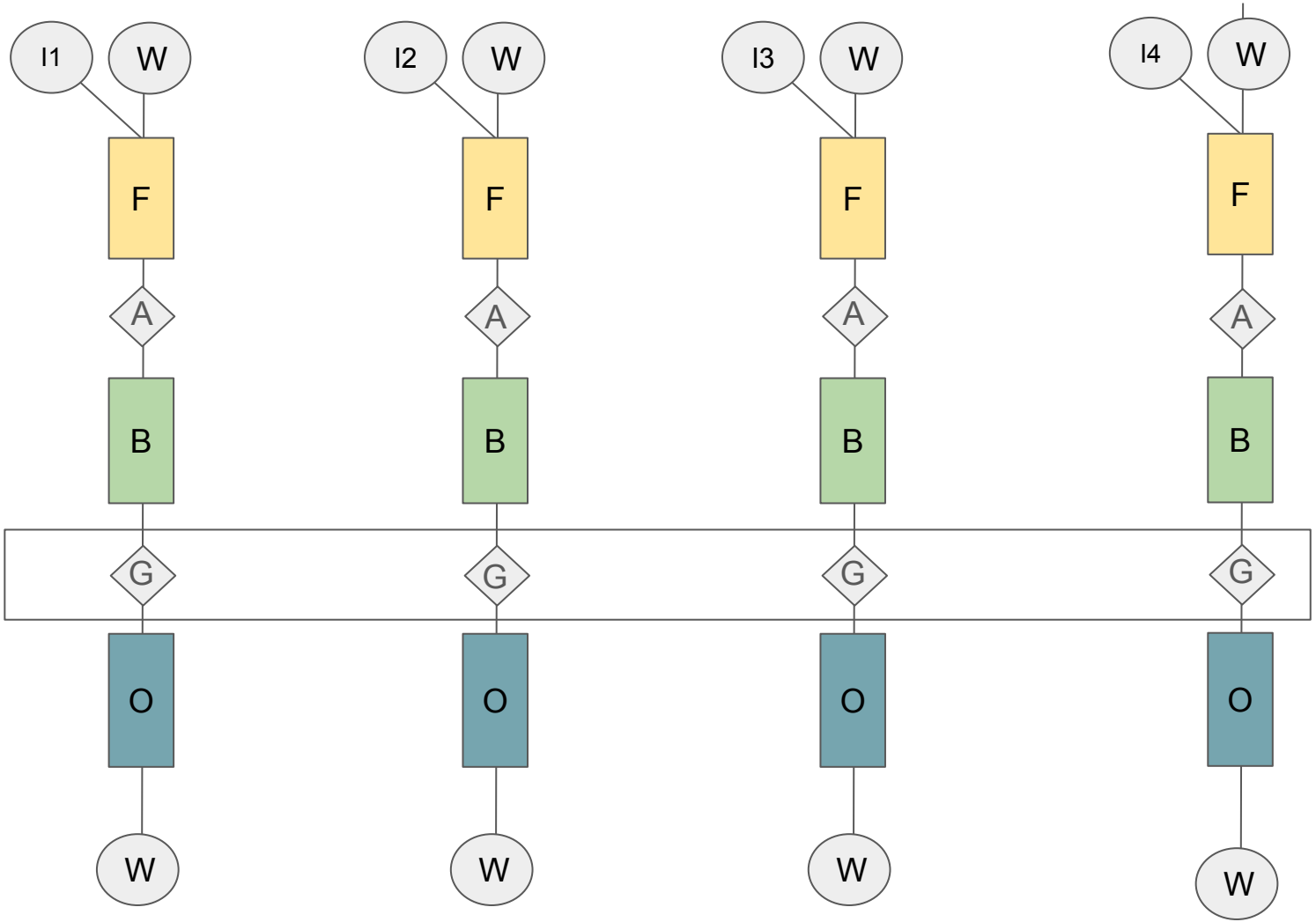



I
W
F
A
B
G
O
W

How to do simple Horizontal Scaling?

1. Divide your input data
2. Load identical training setup on each GPU - same model, optimizer setup
3. Run forward pass on all gpus:
    a. We will have intermediate activations created on each gpu
4. Run backward pass on all gpus:
    a. Now each process has its own gradients
5. Run optimizer on all gpus

This is a problem!

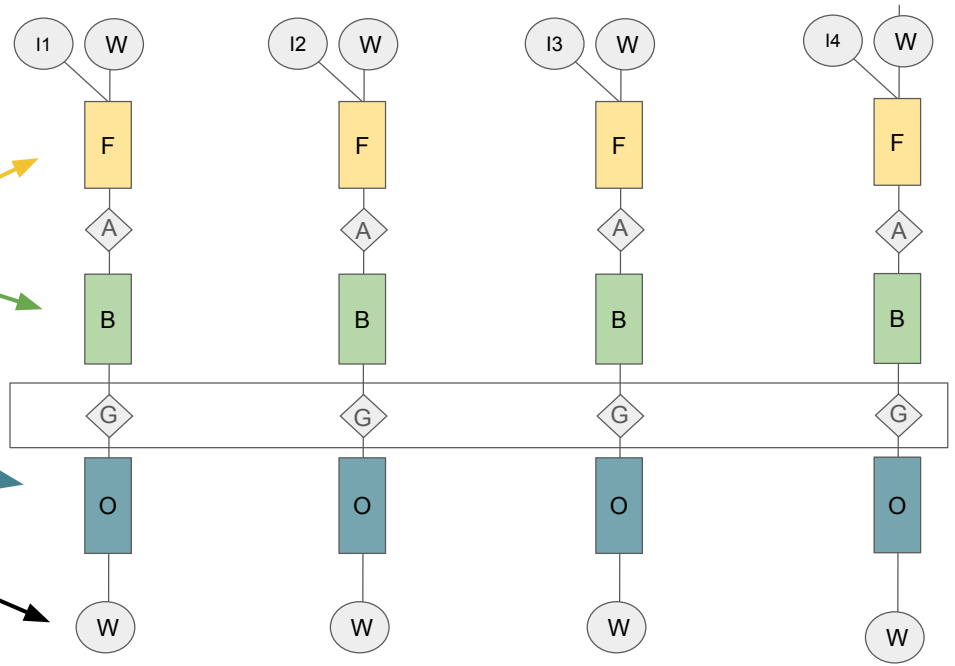

I1
W
I2
W
I3
W
I4
W
F
F
F
F
A
A
A
A
B
B
B
B
G
G
G
G
Synchronize
O
O
O
O
W
W
W
W

DDP Details

We have a single all-reduce of gradients after the backward pass

All processes:

1. Have their own independent forward & backward rounds.
2. But sync gradients: so identical optimizer rounds
3. End of step: all gpus have the same new updated model M'.
4. Next step start again.



What does “Synchronize” mean?

It means, we just add up the gradients.

Why does this work?

Gradients are always accumulate-able

Same mechanism is used during Gradient Accumulation.

How to synchronize gradients?

Each process has its own gradients: Num params * 2/4.

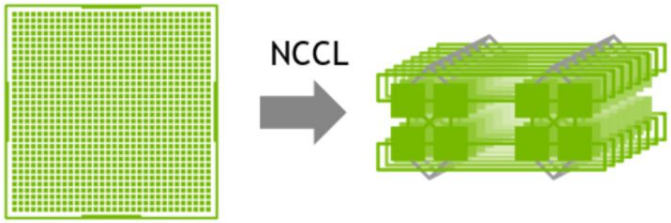
For a 8B model, this can mean 15-25 GB of data.

Gradients are automatically created by Pytorch during the backward pass - all of these “tensors” are in GPU memory.

The all-reduce NCCL primitive

https://developer.nvidia.com/nccl

Download NCCL | **Documentation** | **Developer Guide** | **GitHub** | **Watch GTC Webinar**



Performance

NCCL conveniently removes the need for developers to optimize their applications for specific machines. NCCL provides fast collectives over multiple GPUs both within and across nodes.

Ease of Programming

NCCL uses a simple C API, which can be easily accessed from a variety of programming languages.NCCL closely follows the popular collectives API defined by MPI (Message Passing Interface).

Compatibility

NCCL is compatible with virtually any multi-GPU parallelization model, such as: single-threaded, multi-threaded (using one thread per GPU) and multi-process (MPI combined with multi-threaded operation on GPUs).

CCL: honorable mentions

NCCL is not the only CCL.

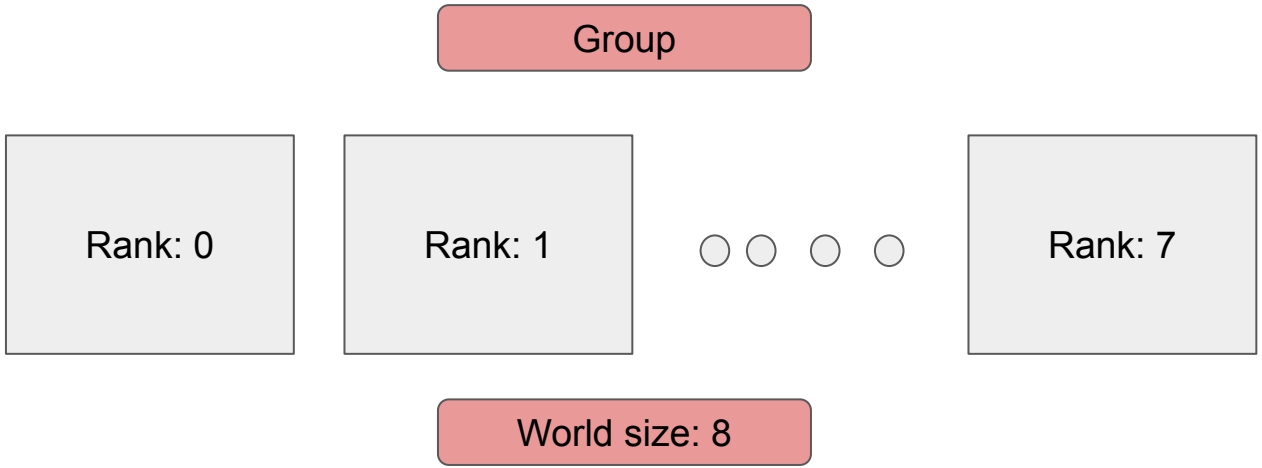
Communication between CPU cores (if using a CPU only training): Gloo

Other specialized hardware: Google TPU, or Intel XPU etc.

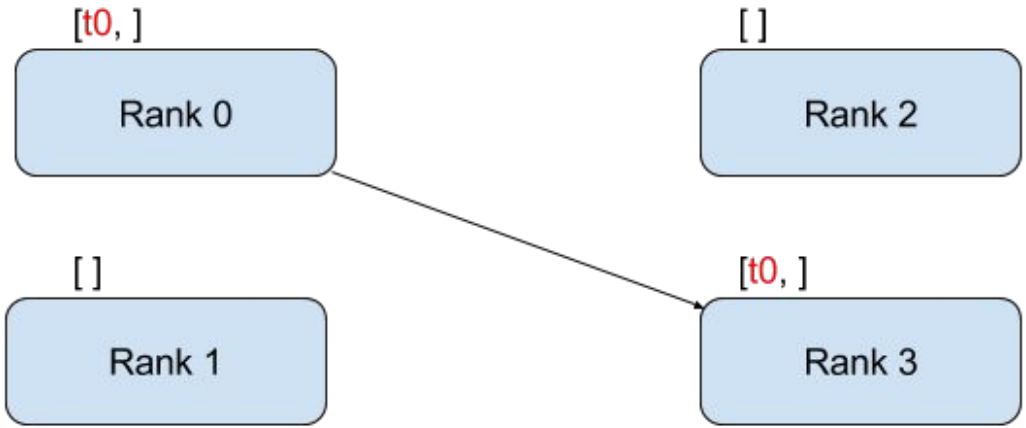
Some research work on optimizing CCLs: eg, UCCL from Berkeley

Collective setup - the higher level api

1. Each process can use the CCL functions as needed, using ranks to refer to others.
2. API exposed by NCCL to Pytorch to higher layers.
    a. At pytorch called `torch.distributed`. Abstracts the underlying CCL lib used.



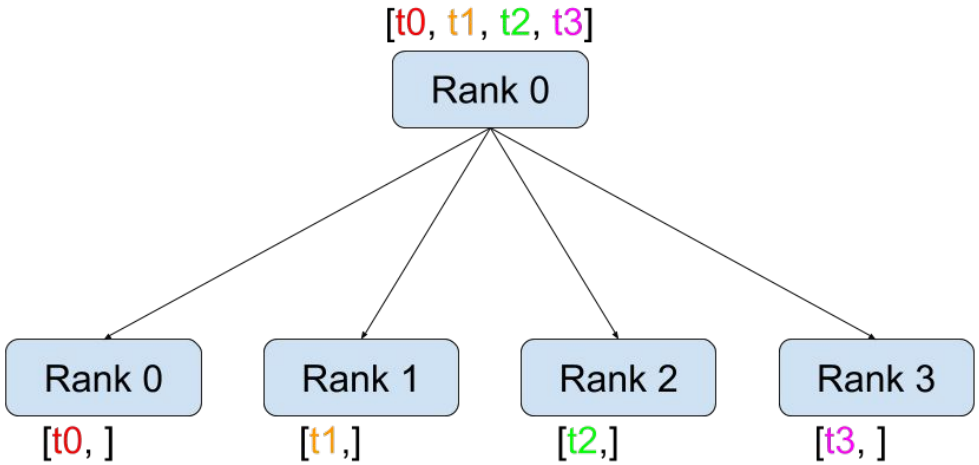
P2P communication



Point to Point Communication:
Send and recv

https://docs.pytorch.org/tutorials/intermediate/dist_tuto.html

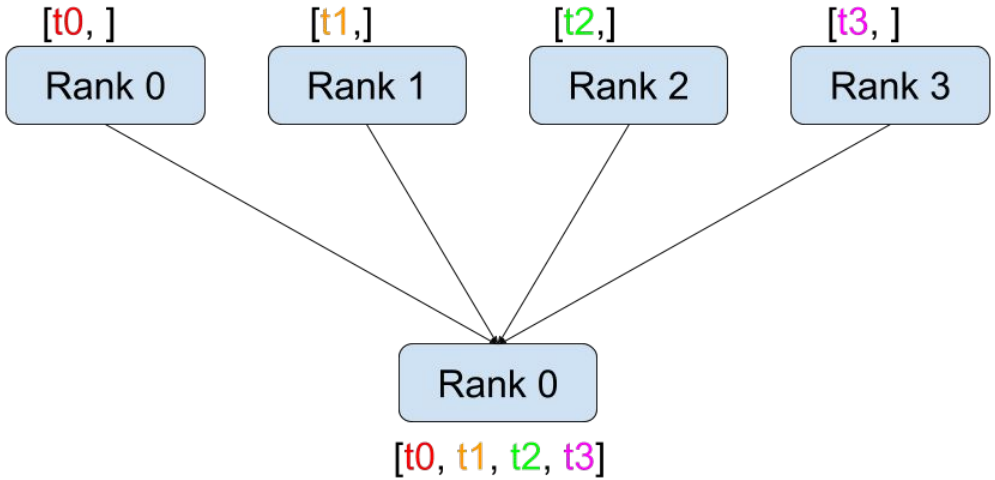
Collective Communication



Scatter

https://docs.pytorch.org/tutorials/intermediate/dist_tuto.html

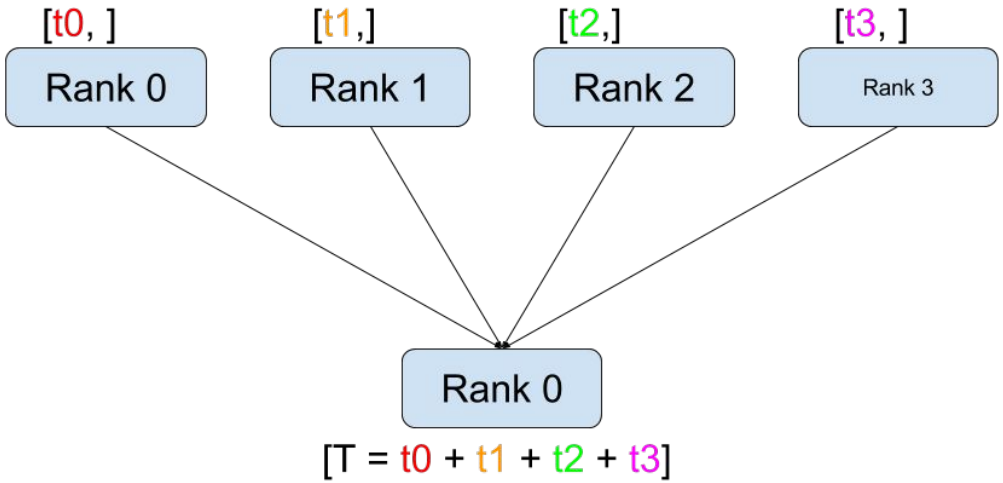
Collective Communication



Gather

https://docs.pytorch.org/tutorials/intermediate/dist_tuto.html

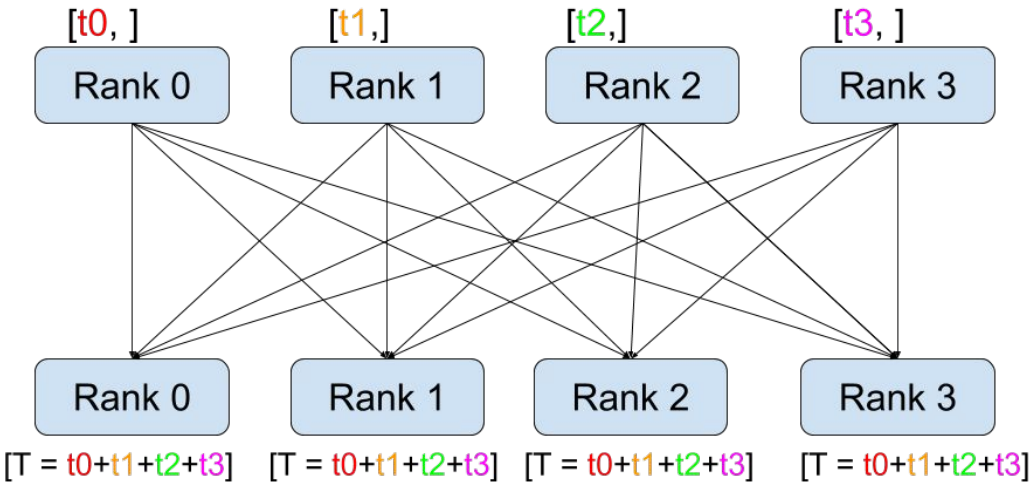
Collective Communication



Reduce

https://docs.pytorch.org/tutorials/intermediate/dist_tuto.html

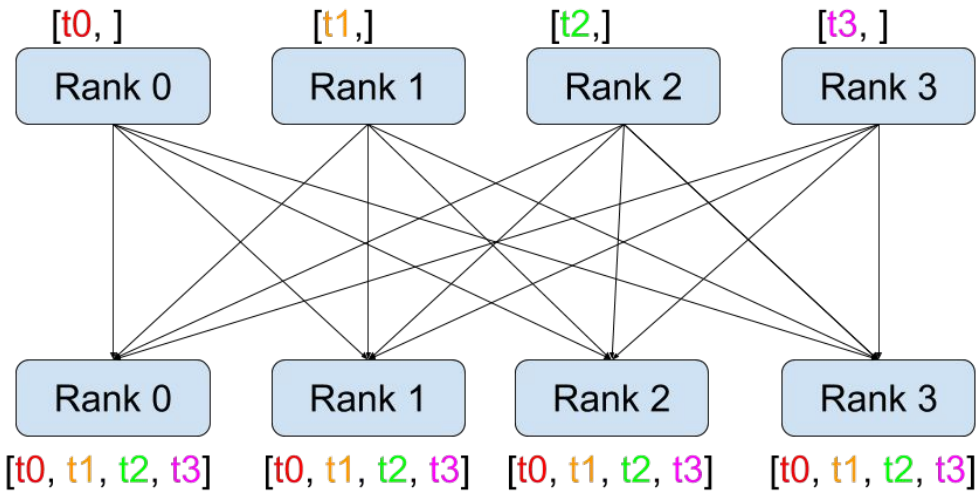
Collective Communication



All-Reduce

https://docs.pytorch.org/tutorials/intermediate/dist_tuto.html

Collective Communication



All-gather

https://docs.pytorch.org/tutorials/intermediate/dist_tuto.html

How does NCCL work?

1. Network paths:
    a. For GPUs on the same node: we have the NVLink.
    b. For GPUs across the node: we have NVSwitch and other higher level switches.
2. Topology management: ring vs tree etc
3. Chunking of tensors -> chunks -> packets for the lower level transport

https://images.nvidia.com/events/sc15/pdfs/NCCL-Woolley.pdf

Hands-on
(CCL primitives: in the toy framework)

How to do a simple hands-on?

We will use a simple Python emulator where:

1. Each “gpu” process is run as a thread.
2. CCL is completely simplified into a single global object with thread IPC

All-reduce test

Core libraries

All programs must be implemented as sub-classes of the core.Runner

```
import core
import ccl

class AllReduceExample(core.Runner):
    def run(self):
        print("Starting Runner: ", self.rank)
        res = self.ccl.all_reduce(self.rank, ccl.OP_REDUCE.SUM)
        print(f"Result on {self.rank} is: {res}")

core.run_world(4, AllReduceExample)
```

Run 4 threads with our functionality

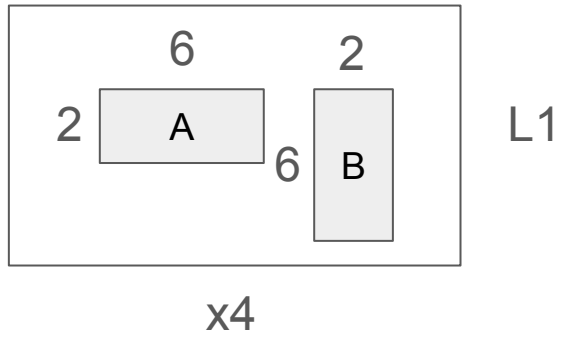
Test the setup

Add random sleep to each process.

Write a program where all even numbered ranks send a random number to the next process, which will print it out.

The Simplified Model (single.py)

```
import core

m = core.Model(4, 2, 6)
# create some samples for input
i, o = core.gen_inp(10, 2)

print("Output from simple single gpu run: ")
out = m.forward(i)
loss = out - o
m.backward(loss)

print("Output: ", out)
print("Loss: ", loss)
m.print_grads()
```



10 samples of size 2 each

Dummy outputs to manage loss gradients

No optimizer step

Look at the Model code in core.py

```
self.n = num_layers
self.in_dim = in_dim
self.hidden_dim = hidden_dim

self.blocks = []
self.inputs = []
self.grads = []
```

```
def forward(self, x):
    # Input is of the form: seqlen * in_dim

    for i in range(self.n):
        self.inputs[i]["a"] = x # save input
        x = x @ self.blocks[i]["a"]
        self.inputs[i]["b"] = x # save input
        x = x @ self.blocks[i]["b"]

    return x
```

The backward pass for simple matmul

$$X = \begin{pmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{pmatrix} \qquad W = \begin{pmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{pmatrix} \tag{1}$$

$$Y = XW \tag{2}$$

$$= \begin{pmatrix} x_{1,1}w_{1,1} + x_{1,2}w_{2,1} & x_{1,1}w_{1,2} + x_{1,2}w_{2,2} & x_{1,1}w_{1,3} + x_{1,2}w_{2,3} \\ x_{2,1}w_{1,1} + x_{2,2}w_{2,1} & x_{2,1}w_{1,2} + x_{2,2}w_{2,2} & x_{2,1}w_{1,3} + x_{2,2}w_{2,3} \end{pmatrix} \tag{3}$$

https://cs231n.stanford.edu/handouts/linear-backprop.pdf

Since L is a scalar and Y is a matrix of shape $N \times M$, the gradient $\frac{\partial L}{\partial Y}$ will be a matrix with the same shape as Y, where each element of $\frac{\partial L}{\partial Y}$ gives the derivative of the loss L with respect to one element of Y:

$$\frac{\partial L}{\partial Y} = \begin{pmatrix} \frac{\partial L}{\partial y_{1,1}} & \frac{\partial L}{\partial y_{1,2}} & \frac{\partial L}{\partial y_{1,3}} \\ \frac{\partial L}{\partial y_{2,1}} & \frac{\partial L}{\partial y_{2,2}} & \frac{\partial L}{\partial y_{2,3}} \end{pmatrix} \tag{4}$$

By the chain rule, we know that:

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial X} \qquad \frac{\partial L}{\partial W} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial W} \tag{5}$$

The terms $\frac{\partial Y}{\partial X}$ and $\frac{\partial Y}{\partial W}$ in Equation 5 are *Jacobian matrices* containing the partial derivative of each element of Y with respect to each element of the inputs X and W.

$$\frac{\partial L}{\partial X} = \begin{pmatrix} \frac{\partial L}{\partial y_{1,1}} w_{1,1} + \frac{\partial L}{\partial y_{1,2}} w_{1,2} + \frac{\partial L}{\partial y_{1,3}} w_{1,3} & \frac{\partial L}{\partial y_{1,1}} w_{2,1} + \frac{\partial L}{\partial y_{1,2}} w_{2,2} + \frac{\partial L}{\partial y_{1,3}} w_{2,3} \\ \frac{\partial L}{\partial y_{2,1}} w_{1,1} + \frac{\partial L}{\partial y_{2,2}} w_{1,2} + \frac{\partial L}{\partial y_{2,3}} w_{1,3} & \frac{\partial L}{\partial y_{2,1}} w_{2,1} + \frac{\partial L}{\partial y_{2,2}} w_{2,2} + \frac{\partial L}{\partial y_{2,3}} w_{2,3} \end{pmatrix} \tag{22}$$

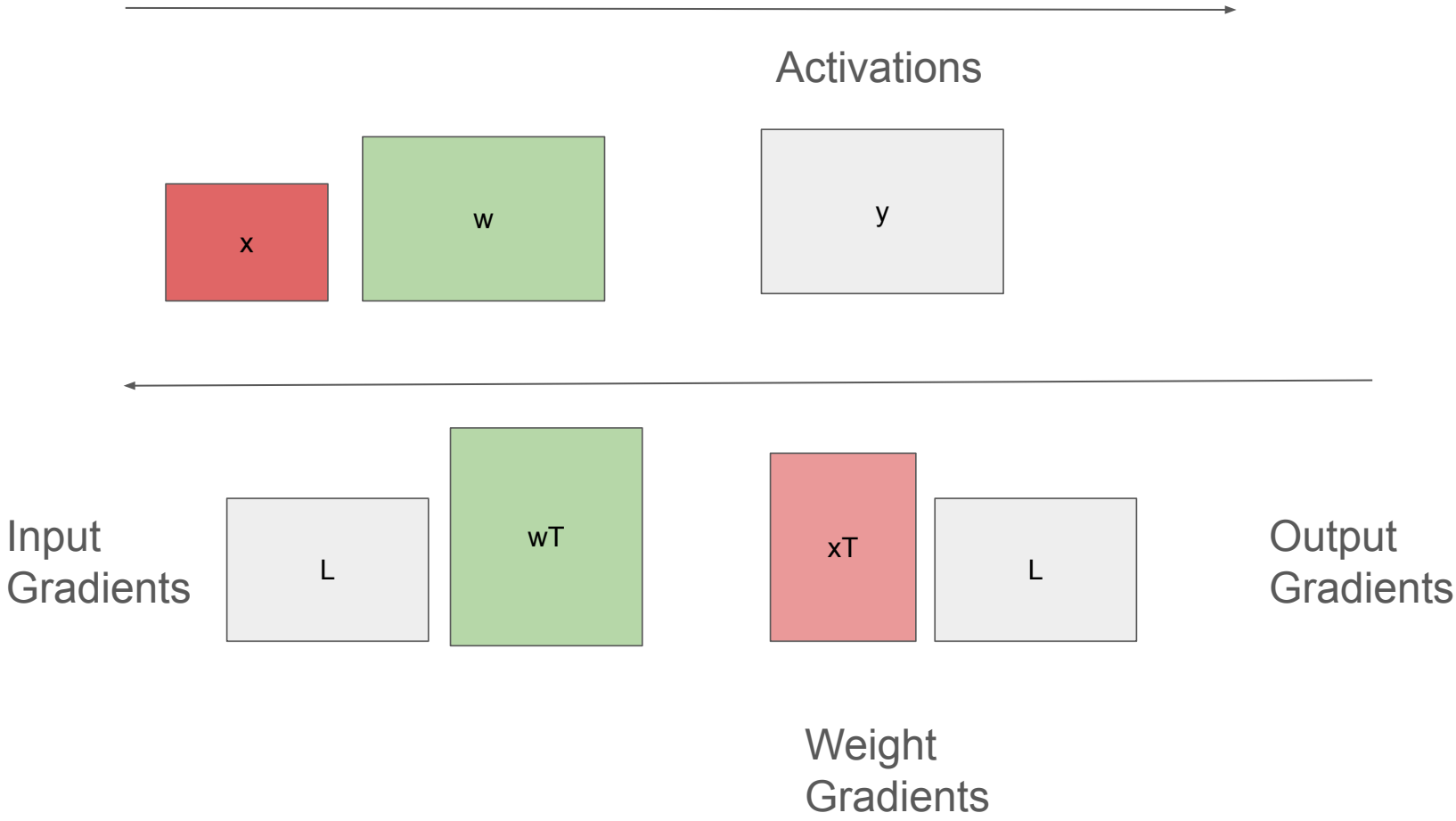
$$= \begin{pmatrix} \frac{\partial L}{\partial y_{1,1}} & \frac{\partial L}{\partial y_{1,2}} & \frac{\partial L}{\partial y_{1,3}} \\ \frac{\partial L}{\partial y_{2,1}} & \frac{\partial L}{\partial y_{2,2}} & \frac{\partial L}{\partial y_{2,3}} \end{pmatrix} \begin{pmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \\ w_{1,3} & w_{2,3} \end{pmatrix} \tag{23}$$

$$= \boxed{\frac{\partial L}{\partial Y} W^T} \tag{24}$$

Using the same strategy of thinking about components one at a time, you can derive a similarly simple equation to compute $\frac{\partial L}{\partial W}$ without explicitly forming the Jacobian $\frac{\partial Y}{\partial W}$:

$$\boxed{\frac{\partial L}{\partial W} = X^T \frac{\partial L}{\partial Y}} \tag{25}$$

```
def backward(self, loss_grad):
    # single var used to track grad propogation across layers
    # starts out with the input loss_grad
    act_grad = loss_grad

    for i in range(self.n-1, 0-1, -1):
        self.grads[i]["b"] = self.inputs[i]["b"].T @ act_grad
        act_grad = act_grad @ self.blocks[i]["b"].T

        self.grads[i]["a"] = self.inputs[i]["a"].T @ act_grad
        act_grad = act_grad @ self.blocks[i]["a"].T
```


Activations
x
w
y
Input Gradients
L
wT
xT
L
Output Gradients
Weight Gradients

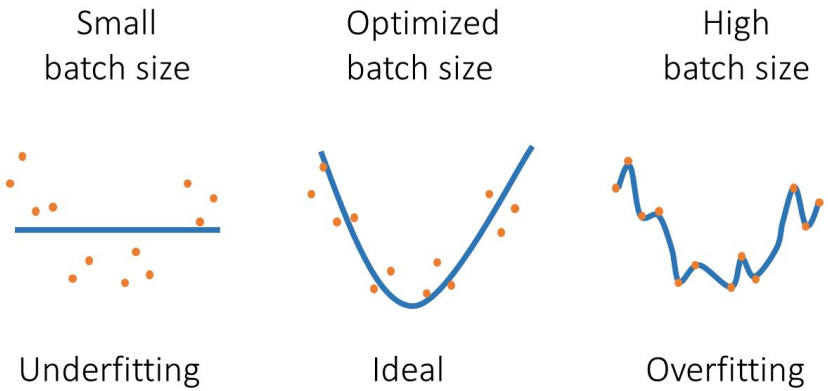
Implementing DDP

```
m = ...
i, o = ...
class DDPRunner(core.Runner):
    def run(self):
        # make a local copy of the model
        _m = copy.deepcopy(m)
        _i = ... # get input shard of relevance to us
        ... # implement forward pass here
        loss = out - _o
        ... # implement backward pass here
        ... # any final synchronization here
        if self.rank == 0:
            global m_final
            m_final = _m
core.run_world(2, DDPRunner)
# single model processing here
# ...
m.compare(m_final)
# This test should return true
```

Problems with DDP

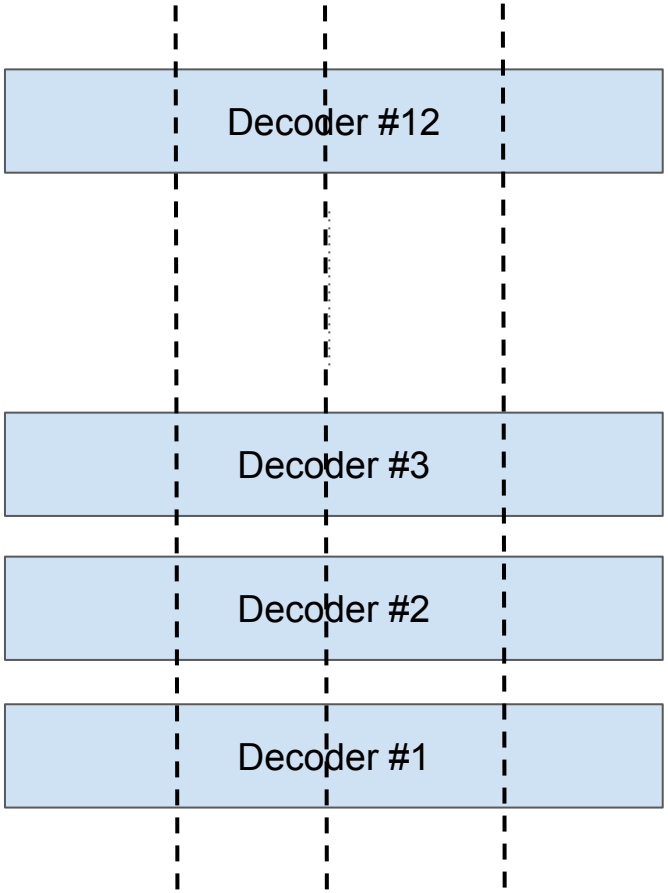
Can only scale by batch size.

Having too large of a batch size is bad for the DL learning

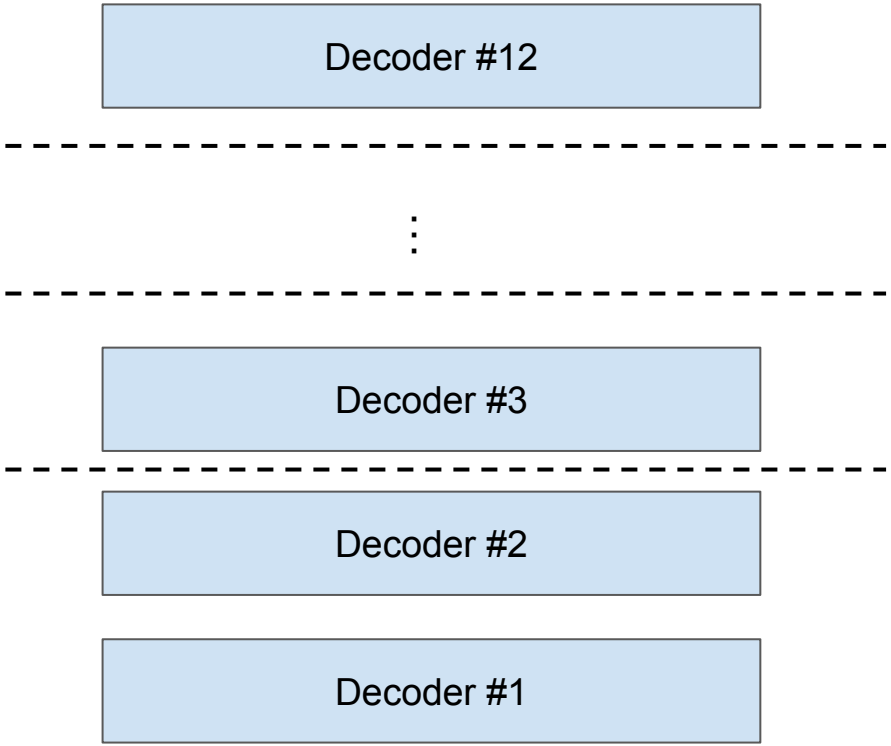


Doesn't work if your model is too big to fit on a single gpu

https://medium.com/thedeephub/optimizing-neural-network-training-understanding-batch-size-in-higher-dimensions-7687aa35cb6b



Split model vertically

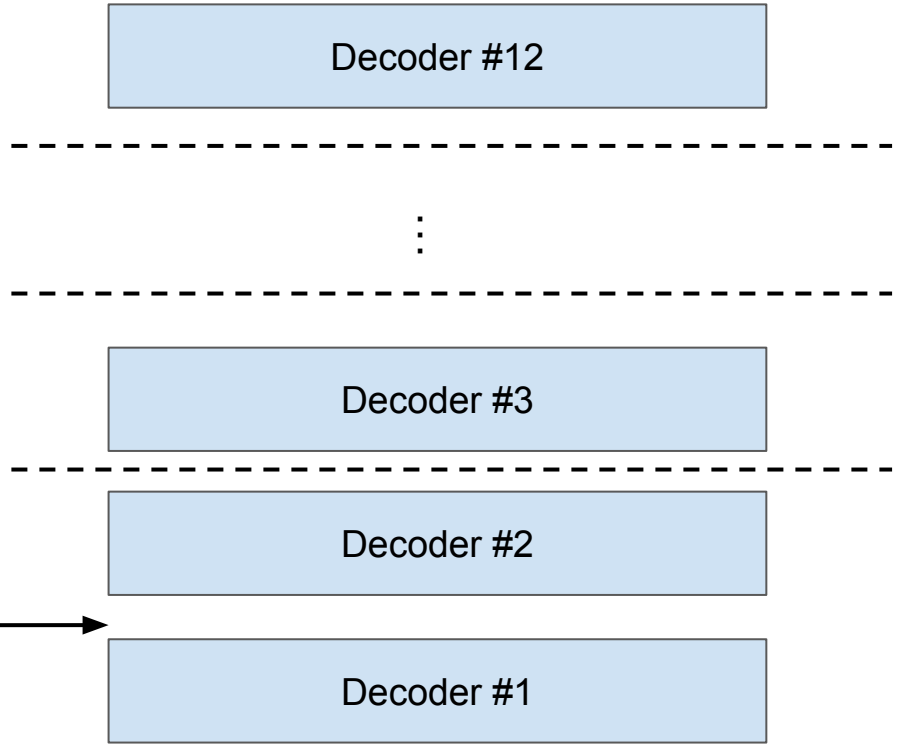


Split model horizontally

Pipeline Parallelism

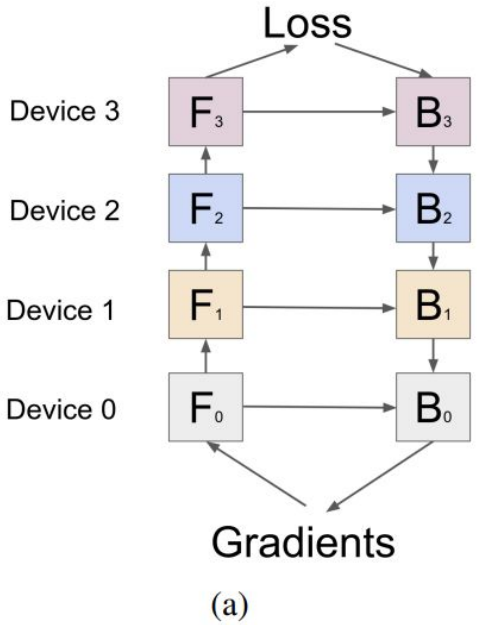
Each GPU has a single stage:
So memory is divided by world_size

A stage can have multiple layers/blocks

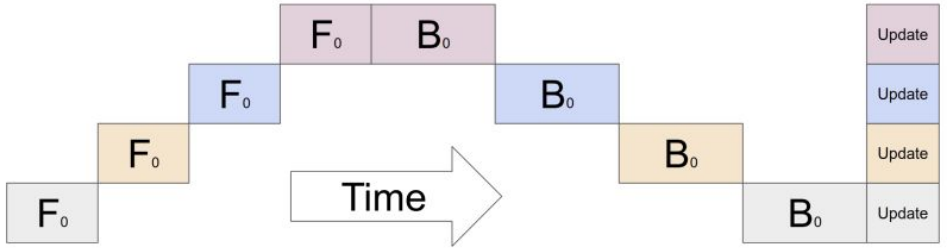


Split model horizontally

Pipeline Parallelism



(a)



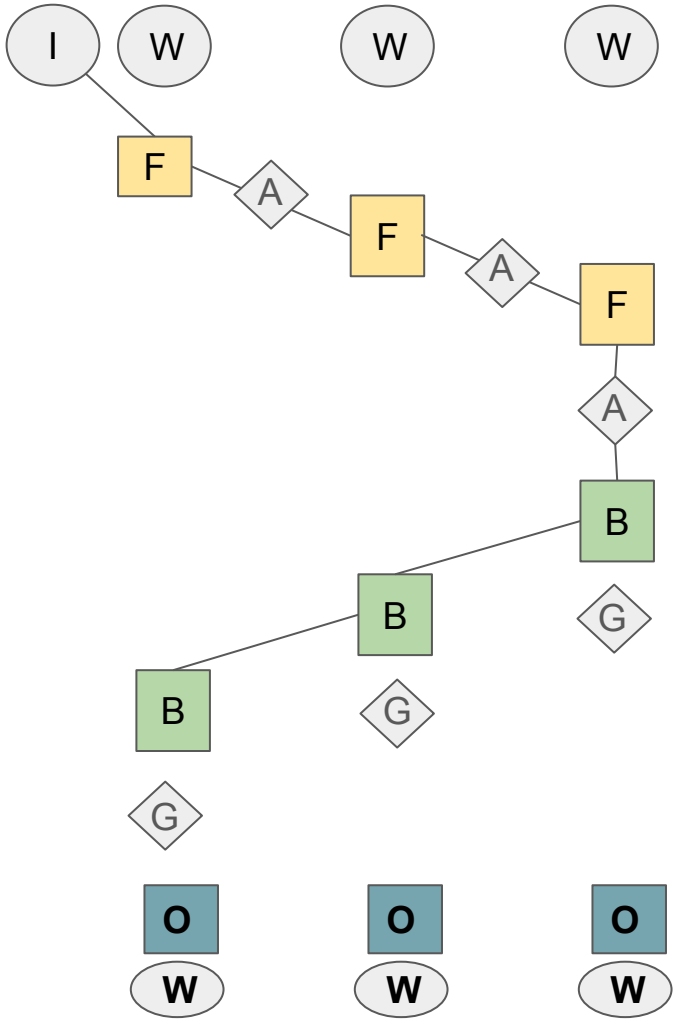
(b)

PP over 4 devices
Why is the block for B around double the size of F?

https://gixtools.net/2023/11/mastering-llm-techniques-inference-optimization/

What is needed?

1. Split up the model - loading only needed layers on each process
2. Within stages:
    a. Run forward/backward normally
3. At stage boundaries:
    a. Use p2p send/recv CCL calls
4. At end of forward: flow in reverse direction for backward pass
5. Each optimizer is only focused on it’s section of the model


I
W
W
W
F
A
F
A
F
A
B
B
G
B
G
G
O
O
O
W
W
W

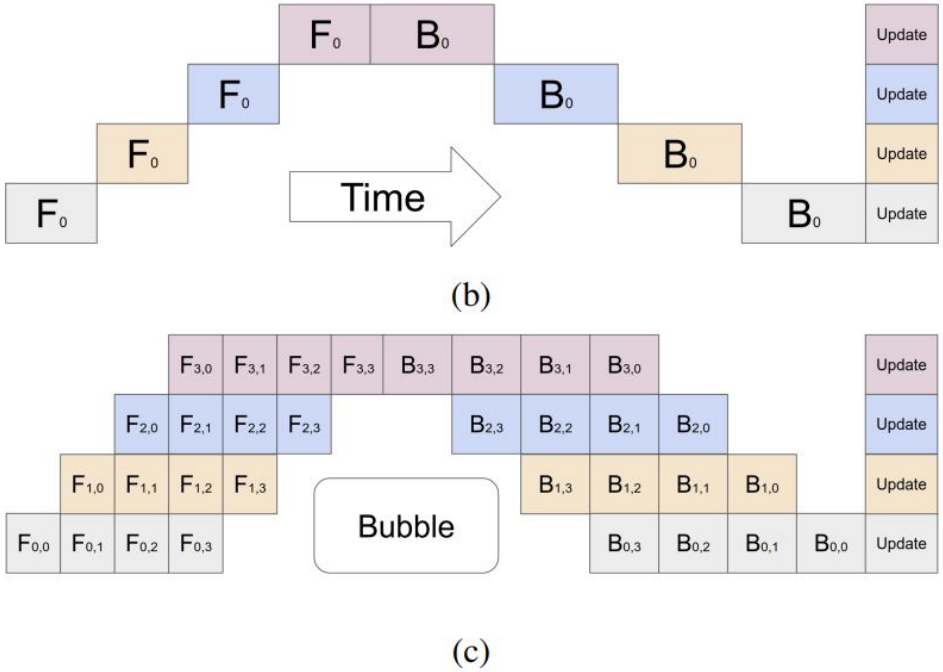
```
m = ...
i, o = ...
class PPRunner(core.Runner):
    def run(self):
        # make a local copy of the model
        _m = copy.deepcopy(m)
        ... # identify what subset of the model we will use for this rank
        ... # implement forward pass depending on our rank,
        # including any sync before and after
        loss = out - _o # only on the final rank
        ... # implement backward pass here, depending on rank
        ... # any final synchronization here
        # purely test code
        global m_final
        ... # copy gradients of our portion into a single copy
core.run_world(4, DDPRunner)
# single model processing here
# ...
m.compare(m_final)
# This test should return true
```

Problem with PP

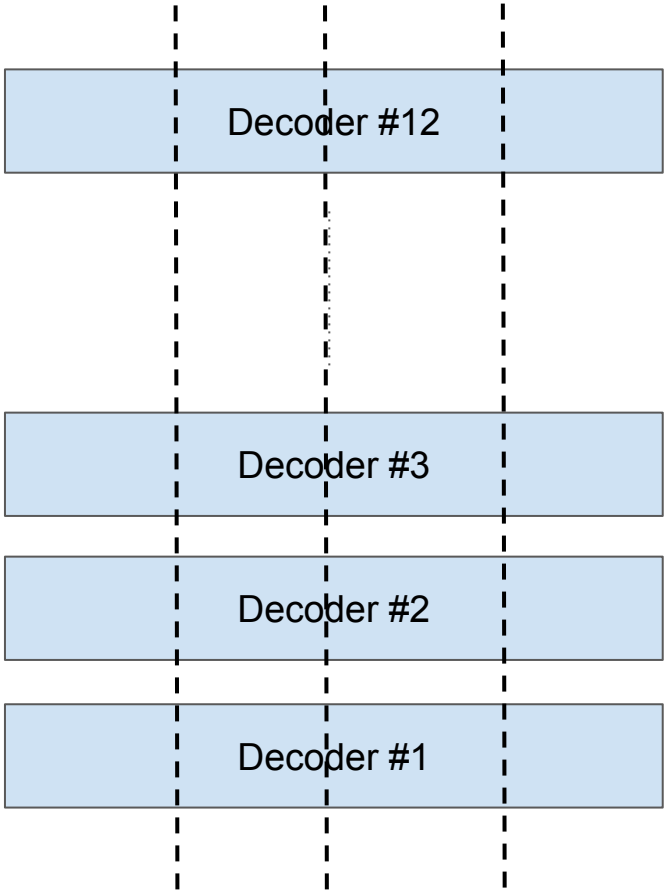
Bubbles

Lot of prior art on bubble management:

1. Mini/micro batches
2. Reuse of existing bubbles for secon
3. Alternative schedules
4. Fusion (of pipelines)



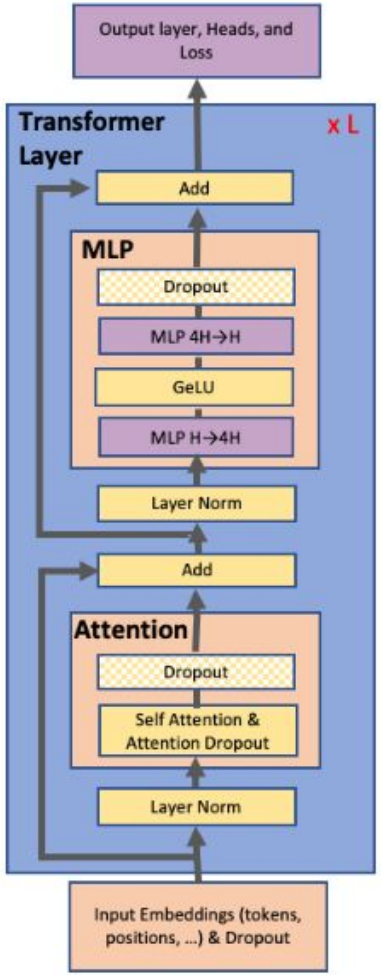
Can we split up the model in a way that does not lead to bubbles?



Split model vertically

What is the challenge with this?

There is **compute balance** between the processes, but how does this process now look like?

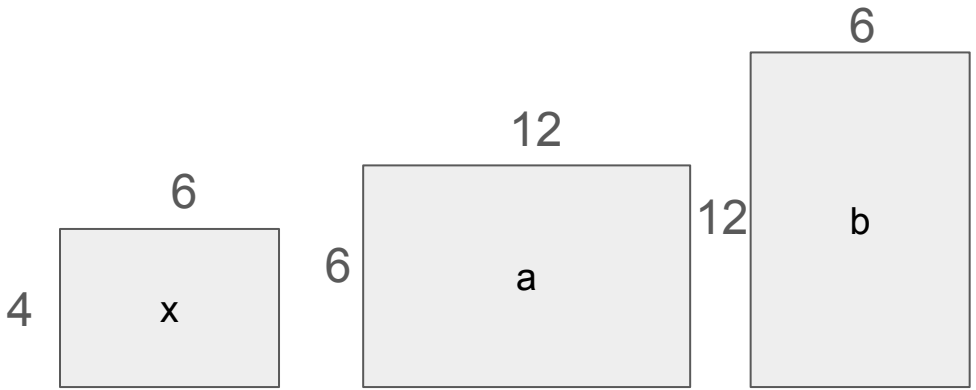


Let's focus on a single MLP block in 1 Decoder layer:

Linear layers A and B, say input is X

X = XA
X = GELU(X)
X = XB
X = DROPOUT(X)

```
import numpy as np

np.random.seed(10)
x = np.random.rand(4, 6)
a = np.random.rand(6, 12)
b = np.random.rand(12, 6)

x = x @ a
```

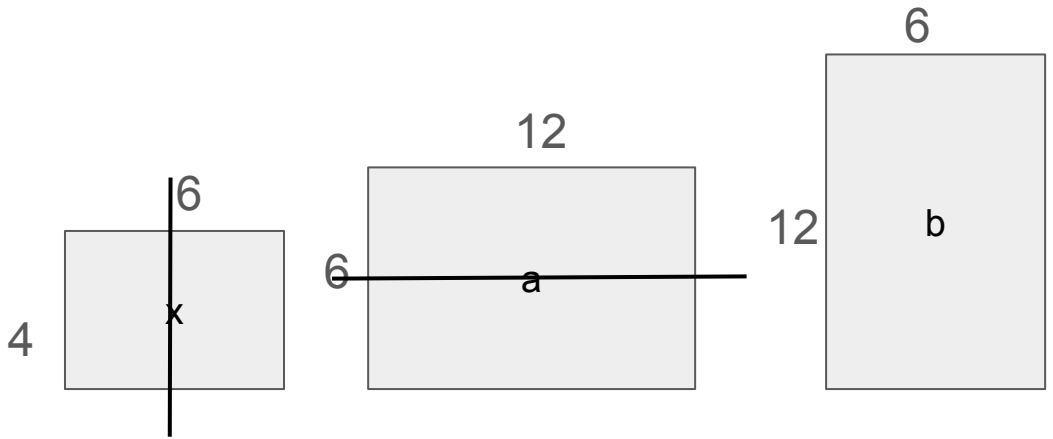



```
a

a[0:2] -> along rows
a[:, 0:2] -> along
columns

# check and verify the
shapes
```

Let’s try one option

One option to parallelize the GEMM is to split the weight matrix A along its rows and input X along its columns as:

$$X = [X_1, X_2],\ A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}. \qquad (2)$$



Solution

```
x @ a

x1 = x[:, 0:3]
x2 = x[:, 3:6]

a1 = a[0:3]
a2 = a[3:6]

(x1@a1) + (x2@a2)
```

How to continue the chain of computation?

X = XA
X = GELU(X)
X = XB
X = DROPOUT(X)

Let’s simplify for the moment and work with:
Y = XA
Z = YB

Y1 = X1A1, Y2 = X2A2

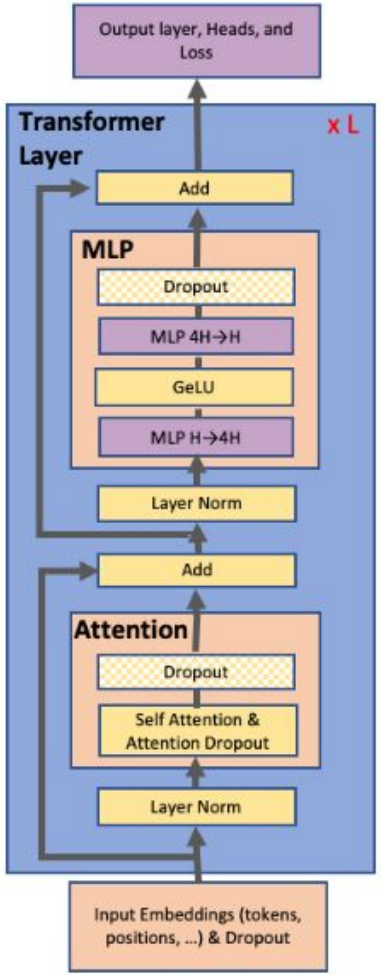
How will you partition B?

How will you arrange the computation so that the 2 processes can proceed independently?

Solution

```
x @ a

x1 = x[:, 0:3]
x2 = x[:, 3:6]

a1 = a[0:3]
a2 = a[3:6]

y1 = x1@a1
y2 = x2@a2
```

```
Keep b as it-is

z1 = y1 @ b
z2 = y2 @ b

z == z1 + z2
```



Let's focus on a single MLP block in 1 Decoder layer:

Linear laters A and B, say input is X

X = XA
X = GELU(X)
X = XB
X = DROPOUT(X)

The problem

So, we can safely synchronize once at the end of the MLP block.

But, this doesn't work because of GELU:

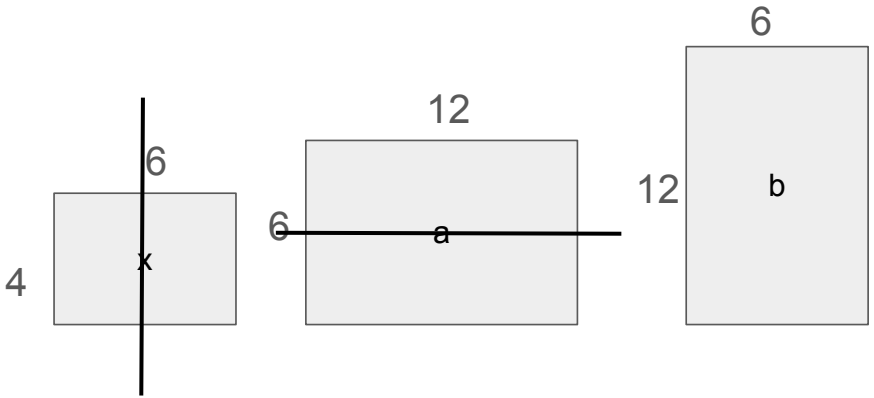
$$GELU_{\tanh}(x) = 0.5x\left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right)\right)$$

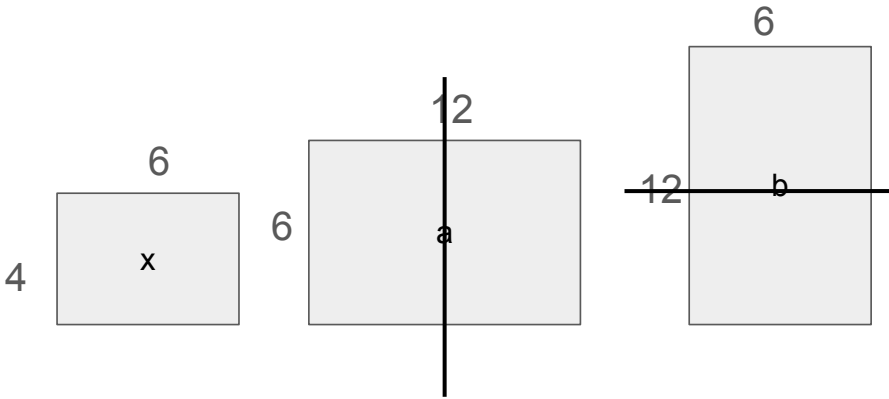
G(x) + G(y) != G(x + y)

This means we need to synchronize before the GELU is applied…

Let’s try another option

Split the a matrix vertically!



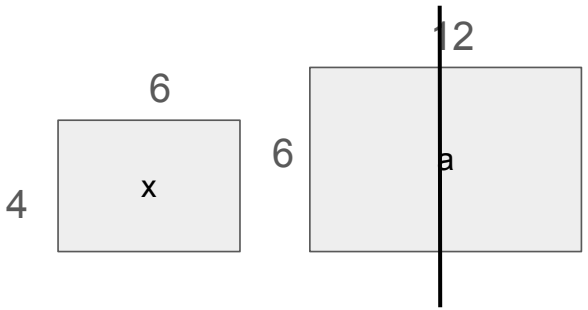


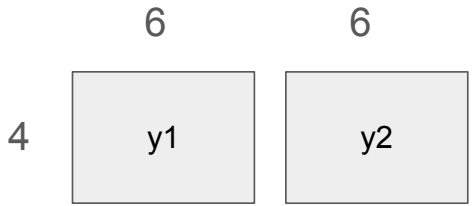
Solution

```
y = x @ a

# keep x as-is
x

a1 = a[:,0:6]
a2 = a[:,6:12]

y1 = x@a1
y2 = x@a2
```

```
b1 = b[0:6]
b2 = b[6:12]

z1 = y1@b1
z2 = y2@b2

z == z1 + z2
```

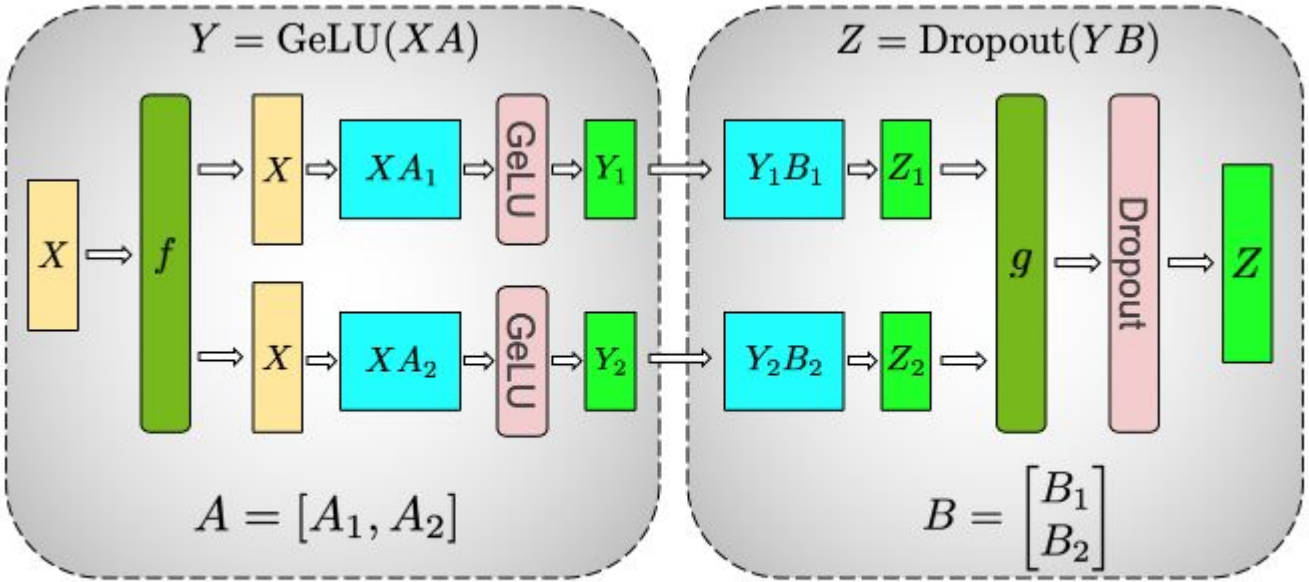
Why does this work?



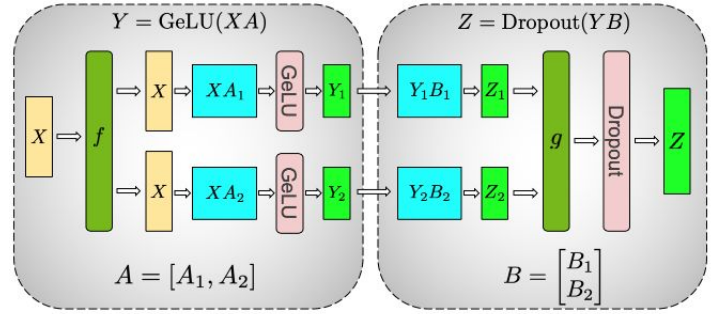


Each element of y is fully-ready
At this point

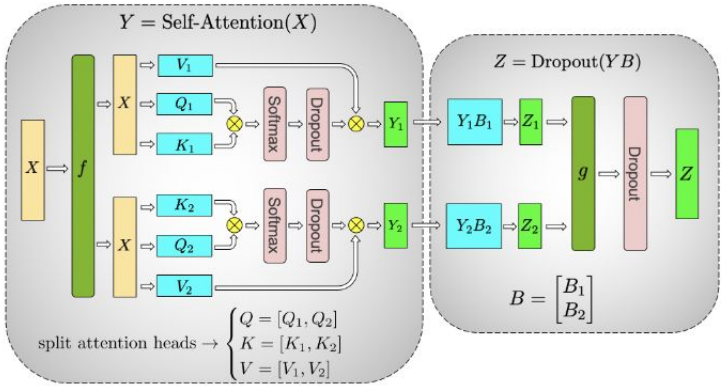
Final logic for the MLP block - with 1 all-reduce for fwd



(a) MLP



(a) MLP

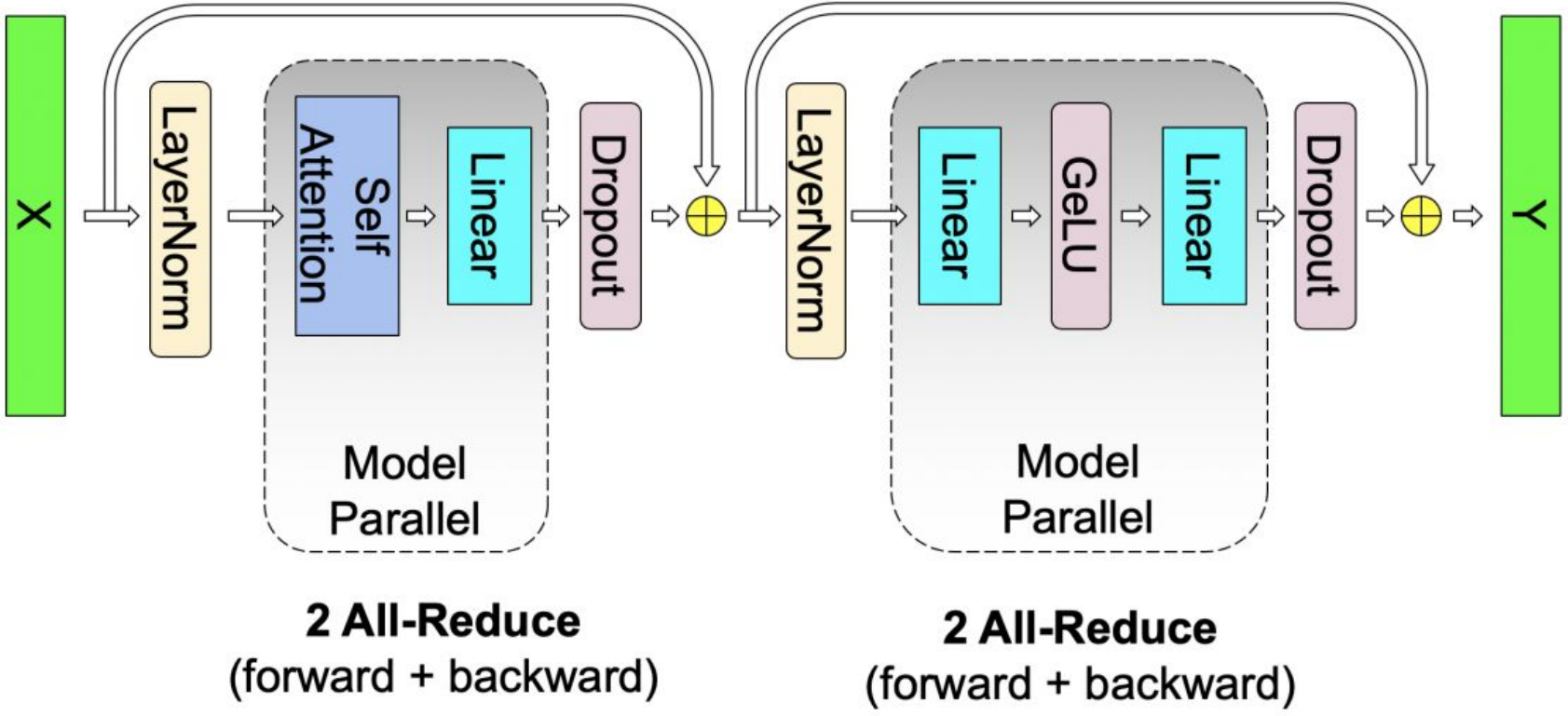


(b) Self-Attention

Figure 3. Blocks of Transformer with Model Parallelism. f and g are conjugate. f is an identity operator in the forward pass and all reduce in the backward pass while g is an all reduce in the forward pass and identity in the backward pass.

Some of the less intensive layers are simply duplicated

Even that has been improved with other advanced techniques


X
LayerNorm
Self Attention
Linear
Dropout
Model Parallel
2 All-Reduce
(forward + backward)
LayerNorm
Linear
GeLU
Linear
Dropout
Model Parallel
2 All-Reduce
(forward + backward)
Y

Complexities

Refer to:

https://danielvegamyhre.github.io/ml/performance/2025/03/30/illustrated-megatron.html for a fuller explanation of all combinations of TP

Implementing TP

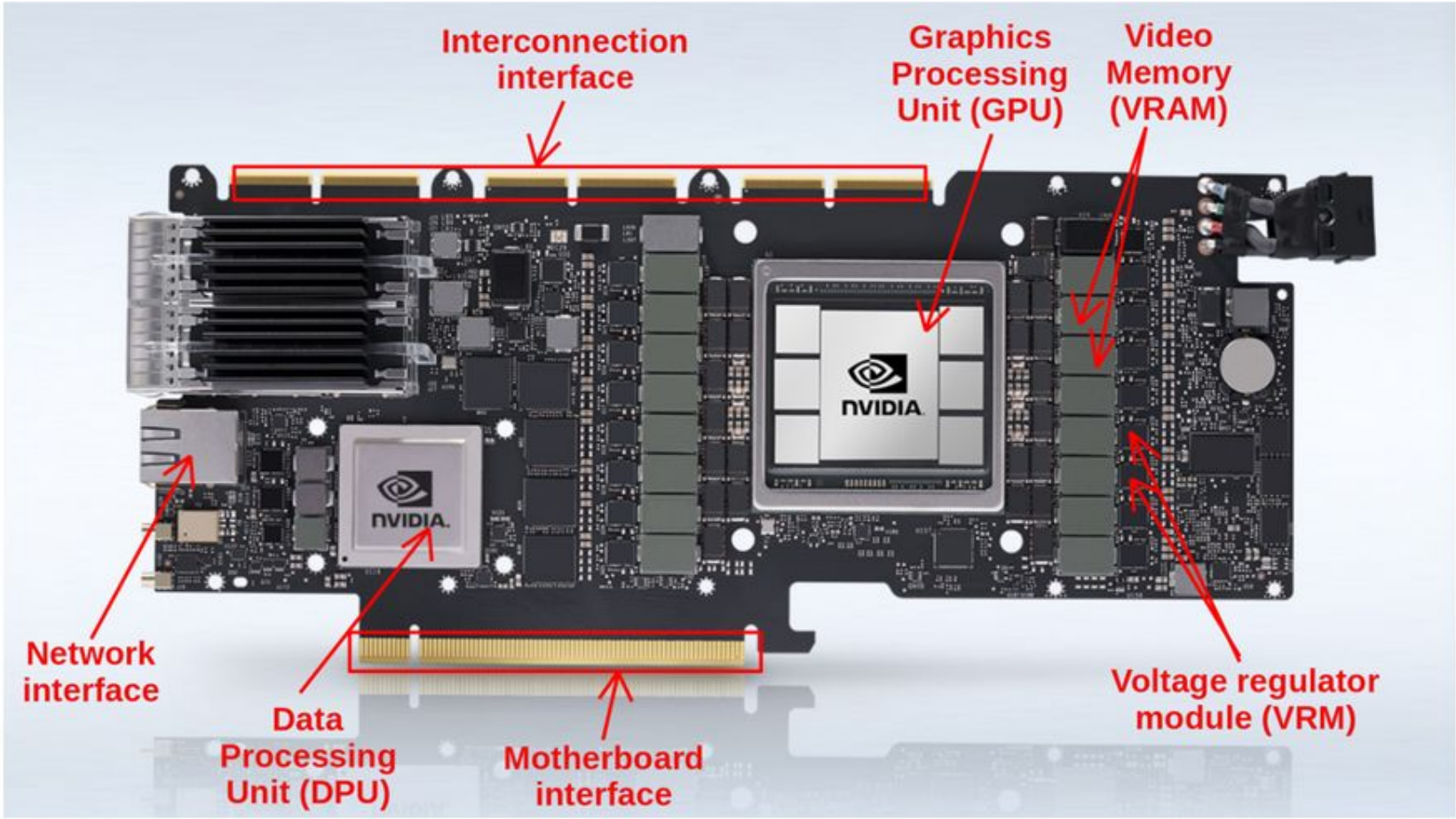
The Problem with TP

Too many network connections: 4 all-reduces per block

All-reduce is very expensive across Node boundaries


NVIDIA Tesla A100 80GB

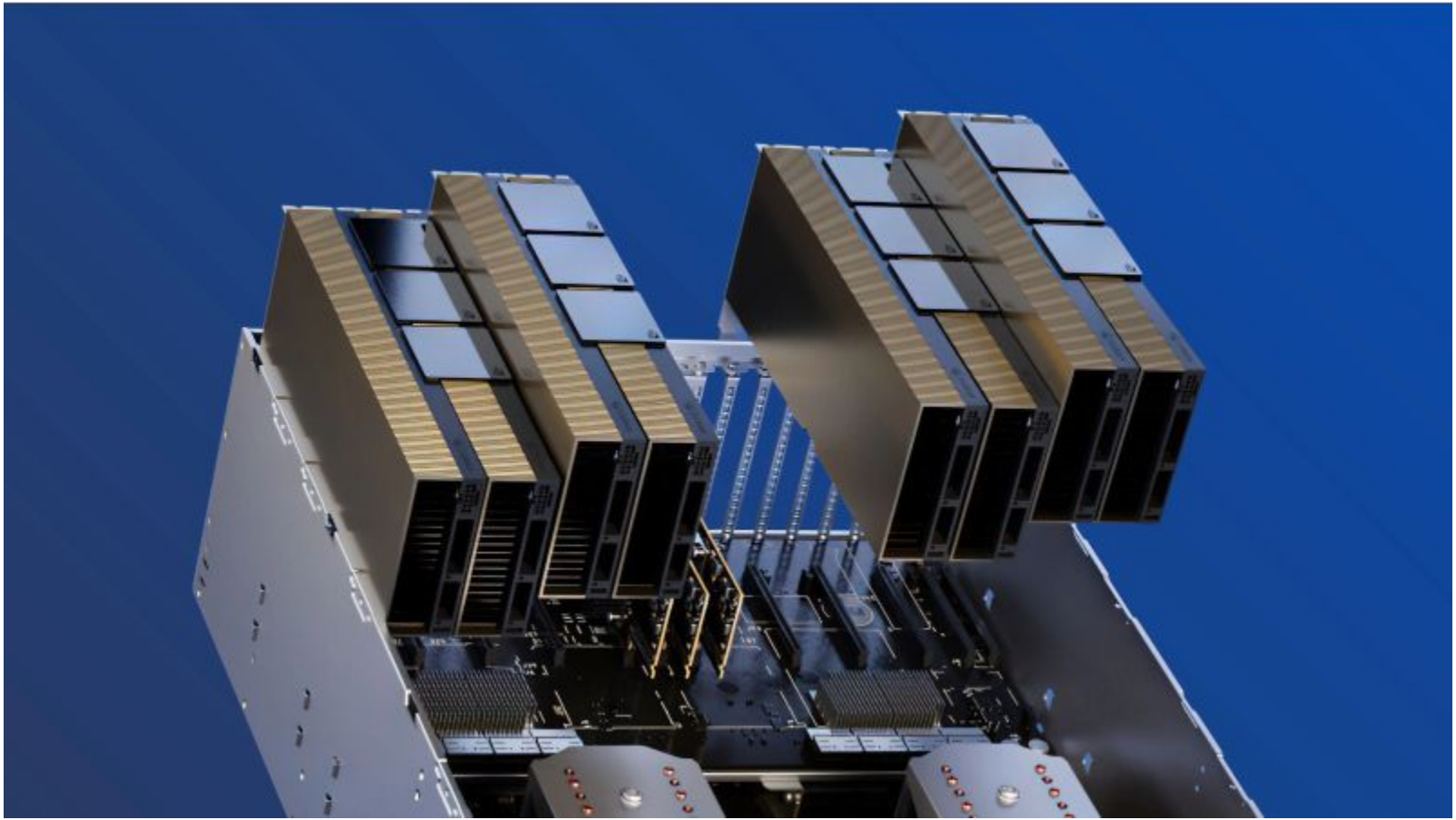
A100
X

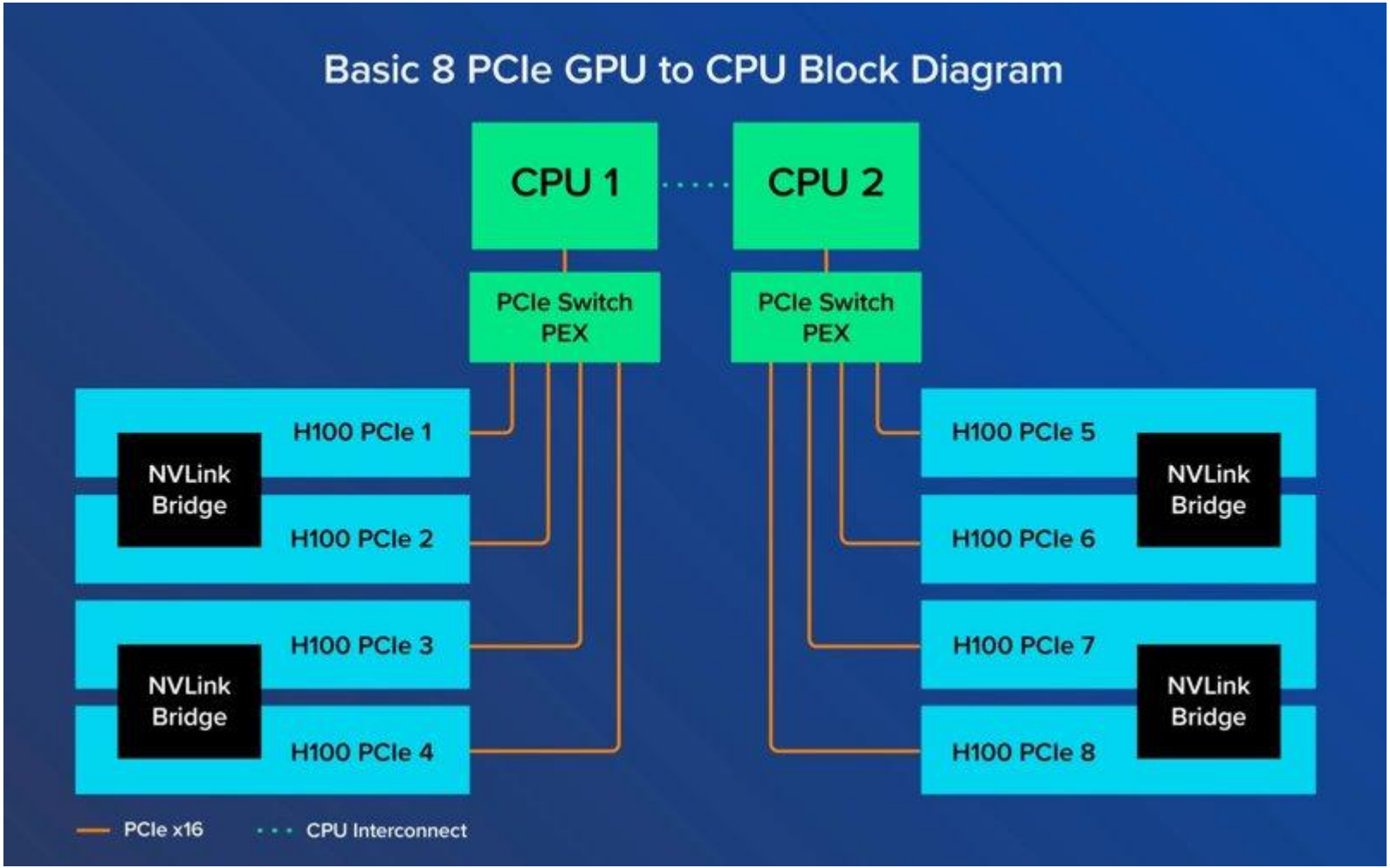


https://blog.paperspace.com/a-complete-anatomy-of-a-graphics-card-case-study-of-the-nvidia-

https://www.youtube.com/watch?v=5daQpuK0KYU

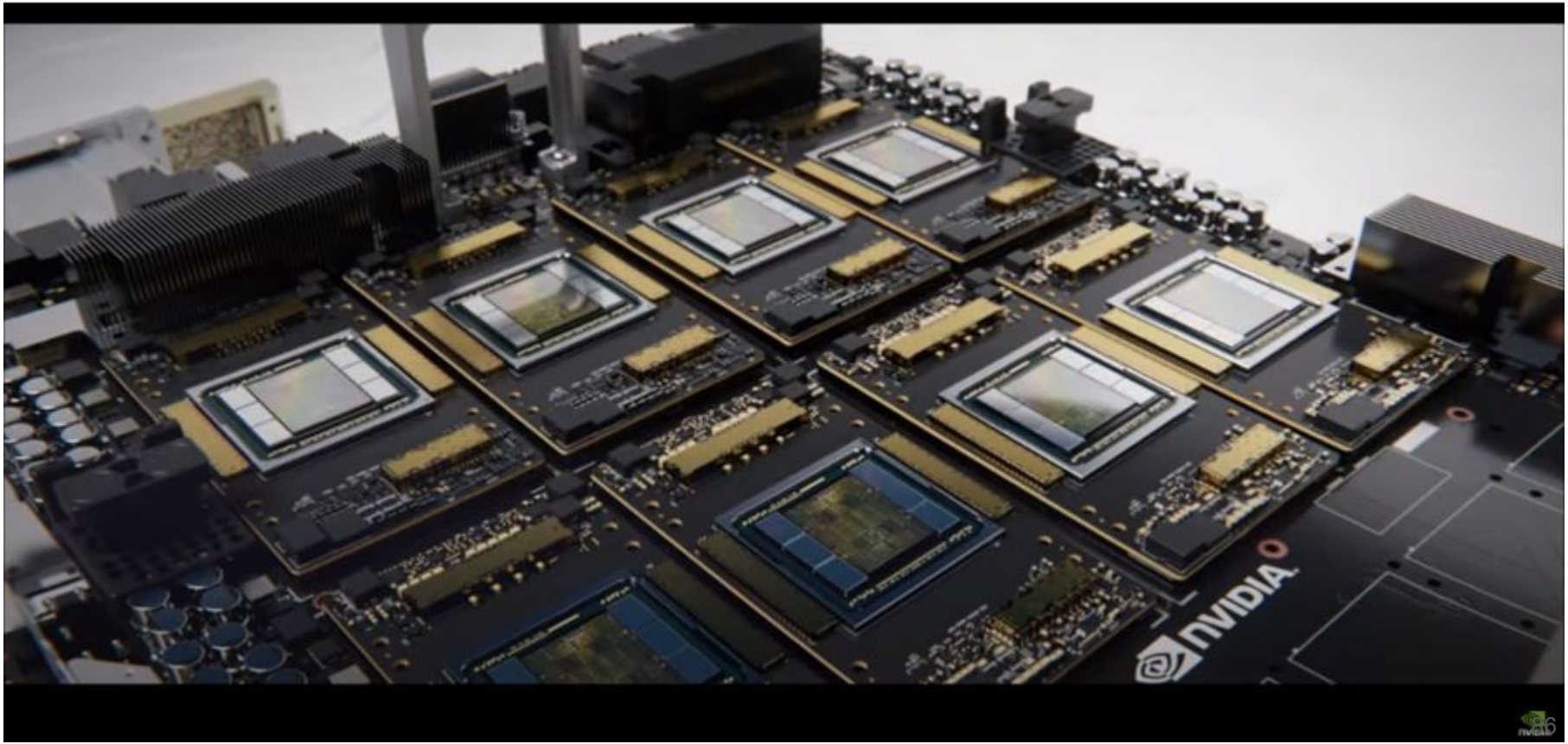
8 Nvidia H100 PCie devices connected into a node

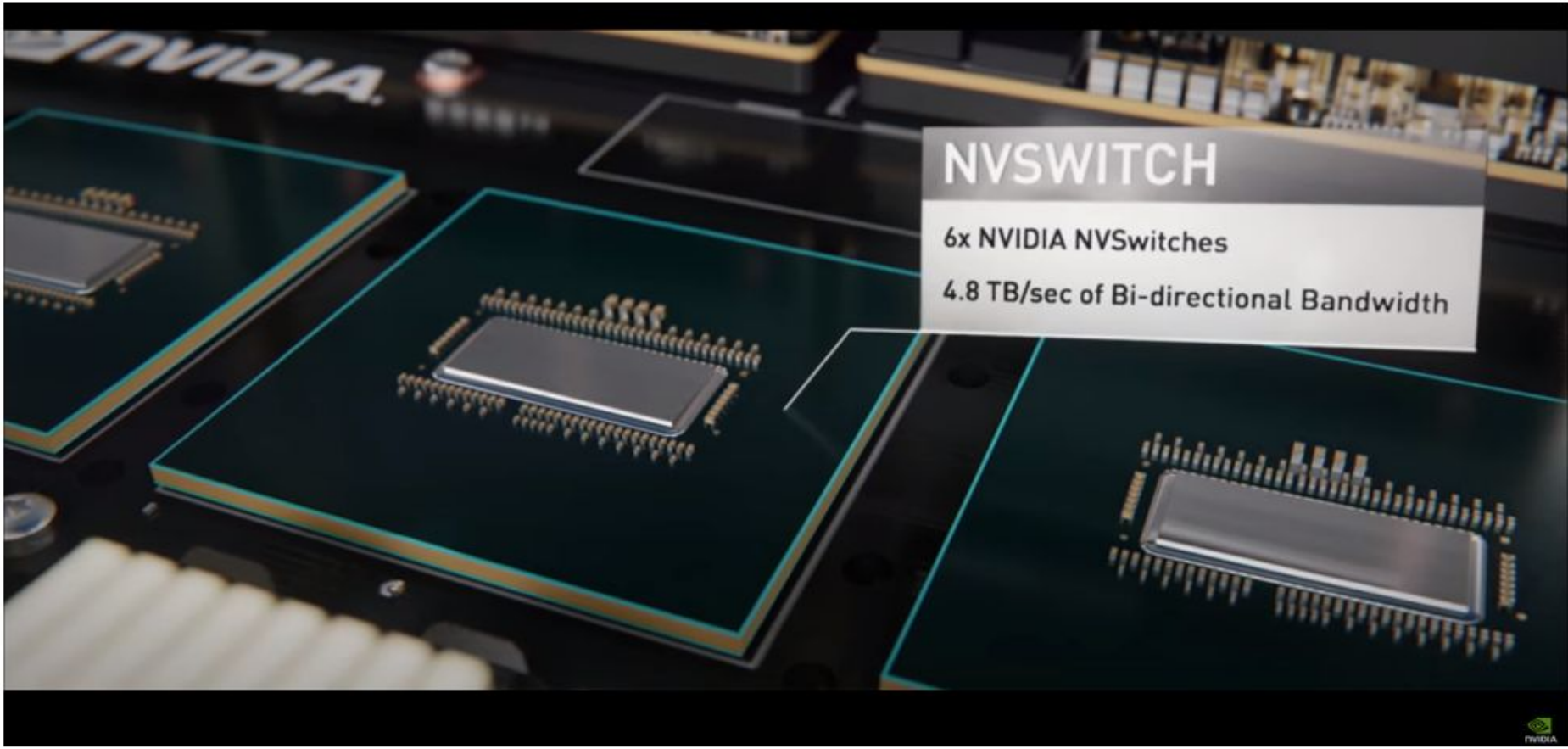


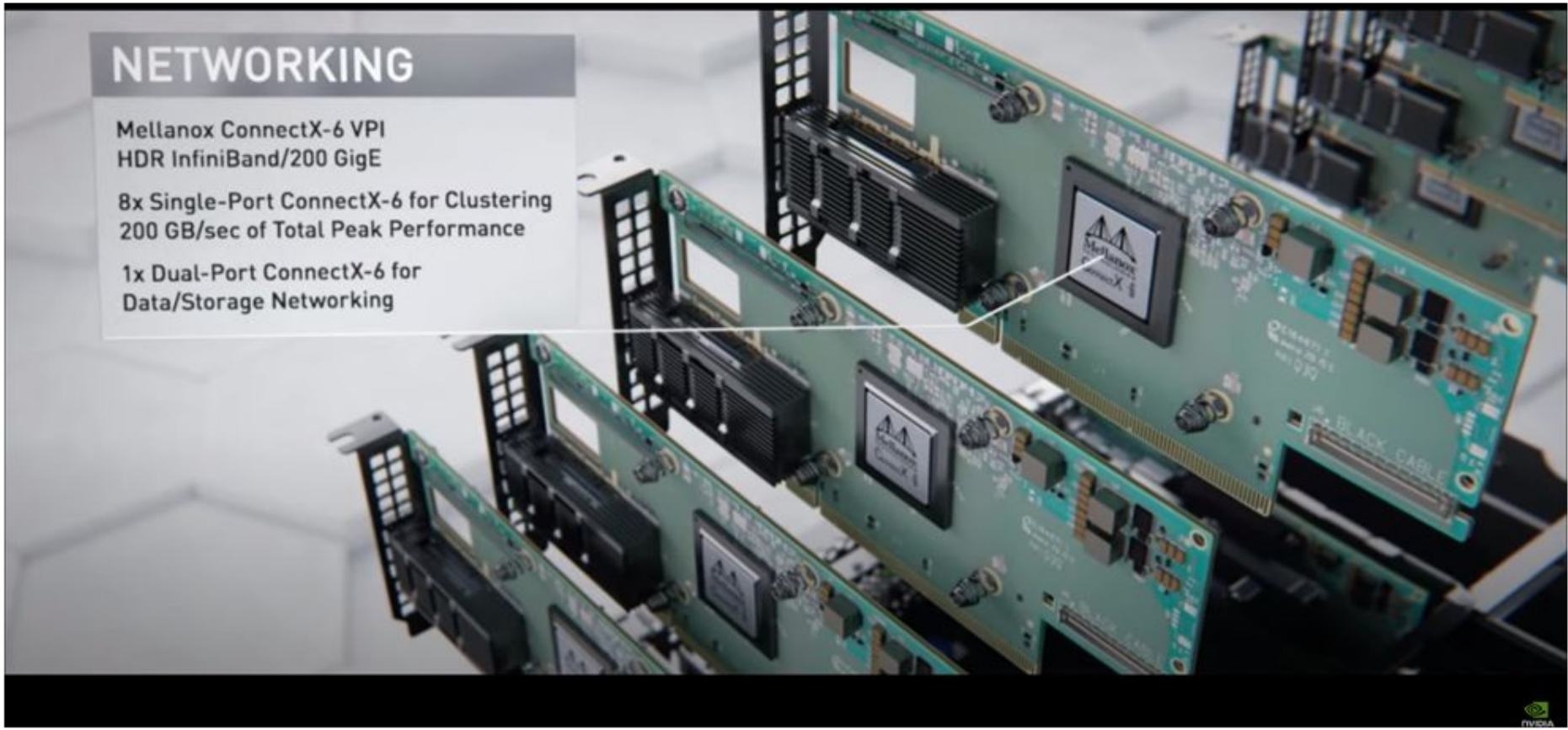

Basic 8 PCIe GPU to CPU Block Diagram
CPU 1
CPU 2
PCIe Switch PEX
PCIe Switch PEX
H100 PCIe 1
H100 PCIe 2
H100 PCIe 3
H100 PCIe 4
H100 PCIe 5
H100 PCIe 6
H100 PCIe 7
H100 PCIe 8
NVLink Bridge
NVLink Bridge
NVLink Bridge
NVLink Bridge
PCIe x16
CPU Interconnect

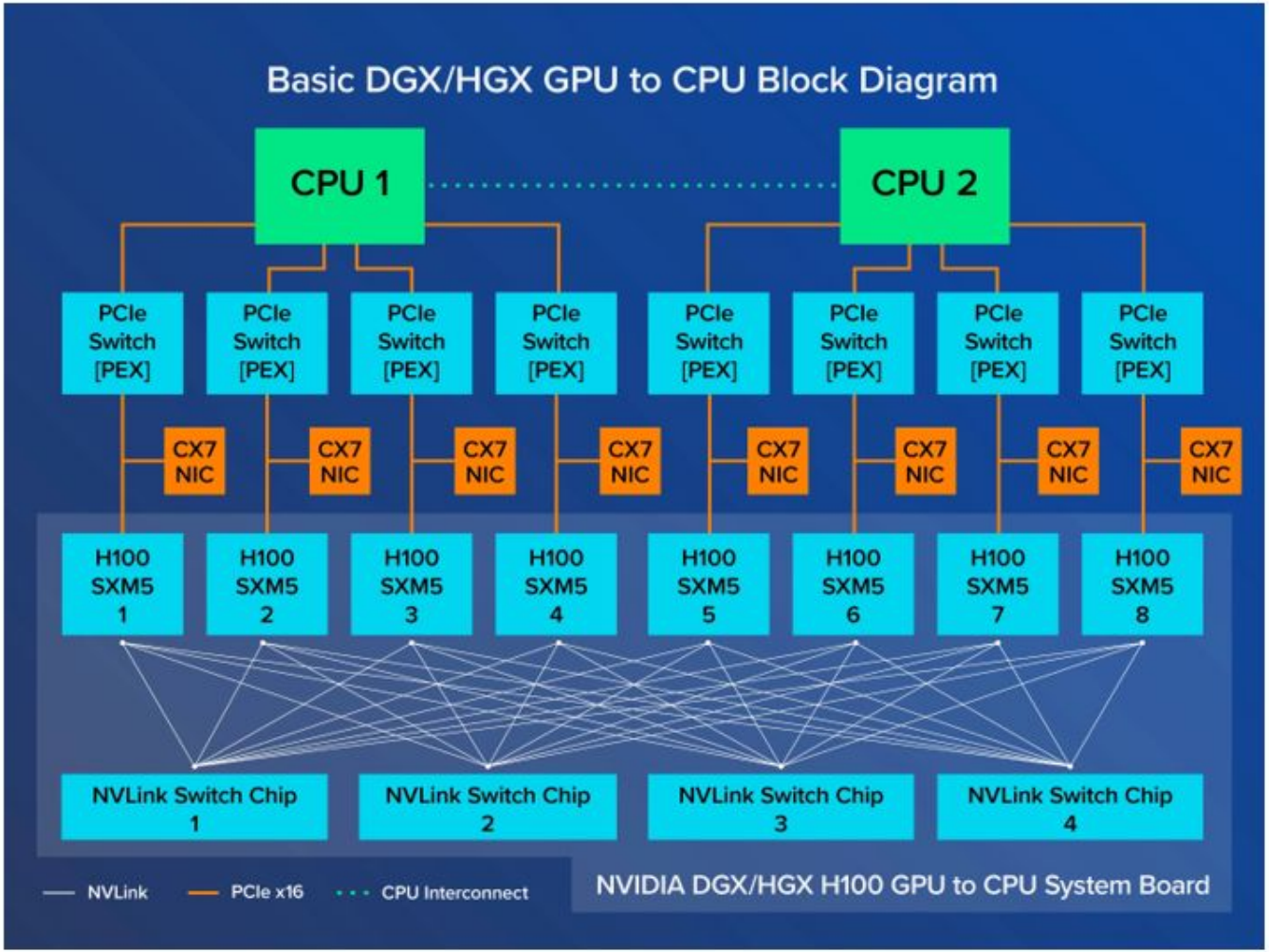
Nvidia DGX A100 “node”

8 A100’s slotted onto a DGX Base




NVIDIA
NVSWITCH
6x NVIDIA NVSwitches
4.8 TB/sec of Bi-directional Bandwidth


NETWORKING
Mellanox ConnectX-6 VPI
HDR InfiniBand/200 GigE
8x Single-Port ConnectX-6 for Clustering
200 GB/sec of Total Peak Performance
1x Dual-Port ConnectX-6 for
Data/Storage Networking


Basic DGX/HGX GPU to CPU Block Diagram
CPU 1
CPU 2
PCIe Switch [PEX]
PCIe Switch [PEX]
PCIe Switch [PEX]
PCIe Switch [PEX]
PCIe Switch [PEX]
PCIe Switch [PEX]
PCIe Switch [PEX]
PCIe Switch [PEX]
CX7 NIC
CX7 NIC
CX7 NIC
CX7 NIC
CX7 NIC
CX7 NIC
CX7 NIC
CX7 NIC
H100 SXM5 1
H100 SXM5 2
H100 SXM5 3
H100 SXM5 4
H100 SXM5 5
H100 SXM5 6
H100 SXM5 7
H100 SXM5 8
NVLink Switch Chip 1
NVLink Switch Chip 2
NVLink Switch Chip 3
NVLink Switch Chip 4
NVLink
PCIe x16
CPU Interconnect
NVIDIA DGX/HGX H100 GPU to CPU System Board

Now what?

All-reduce within a single node is very fast,

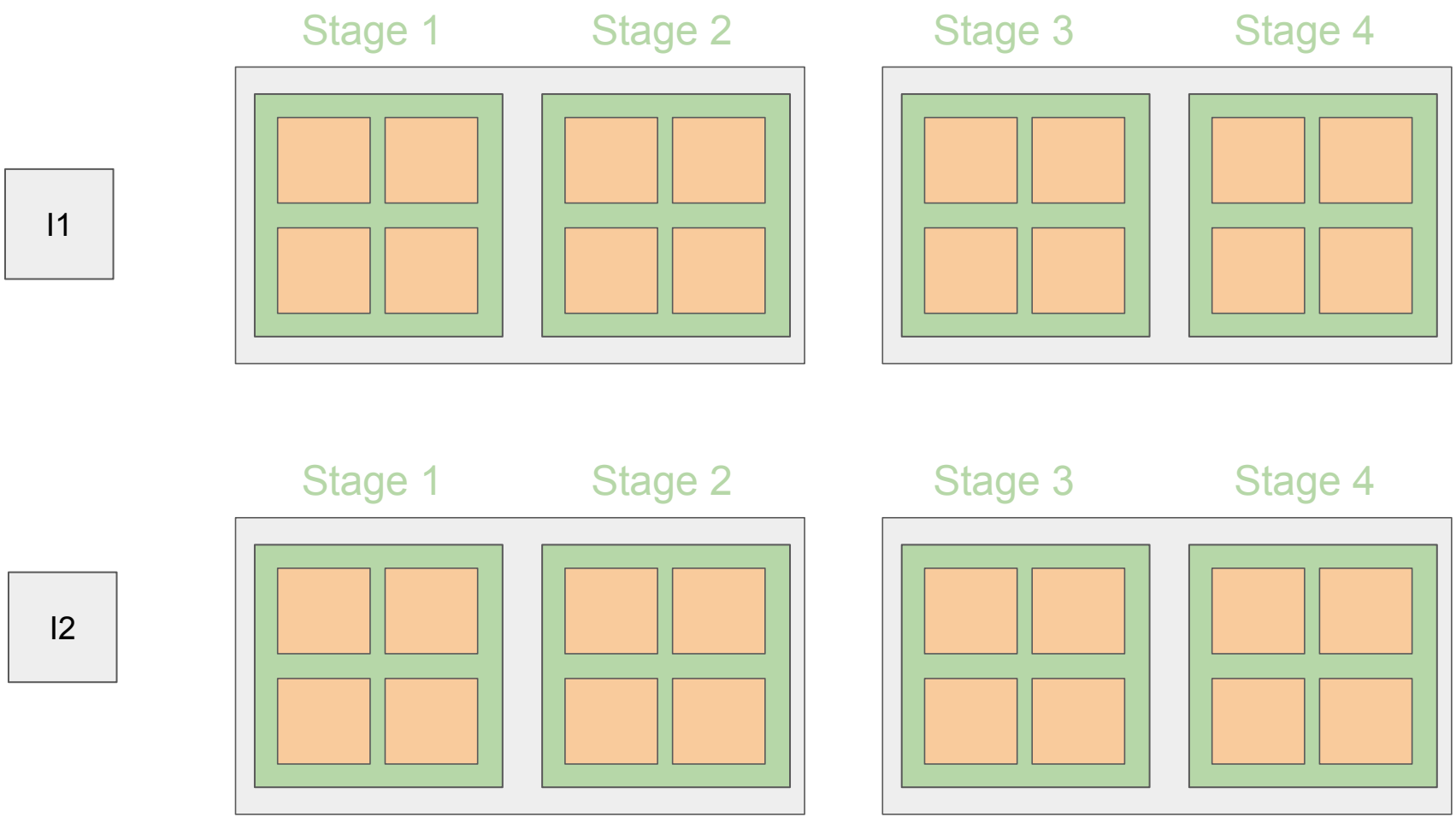
But the moment you have to go to multiple Nodes, > 8 gpus, hit is going to be very high..

What can you do about this?

3 techniques compared

DDP	• Easy to implement	• Does not scale too much • Can’t use if model is too big
PP	• Simple Network communication	• Too much wasted compute in terms of bubbles
TP	• Best Compute balance	• Very costly over multiple nodes

3D Parallelism: d-t-p


Stage 1
Stage 2
Stage 3
Stage 4
I1
Stage 1
Stage 2
Stage 3
Stage 4
I2

3d-parallelism

Tries to use best of all 3 techniques

Optimal schedule, it d-p-t depends on a lot of factors:

1. Model size
2. Data size
3. Number of gpus
4. GPU networking interconnects
5. Task-characteristics: inference vs training
6. etc

FSDP: Fully sharded Data Parallel

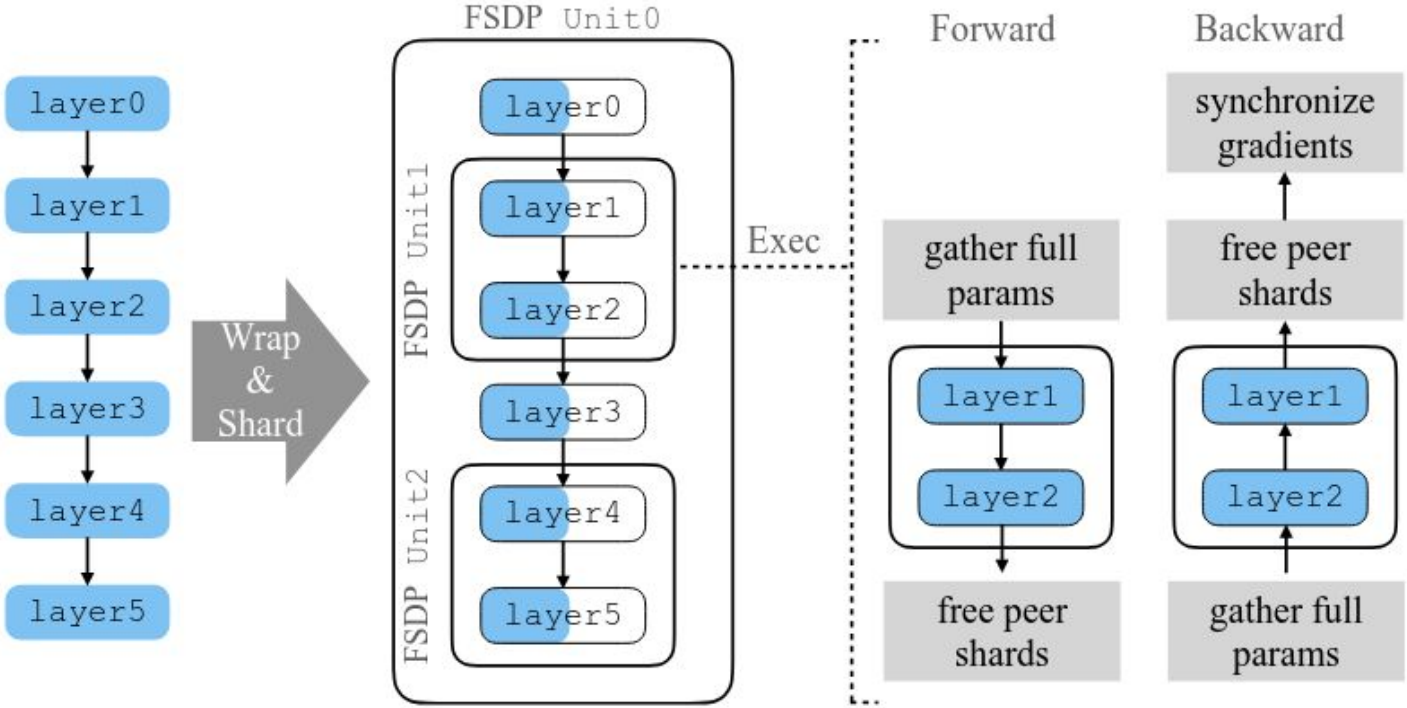


Figure 1: FSDP Algorithm Overview

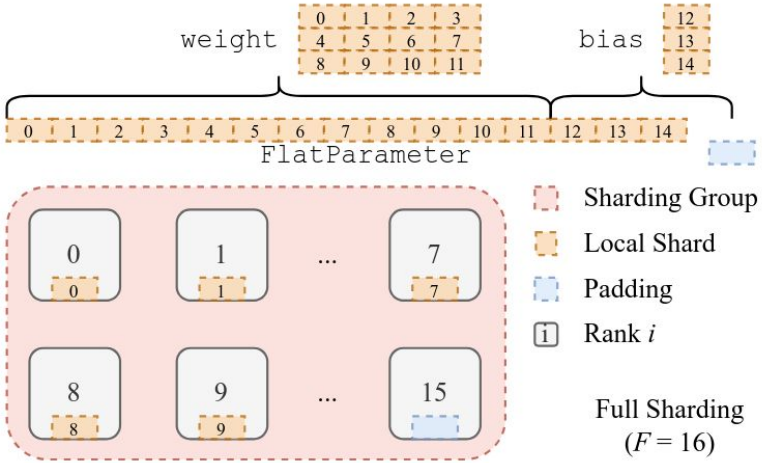


Figure 3: Full Sharding Across 16 GPUs

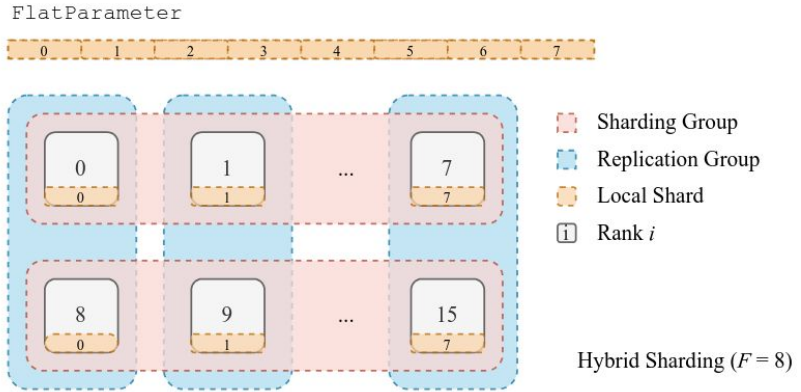


Figure 4: Hybrid Sharding on 16 GPUs: GPUs are configured into 2 sharding groups and 8 replication groups

Implementing FSDP

Challenges

Computation-communication overlap

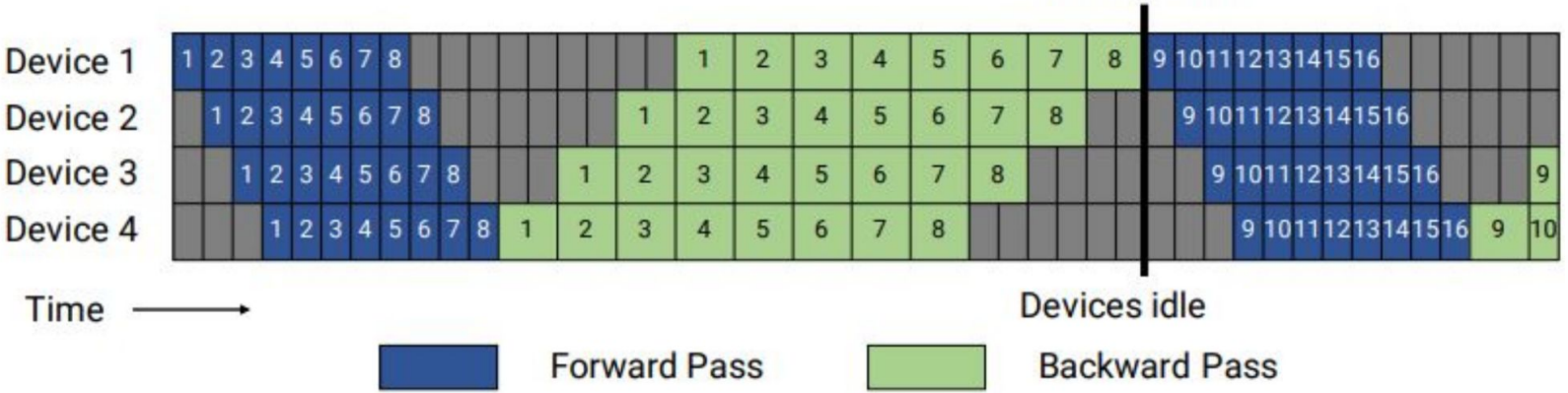
Network speeds dominate computation speeds: leading to around 50% wasted compute on large clusters

Designing large clusters of gpus to manage communications

Congestion control due to bursty nature of training communication

Some Prior Art

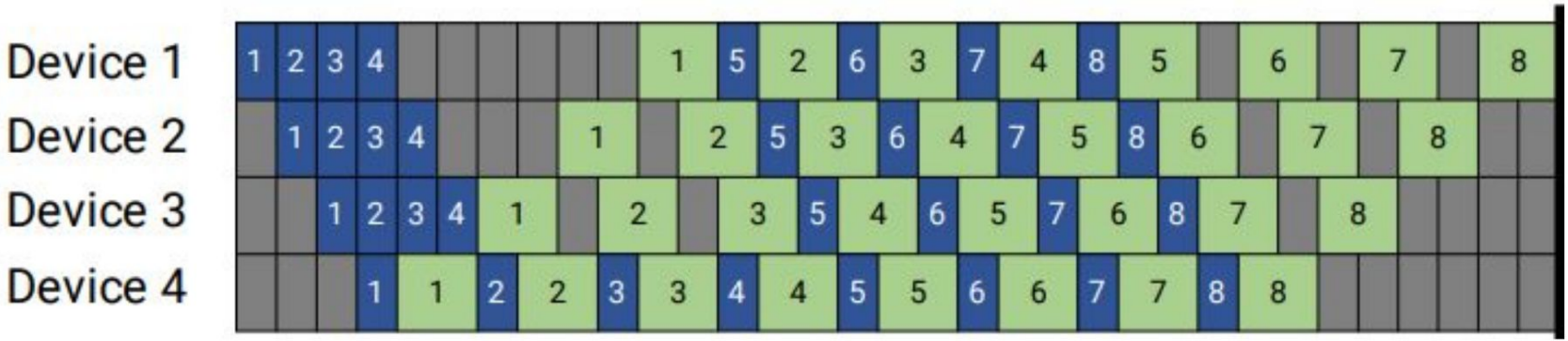
Pipeline Parallelism advancements - GPipe



GPipe

https://www.cs.toronto.edu/~cmaddis/courses/csc2541_w25/presentations/ptd-p.pdf

Pipeline Parallelism advancements - PipeDream



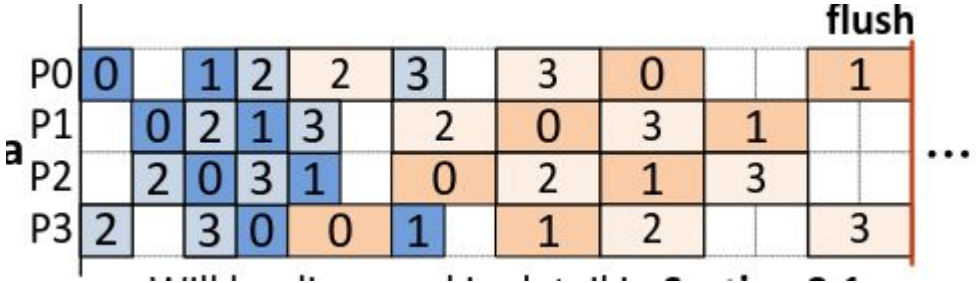
https://www.cs.toronto.edu/~cmaddis/courses/csc2541_w25/presentations/ptd-p.pdf

Chimera

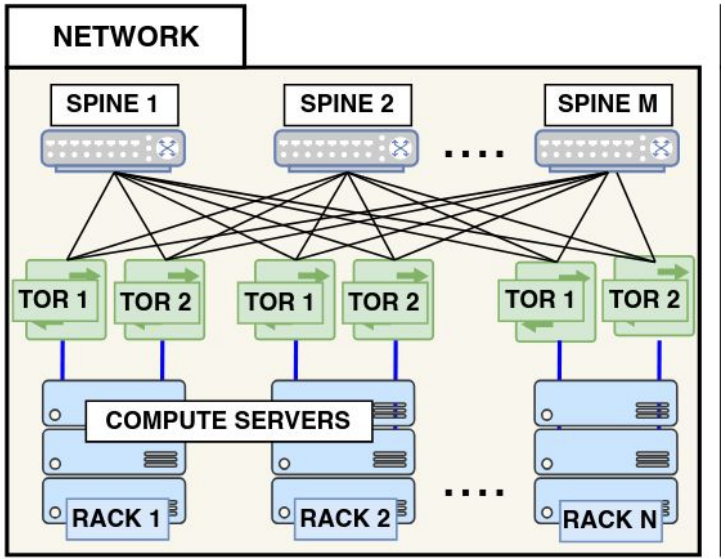
https://dl.acm.org/doi/10.1145/3458817.3476145

Table 2: Comparison between different pipeline schemes.

Pipeline Schemes	Bubble Ratio	Weights Memory	Activations Memory	Convergence Friendly
PipeDream [38]	≈ 0 👍👍	$[M_\theta,\ D * M_\theta]$ [1] 👎	$[M_a,\ D * M_a]$ [1] 👍	Asynchronous 👎
PipeDream-2BW [39]	≈ 0 👍👍	$2M_\theta$ 👍	$[M_a,\ D * M_a]$ [1] 👍	
GPipe [26]	$(D-1)/(N+D-1)$ 👎	M_θ 👍	$N * M_a$ 👎👎	Synchronous 👍
GEMS [28]	$\approx (D-1)/(D+\frac{1}{2})$ 👎👎	$2M_\theta$ 👍	M_a 👍👍	
DAPPLE [16]	$(D-1)/(N+D-1)$ 👎	M_θ 👍	$[M_a,\ D * M_a]$ [1] 👍	
Chimera (this work)	$(D-2)/(2N+D-2)$ 👍	$2M_\theta$ 👍	$[(D/2+1)M_a, D * M_a]$ [1] 👍+	

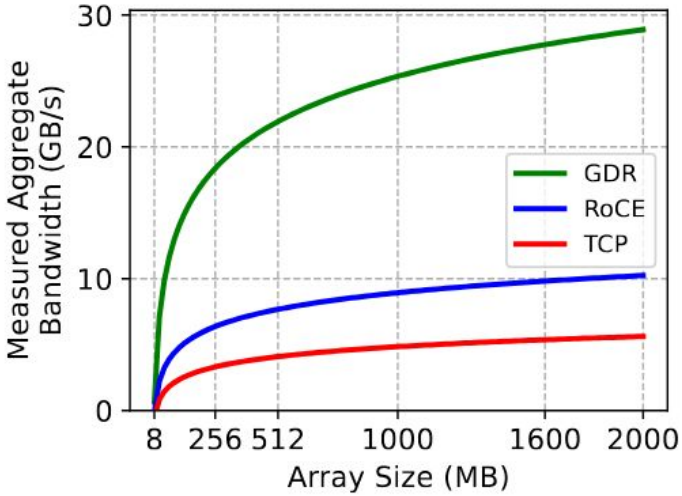


IBM’s Compute Setup - Vela (ASPLOS’25)



https://dl.acm.org/doi/10.1145/3676641.3716280

Vela



(a) 1024 GPUs AllReduce performance.

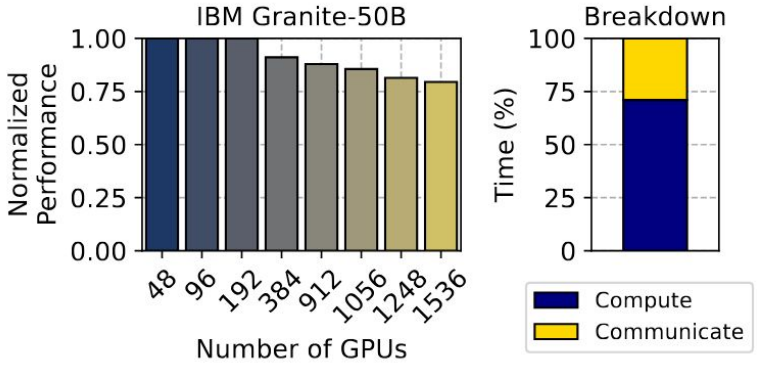


Figure 1. Training performance of IBM Granite-50B model on Vela at different scales normalized to ideal throughput.

Alibaba HPN (SIGCOMM ‘24)

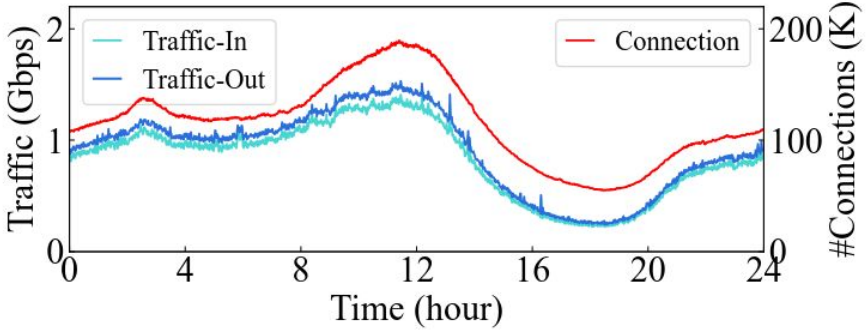


Figure 1: Traditional cloud computing traffic pattern.

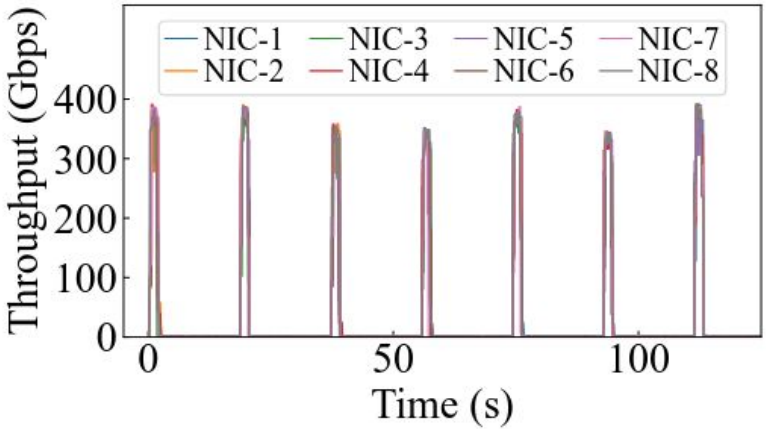


Figure 2: NIC egress traffic pattern during production model training.

SuperBench (Microsoft), ATC'24

Cloud Environment Cloud infrastructure environments can introduce additional incidents, as they differ from vendors' qualification environments in terms of power, temperature, and other factors. For example, data centers in tropical areas experience more incidents due to higher temperatures. We observed a 35× increase in defective InfiniBand links with $> 10^{-12}$ bit error rate in data centers in tropical areas compared to data centers in higher latitudes, leading to significantly degraded performance for training and inference. Another example is GPU throttling. Even within the same data center, different racks or locations within the same rack can exhibit varying temperatures. However, all GPUs are designed with identical cooling and heatsinks by vendors, resulting in GPUs located in warmer locations potentially experiencing thermal throttling if they cannot receive more cool air.

Astral (Tencent) (SIGCOMM ‘25)

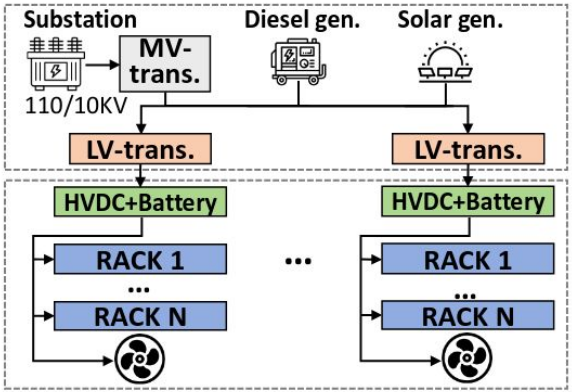


Figure 4: Hierarchy of the distributed HVDC power system.

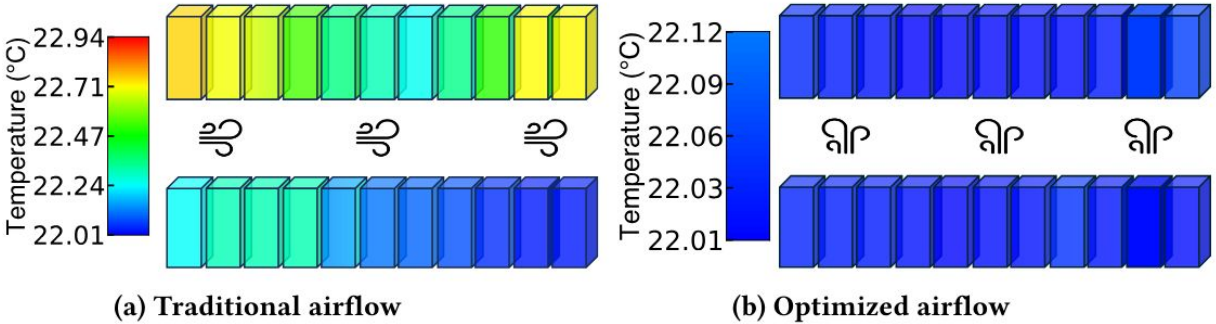


Figure 5: Temperature distribution with air cooling.

https://dl.acm.org/doi/pdf/10.1145/3718958.3750521

Recap

-> How multiple GPUs are used in concert

-> What are the challenges, research areas?

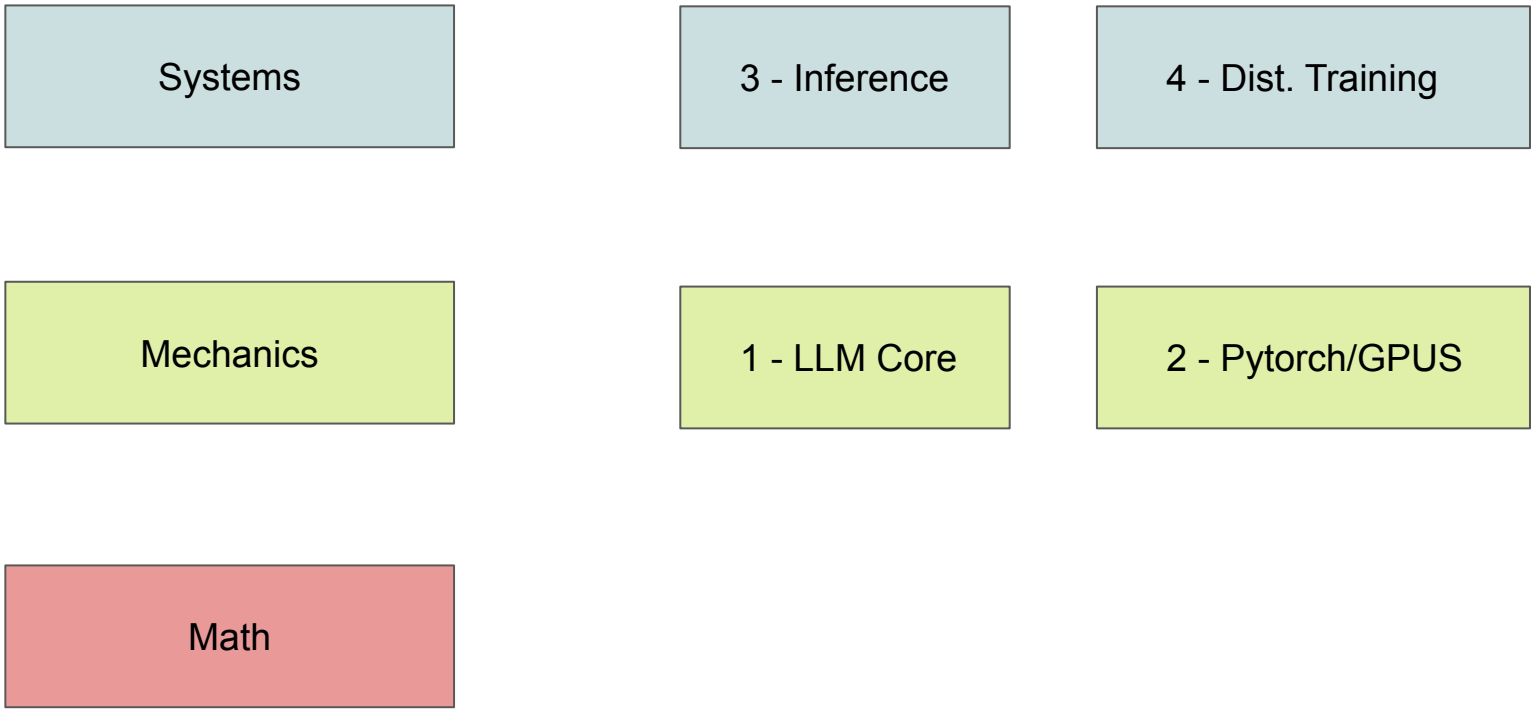
Distributed Training Algorithms vs Systems

- Data Parallelism (DDP)
- Tensor Parallelism (TP)
- Pipeline Parallelism (PP)

- Full Sharded Data Parallelism (FSDP)
  - With variants - like Full/Hybrid Shard

- 3D parallelism
- Context/Sequence Parallelism
- Expert Parallelism (EP)

- Pytorch in-built DDP
- Hugging Face Accelerate
- Nvidia Megatron
- Pytorch in-built FSDP
- Microsoft DeepSpeed ZeRO family


Systems
3 - Inference
4 - Dist. Training
Mechanics
1 - LLM Core
2 - Pytorch/GPUS
Math