

# AWS re:Invent

NOV. 28 – DEC. 2, 2022 | LAS VEGAS, NV

CMP401

# Scaling ML training and inference workloads on Amazon EC2 and PyTorch

Hamid Shojanazeri

ML Engineer, Applied AI team  
Meta

Ankur Srivastava

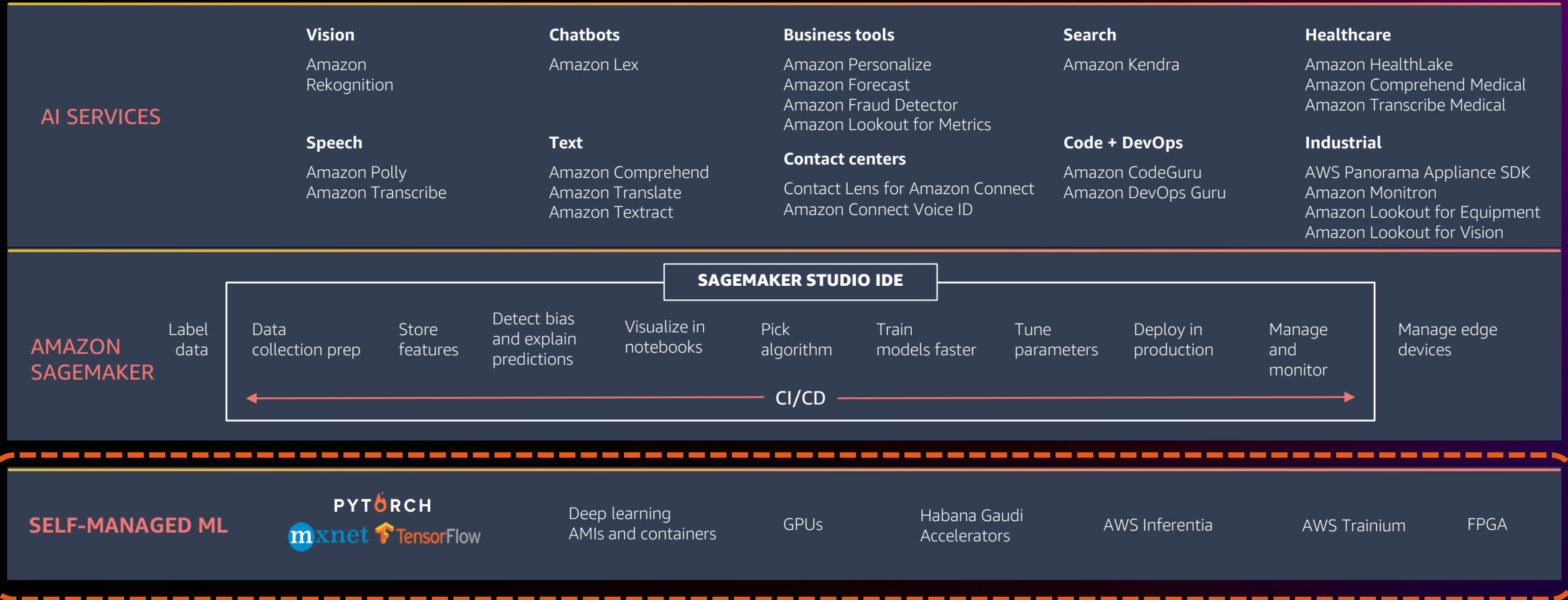
Sr. Solution Architect, AWS  
AWS



© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# The AWS ML Stack

## BROAD SET OF MACHINE LEARNING CAPABILITIES



ML Frameworks WWSO helps customers developing custom ML



# What are we talking about today?

The scaling problem and where PyTorch FSDP comes in

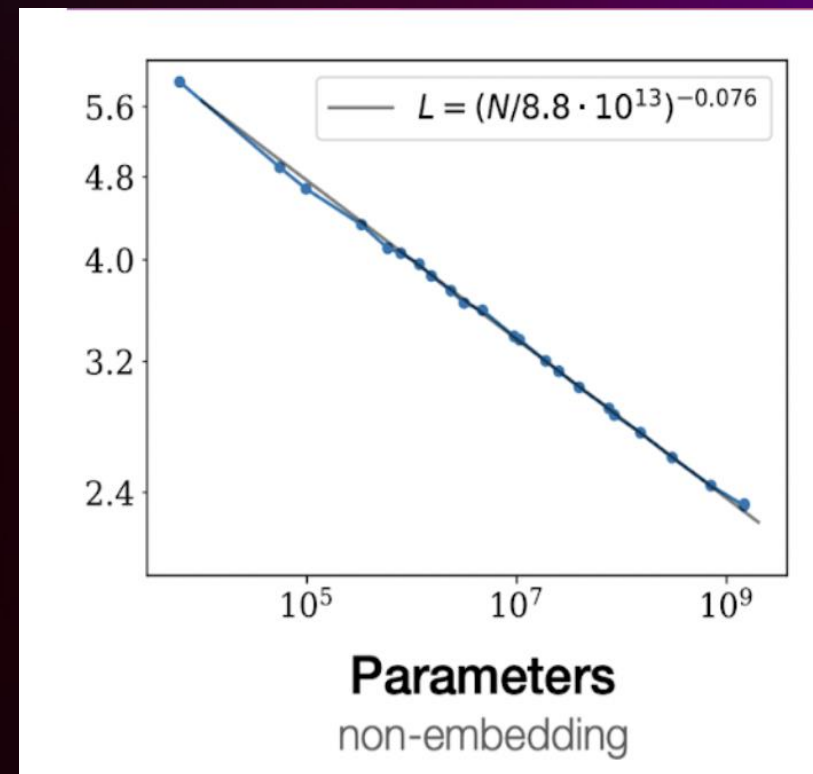
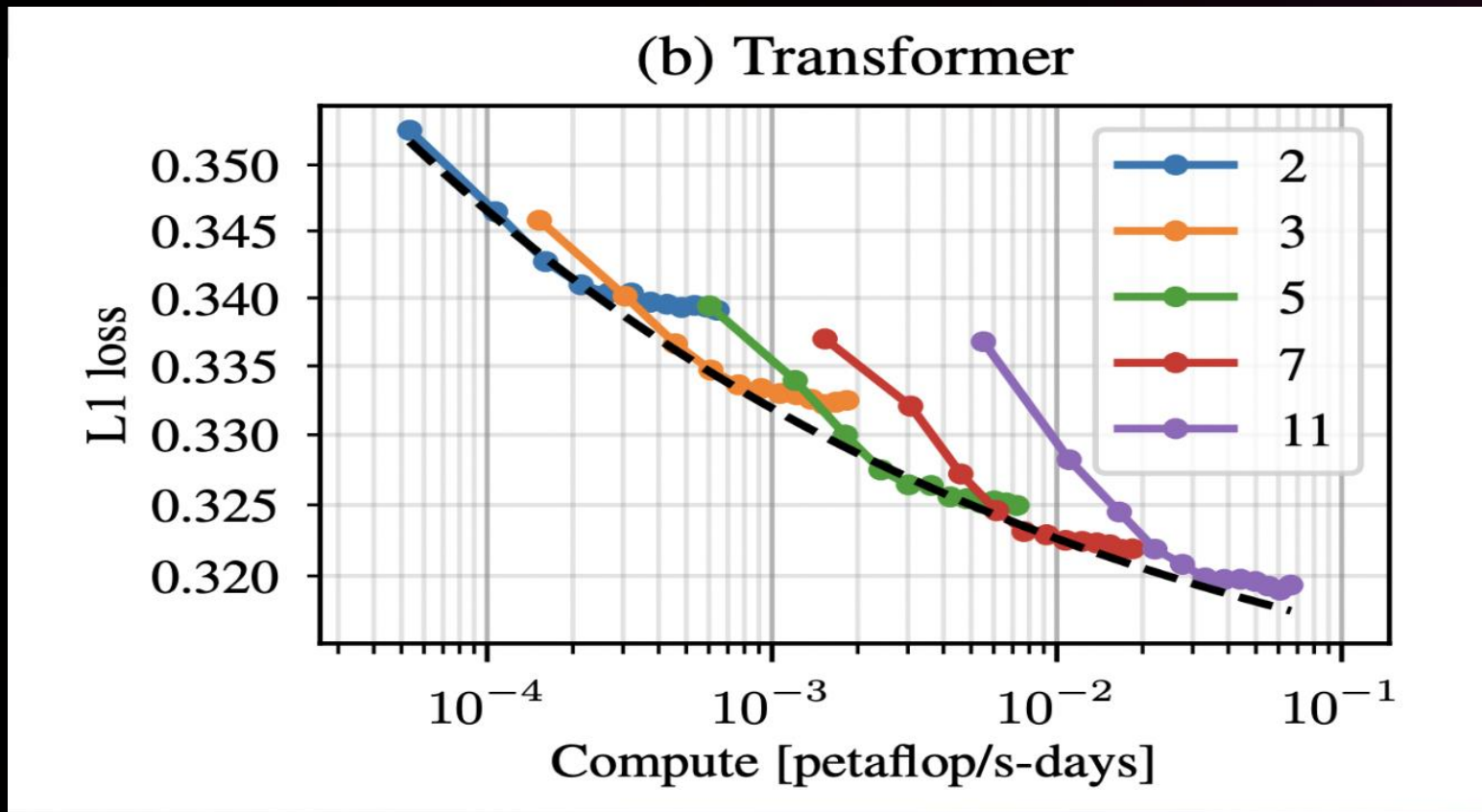
What is FSDP exactly?

Code snippets

Details: Activation Checkpointing, Sharding Strategies

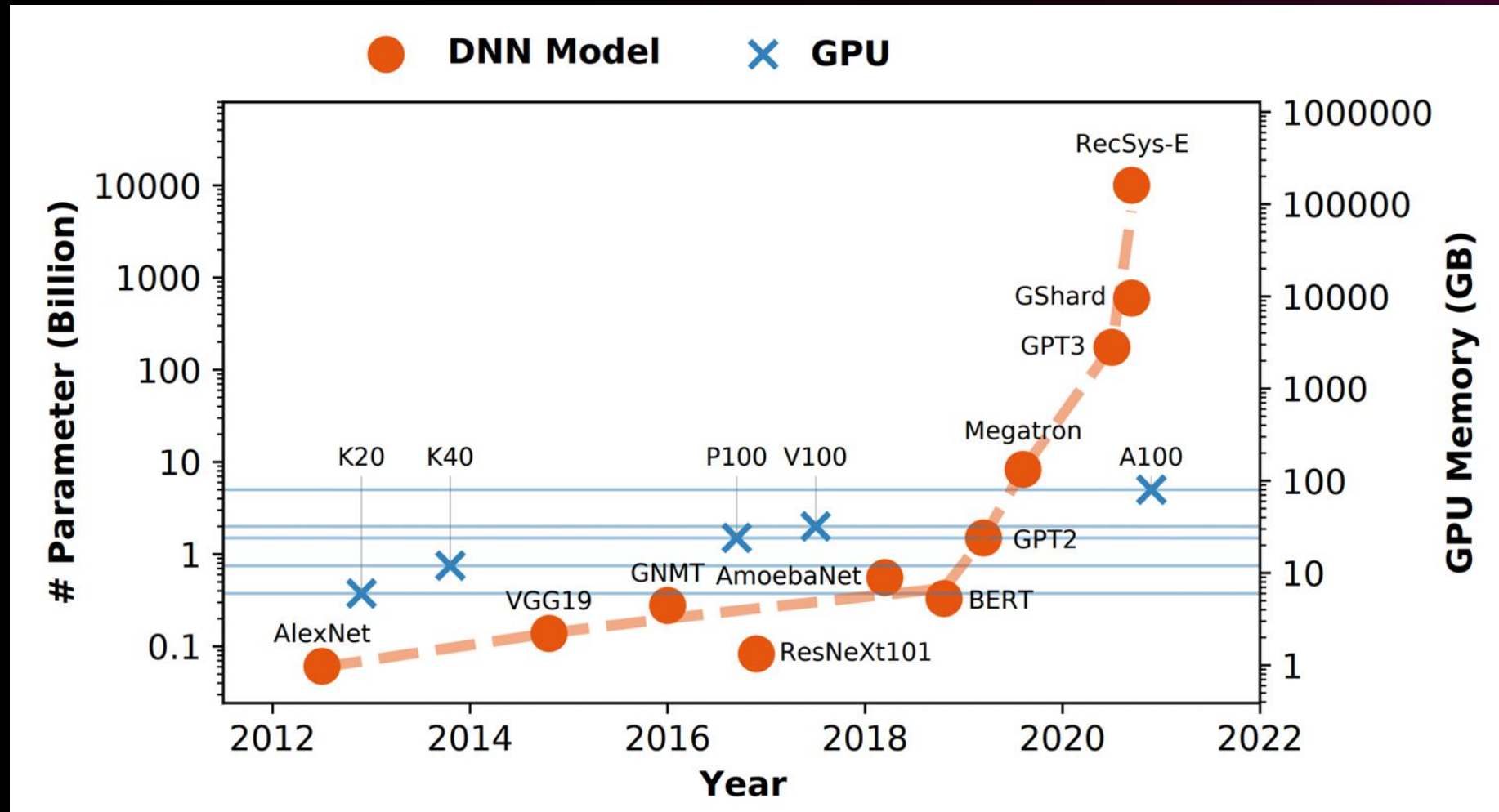
Additional new PyTorch features for extremely large models

# Scaling problem: Strong payoff from increasing model size



Performance improves smoothly as you increase model size, compute time and dataset size.  
(power law or power law + constant)

# Thus, AI models have gone up 10,000x in size (but... GPU memory has only gone up ~10x)



# FSDP ( Fully Sharded Data Parallel)

Train a much larger model with same resources

Resource efficiency : Significantly reduce memory footprint on each GPU

Compute efficiency : Overlapping compute and communication

Ease of use : Lightweight config, just few knobs

Model	Number of layers	Hidden size	Attention heads	Model size, billions of parameters
GPT 175B	96	12288	96	175
GPT 1T	128	25600	160	1008

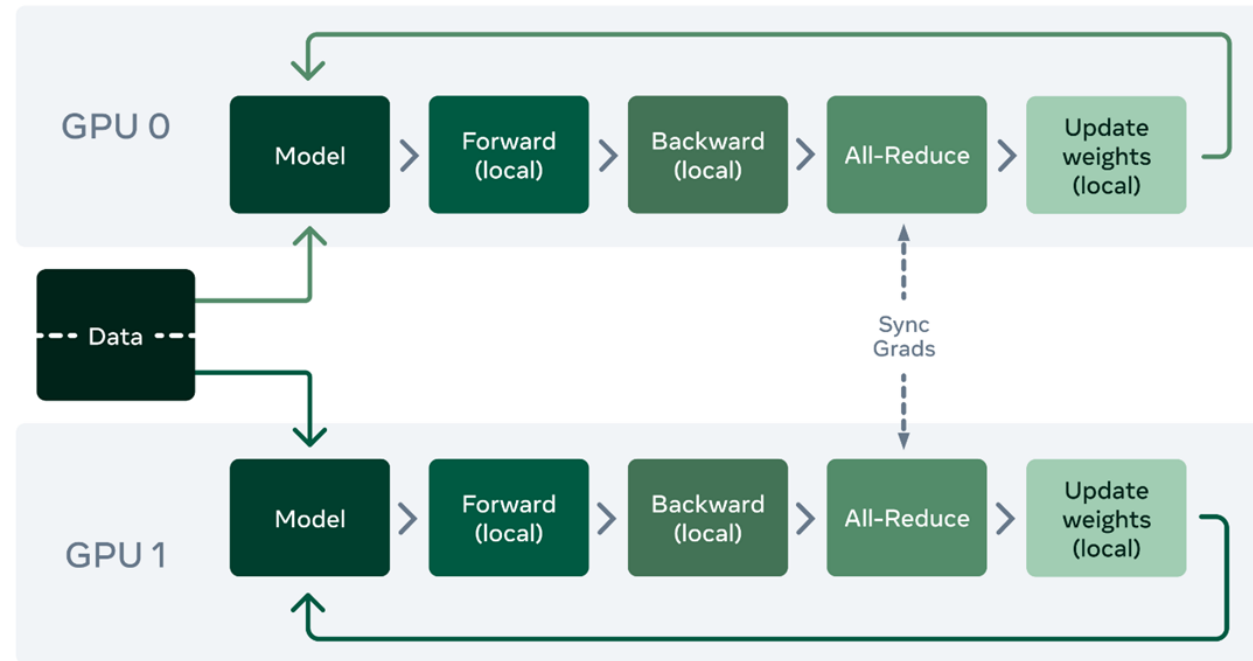


# FSDP compared to DDP. DDP = sharding batch

**Distributed Data Parallel** = models are duplicated on multiple GPUs.

Data is sharded and submitted, each GPU processes.

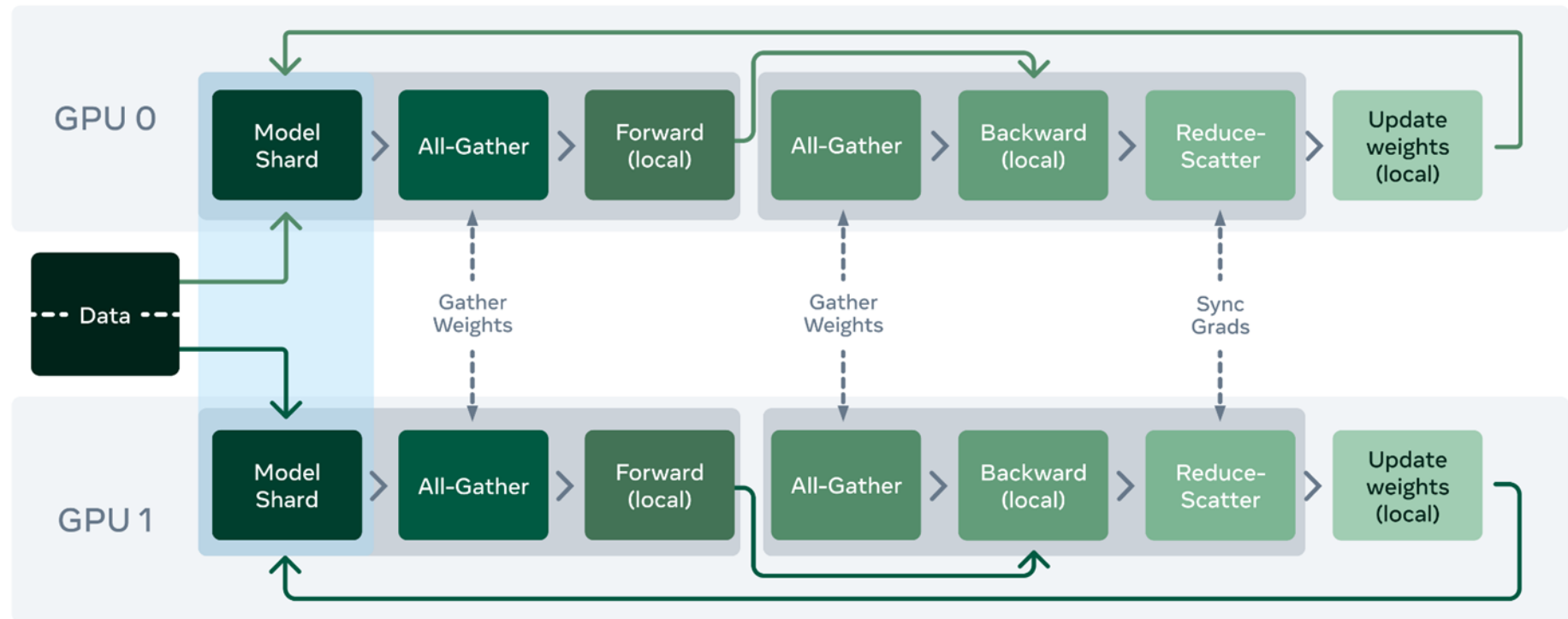
Standard data parallel training





# Fully Sharded Data Parallel (FSDP)

Fully sharded data parallel training



With sufficient inter-node communication speed, scaling with FSDP can be linear.

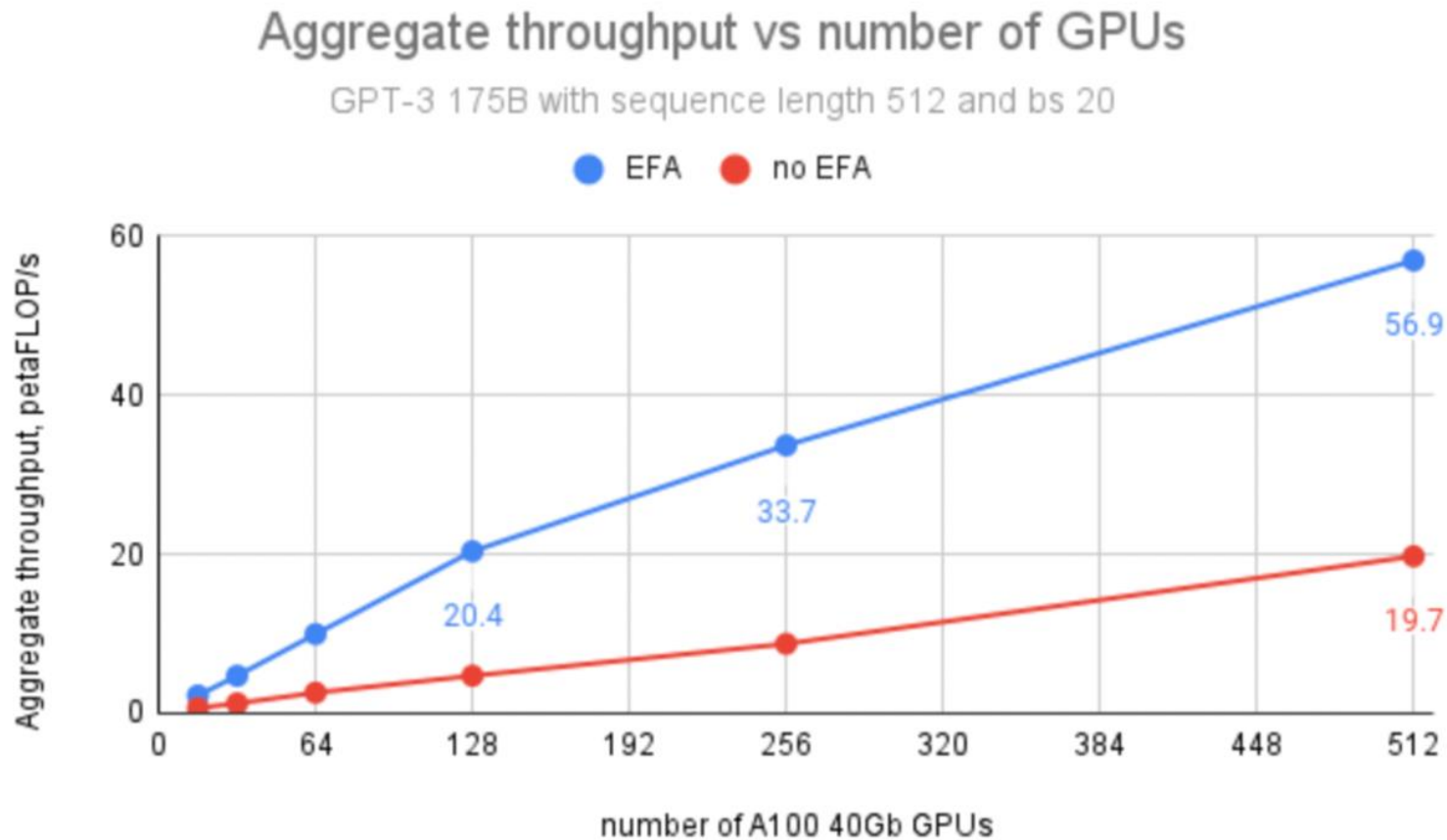


Figure 7: Aggregate throughput vs number of GPUs (175B)

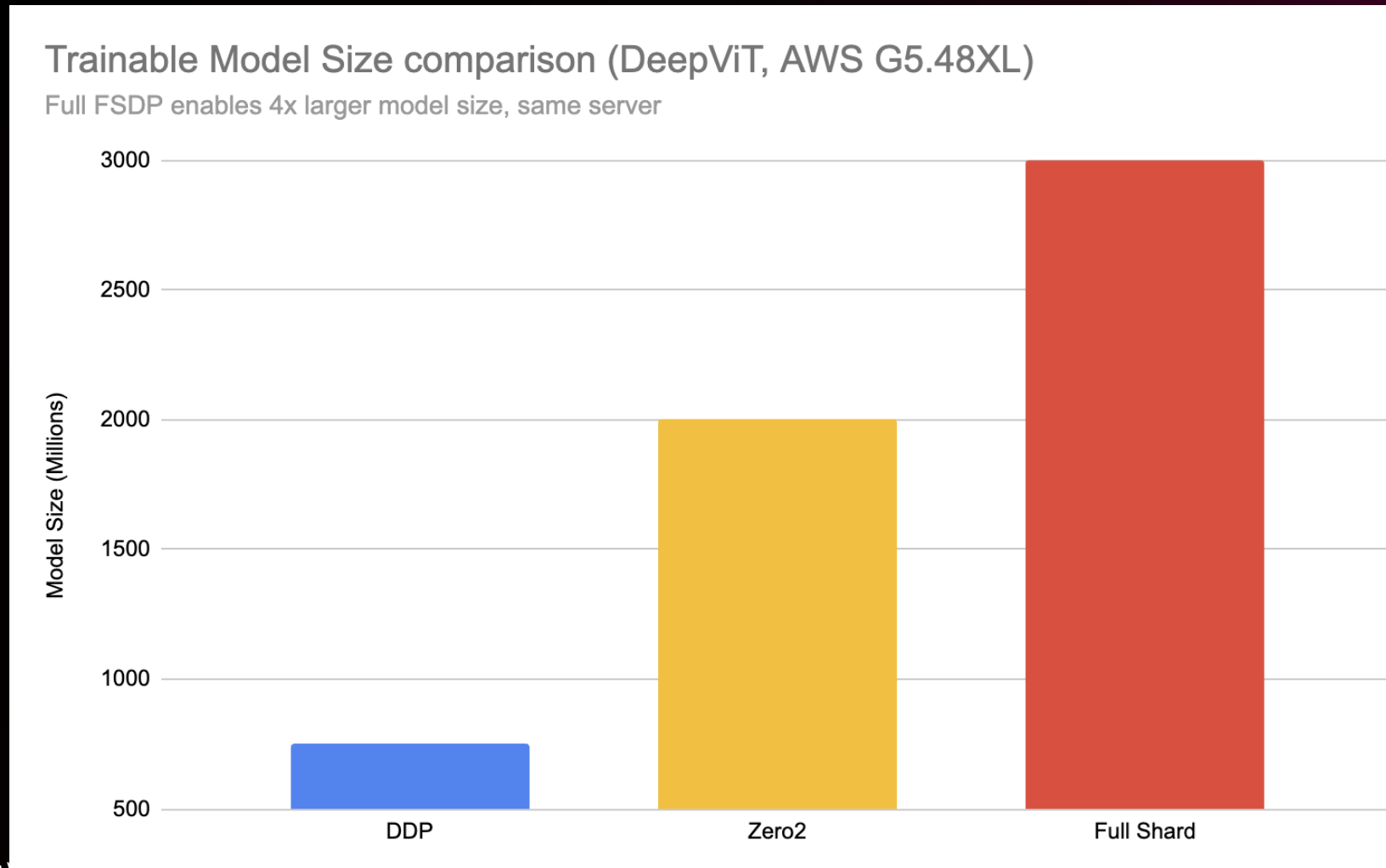
# PyTorch FSDP Sharding Strategy Details:

```
198 # import FSDP, including the class enum ShardingStrategy:
199 from torch.distributed.fsdp import (
200     FullyShardedDataParallel as FSDP,
201     ShardingStrategy,
202 )
203
204 # Three available sharding strategies – tradeoff memory size vs communication overhead:
205 ShardingStrategy.FULL_SHARD # default! Model, optimizer and gradient are all sharded (communicated) ...
206 | | | | | | | # max model size support
207 ShardingStrategy.SHARD_GRAD_OP # Zero2 mode – model parameters are not freed after forward pass,
208 | | | | | | | # reducing communication needs
209 ShardingStrategy.NO_SHARD # DDP mode – each GPU keeps a full copy of the model, optimizer and gradients
210 | | | | | | | # only grad synch needed
211
212 # Future support:
213 ShardingStrategy.HYBRID_SHARD #FSDP Full shard within each node, but No Shard (DDP) between each nodes.
214
215 # To use – just pass in desired sharding at FSDP init:
216 # ----- main FSDP init -----
217 model = FSDP(
218     model,
219     auto_wrap_policy=my_auto_wrap_policy,
220     mixed_precision=mp_policy,
221     backward_prefetch=prefetch_policy,
222     # sharding control
223     sharding_strategy=ShardingStrategy.SHARD_GRAD_OP, # Zero2 or DDP, or Full_Shard (FSDP default)
224     device_id=torch.cuda.current_device(),
225 )
```

Fast network →  
full-shard

Slower network →  
experiment with  
Zero2 and DDP

**Scaling benefits with FSDP - 4x larger models can be trained, same hardware with no other changes.**  
**(Adding in Activation Checkpointing and CPU offloading can go 21x+)**



# Fine grained Mixed Precision control via Policies

```
20  bfSixteen = MixedPrecision(  
21      # Parameter precision  
22      param_dtype=torch.bfloat16,  
23      # Gradient communication precision.  
24      reduce_dtype=torch.bfloat16,  
25      # Buffer precision.  
26      buffer_dtype=torch.bfloat16,  
27  )
```

# Fine grained Mixed Precision control via Policies

```
from torch.distributed.fsdp.fully_sharded_data_parallel import (
    FullyShardedDataParallel as FSDP,
    CPUOffload,
    BackwardPrefetch,
    MixedPrecision,
)

bf16 = MixedPrecision(
    # Param precision
    param_dtype=torch.bfloat16,
    # Gradient communication precision.
    reduce_dtype=torch.bfloat16,
)

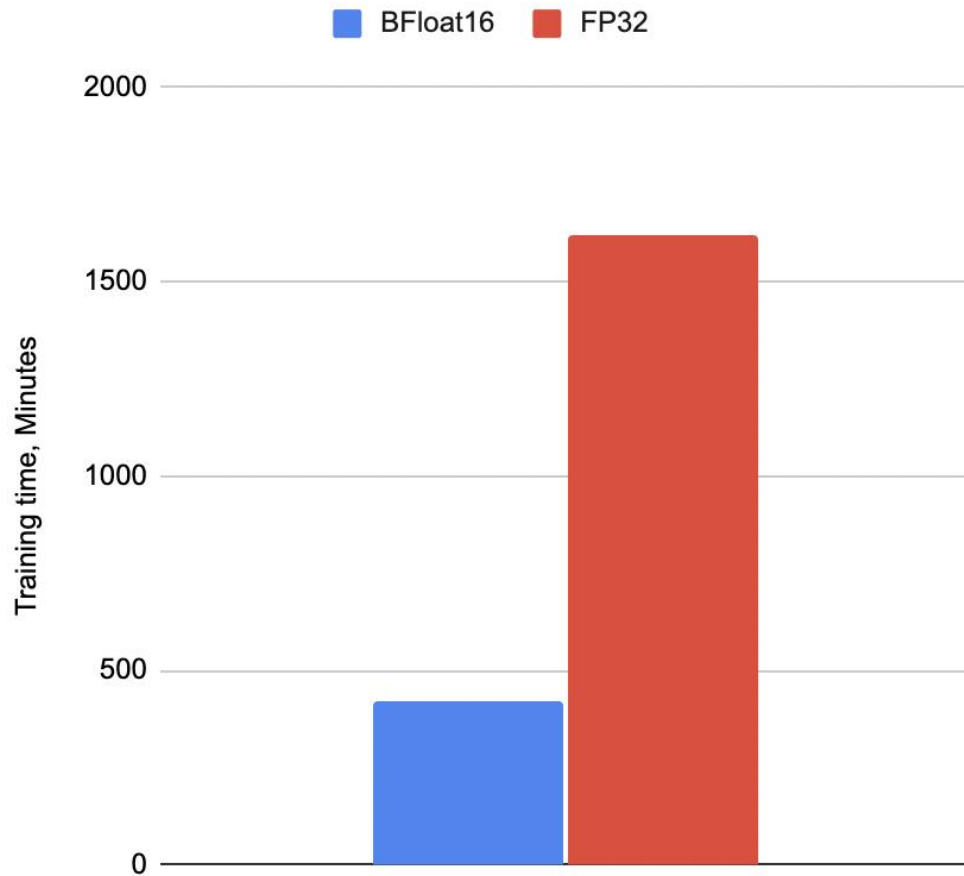
model = FSDP(
    model,
    MixedPrecision=bf16, # bf16, fp16
)
```

*BF16 is only available on Ampere GPUs. V100 may not complain but results in slowdowns.*

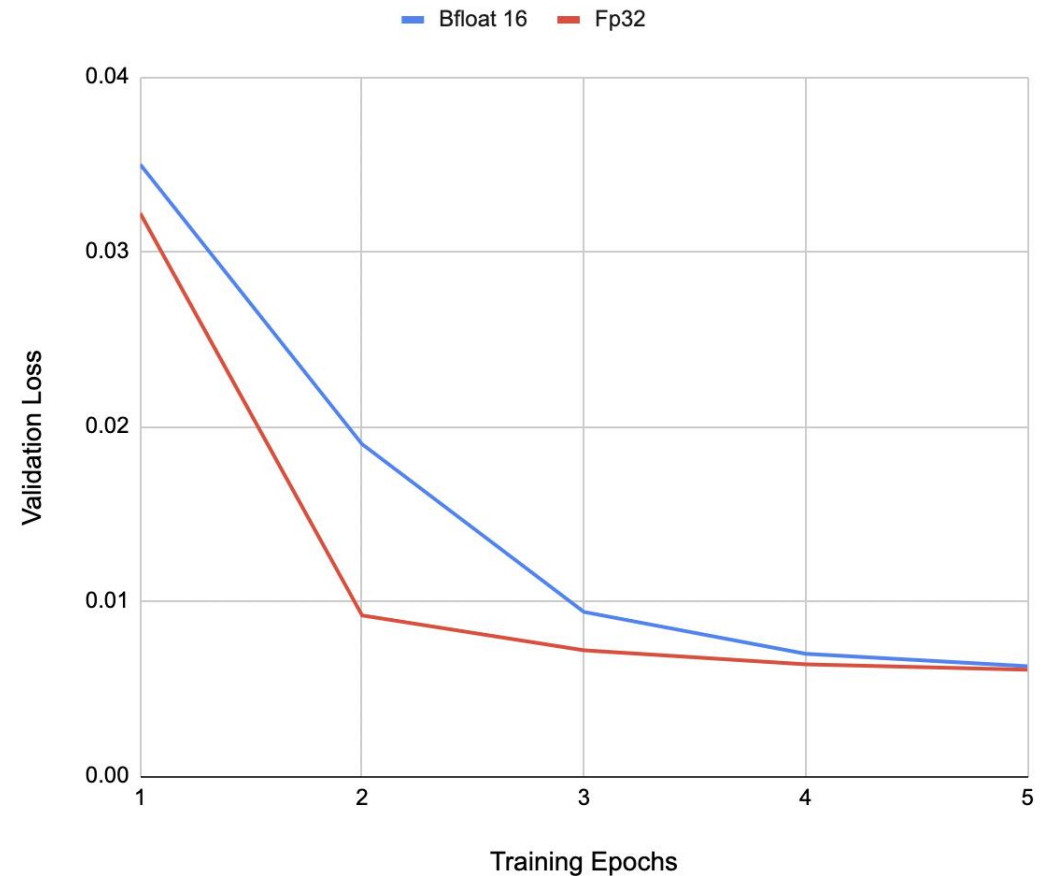
*Make sure if using FP 16 use ShardedGradScaler from FSDP*

# BFloat16 can deliver up to 4x training speed vs FP32

Total Training Time



Bfloat16 and Fp32 - Validation Loss (T5, 3 Billion Params)





# Transformer wrapping policy :

```
from torch.distributed.fsdp.wrap import (  
    transformer_auto_wrap_policy,  
)  
  
from transformers.models.t5.modeling_t5 import T5Block  
  
transformer_auto_wrapper_policy = functools.partial(  
    transformer_auto_wrap_policy,  
    transformer_layer_cls={  
        T5Block, # < ---- Your Transformer layer class  
    },  
)  
  
model = FSDP(  
    model,  
    auto_wrap_policy=transformer_auto_wrapper_policy,  
)
```

Transformer Wrapping  
policy → more  
finegrained and  
balanced FSDP units.

Significantly  
improves  
communication  
efficiency.

2x throughput  
increase

***alternative to:***

```
sized_auto_wrap_policy = functools.partial(  
    size_based_autowrap_policy, min_num_params=20000  
)  
  
#Default = FSDP puts the whole module in one FSDP unit
```

# Activation Checkpointing:

```
# verify we have FSDP activation support ready by importing:
from torch.distributed.algorithms._checkpoint.checkpoint_wrapper import (
    checkpoint_wrapper,
    CheckpointImpl,
    apply_activation_checkpointing_wrapper,
)

# first step - we have to make a check function to find what layers we want to checkpoint.
# For transformers, you'll want to use the same layers as you used for wrapping your transformer.

from transformers.models.t5.modeling_t5 import T5Block

# second create the submodule check function as a lambda:
check_fn = lambda submodule: isinstance(submodule, T5Block)

# create a non-reentrant wrapper.
# This is basically to provide some options for the checkpoint wrapper,
# and we use non-reentrant style for best performance.

non_reentrant_wrapper = partial(
    checkpoint_wrapper,
    offload_to_cpu=False,
    checkpoint_impl=CheckpointImpl.NO_REENTRANT,
)

# Important - the next step is actually to init your model with FSDP.
# Activation checkpointing is shard aware, so it must be done ** after ** FSDP init:
model = FSDP(model)

# finally, we'll apply the checkpoint wrapper, and submodule check lambda to your sharded model
# to complete the activation checkpointing process:

apply_activation_checkpointing_wrapper(
    model, checkpoint_wrapper_fn=non_reentrant_wrapper, check_fn=check_fn
)
```

Generally save 30% memory

Reinvest in batch size →  
~7x throughput on DeepVit

FSDP Activation  
checkpointing is shard  
aware → use after FSDP  
init.

# Backward Prefetch:

```
# import FSDP and BackwardPrefetch class
from torch.distributed.fsd import (
    FullyShardedDataParallel as FSDP,
    BackwardPrefetch,

# None (i.e. don't pass anything) = summon for next FSDP unit comes after the all_reduce for current layer gradients

# BackwardPrefetch.BACKWARD_POST = prefetch at the end of current FSDP unit computation, after params are dropped

# BackwardPrefetch.BACKWARD_PRE = prefetch at start of current FSDP unit computation (earliest, adds to peak memory)

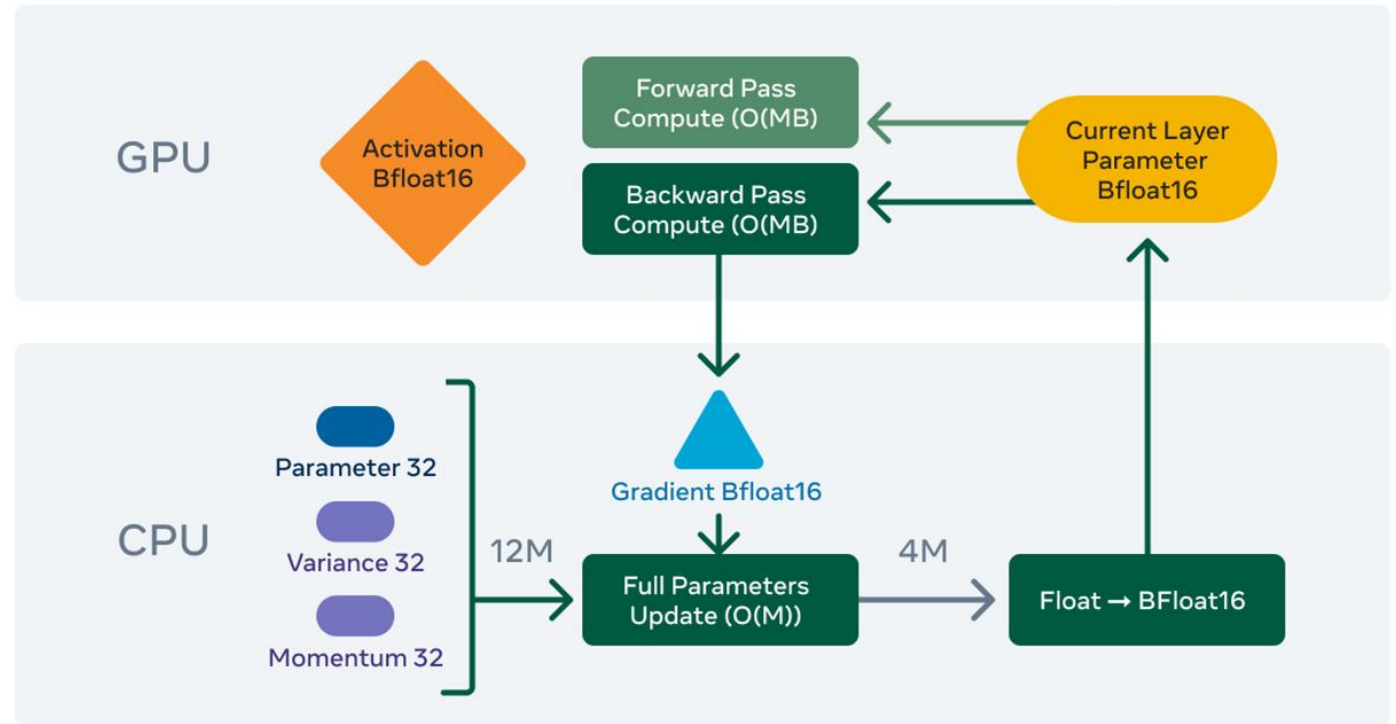
prefetch_policy = BackwardPrefetch.BACKWARD_PRE # BackwardPrefetch.BACKWARD_POST or None

model = FSDP(
    model,
    backward_prefetch=prefetch_policy,
)
```

# CPU Offloading

Leverage CPU Memory to house 'non-active' parameters

Allows expansion beyond the total sum of GPU memory.



# Saving your model - leverages CPU memory

```
if cfg.save_model:
    # assembling model on rank0 and stream it to cpu to avoid OOM
    save_policy = FullStateDictConfig(offload_to_cpu=True, rank0_only=True)

    with FSDP.state_dict_type(
        model, StateDictType.FULL_STATE_DICT, save_policy
    ):
        cpu_state = model.state_dict()

    if rank == 0:
        print(f"--> saving model ...")
        currEpoch = "-" + str(epoch) + "-train.pt"
        model_save_name = save_name + currEpoch

        torch.save(cpu_state, model_save_name)

        print(f"--> saved {model_save_name} to disk")
```

Avoid OOMs on rank0  
GPU

Leverage CPU memory  
→ work for very  
large models ~ 20B

# Example of auto-wrapping (just print the model to view!)

```
1  model = t5-large, sharded with 2000000 parameters per fsdp_unit
2
3
4  --> t5-large has 737.668096 Million params
5
6  model wrapping =
7  FullyShardedDataParallel(
8      (_fsdp_wrapped_module): FlattenParamsWrapper(
9          (_fpw_module): T5ForConditionalGeneration(
10             (shared): Embedding(32128, 1024)
11             (encoder): T5Stack(
12                 (embed_tokens): Embedding(32128, 1024)
13                 (block): ModuleList(
14                     (0): T5Block(
15                         (layer): ModuleList(
16                             (0): T5LayerSelfAttention(
17                                 (SelfAttention): T5Attention(
18                                     (q): Linear(in_features=1024, out_features=1024, bias=False)
19                                     (k): Linear(in_features=1024, out_features=1024, bias=False)
20                                     (v): Linear(in_features=1024, out_features=1024, bias=False)
21                                     (o): Linear(in_features=1024, out_features=1024, bias=False)
```

# PyTorch FSDP implementation:

Highlights of the major sub-sections:

```
208 # ----- main FSDP init -----
209 model = FSDP(
210     model,
211     auto_wrap_policy=my_auto_wrap_policy,
212     mixed_precision=mp_policy,
213     backward_prefetch=prefetch_policy,
214     device_id=torch.cuda.current_device(),
215     sharding_strategy=ShardingStrategy.FULL_SHARD, # Sharding options ...
216     |         |         |         |         |         |         |         |         |         |         | # SHARD_GRAD_OP = Zero2,
217     |         |         |         |         |         |         |         |         |         |         | # NO_SHARD = DDP
218     cpu_offload=cpu_policy,
219     forward_prefetch=True,
220 )
221
222 if cfg.fsdp_activation_checkpointing:
223     fsdp_checkpointing(model)
224     if local_rank==0:
225         print(f"--> FSDP activation checkpointing in use")
226
```



# Some Best practices:

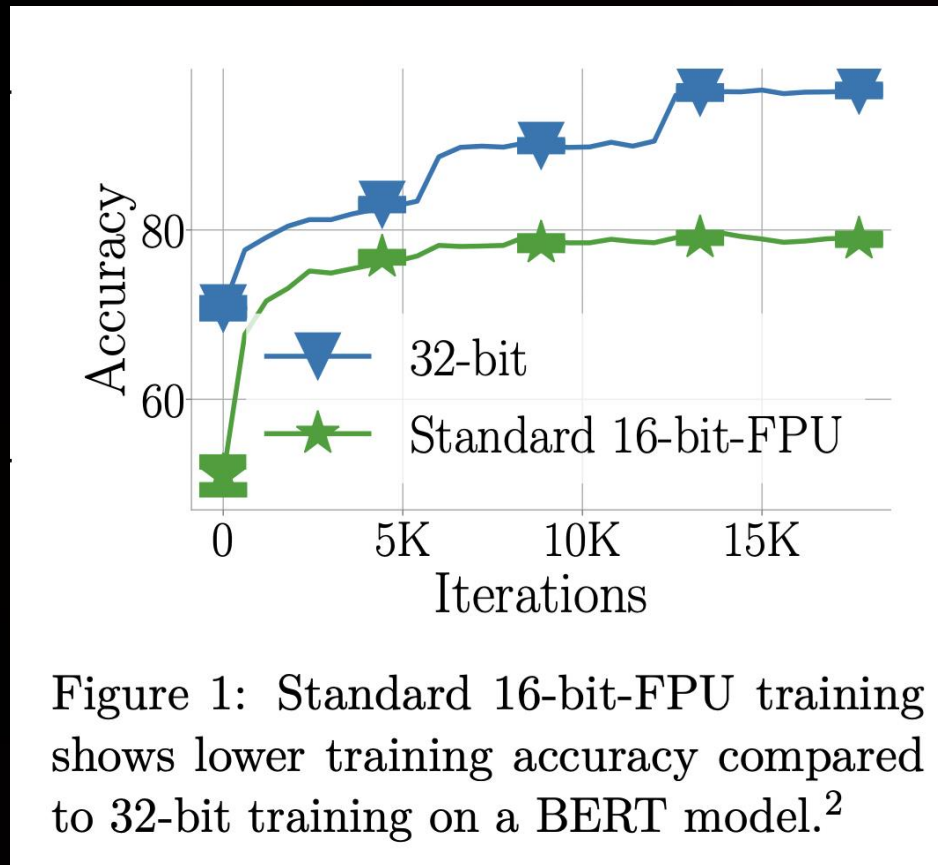
- 1 - BFloat16 increases training speed, 35% to 4x...and 32% memory reduction.
- 2 - Backward pre fetch via BACKWARD\_PRE ...2 - 10% training speedup.
- 3 - Use the rank\_0 cpu saving to avoid any OOM issues during large model saving.
- 4 - Use activation checkpointing - this frees up large amounts of memory during training.
- 5 - Proper wrapping - For Transformers use Transformer wrapping policy

# AnyPrecision Optimizer - run FSDP in pure BF16

Q - Why only mixed precision? Why not 100% BF16?

A - Pure BF16 doesn't work...

*Problem = weight stagnation due to small updates being lost*



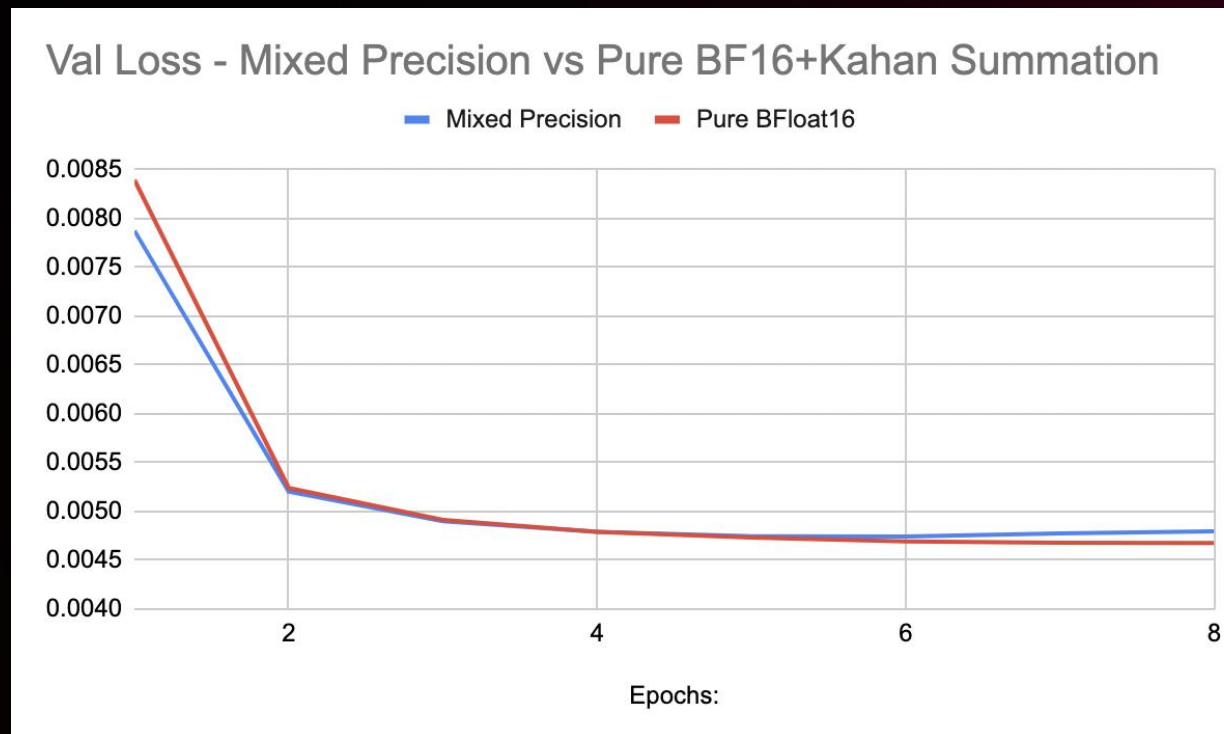
Paper - <https://arxiv.org/abs/2010.06192> (Revisiting BFloat16 training)

# AnyPrecision Optimizer - run FSDP in pure BF16

Solution - AnyPrecision uses Kahan summation in the optimizer for weight updates.

Significant memory, speed improvements with pure BF16, while matching or exceeding FP32. (~ 48% GPU memory reductions in initial testing vs FP32).

Drop in replacement for AdamW



Available in Torchdistx

<https://github.com/pytorch/torchdistx>



© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# Working with extremely large models

PyTorch has added several features to allow for instantiating models without having to load the actual weights.

This allows extremely large models (think 175B+ that would OOM on CPU) to be loaded, inspected and ultimately sharded as only the shapes, not the weights, are loaded.

This includes:

Meta device

Deferred initialization

Fake Tensor



# META DEVICE

Instantiate Module/  
tensor onto a meta  
device

It has tensor shape,  
does not allocate  
any storage

```
# 1. Initialize module on the meta device; all torch.nn.init ops have  
# no-op behavior on the meta device.  
m = nn.Linear(10, 5, device='meta')  
  
# 2. Materialize an uninitialized (empty) form of the module on the CPU device.  
# The result of this is a module instance with uninitialized parameters.  
m.to_empty(device='cpu')  
  
meta_tensor = torch.randn(100000, 100000, device="meta")
```

[https://pytorch.org/tutorials/prototype/skip\\_param\\_init.html#implementation-details](https://pytorch.org/tutorials/prototype/skip_param_init.html#implementation-details)

# Deferred initialization

Deferred module init  
has

`deferred_init()` ,  
`materialize_module()`  
,  
`materialize_tensor()`.

`deferred_init()` ,  
construct model  
without allocating  
storage for their  
tensors

`materialize_module()`,  
`materialize_tensor()`,  
fully or partially  
materialize modules.

```
import torch

from torchdistx.deferred_init import deferred_init, materialize_module

class MyModule(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.param = torch.nn.Parameter(torch.ones([3], device="cpu"))

m = deferred_init(MyModule):
m.param
#Parameter containing:
#tensor(..., device='cpu', size=(10, 10), requires_grad=True, fake=True)

materialize_module(m)
m.param
#Parameter containing:
#tensor([1., 1., 1.], requires_grad=True)
```

*Available in Torchdistx*

<https://github.com/pytorch/torchdistx>



# Fake Tensor

Similar to meta\_tensors,  
does not contain any  
data

Fake tensors act as if  
allocated to a real device

It has been meant  
mostly to use a building  
block for deferred  
module init.

It can be useful to load  
the model without  
initializing it with  
data.

```
import torch
from torchdistx.fake import fake_mode
# Meta tensors are always "allocated" on the `meta` device.
a = torch.ones([10], device="meta")
a
#tensor(..., device='meta', size(10,))
a.device
#device(type='meta')

# Fake tensors are always "allocated" on the specified device.
with fake_mode():
    b = torch.ones([10])

#tensor(..., size(10,), fake=True)

b.device
#device(type='cpu')
```

```
import torch

from transformers import BlenderbotModel, BlenderbotConfig

from torchdistx.fake import fake_mode

# Instantiate Blenderbot on a personal laptop with 8GB RAM.
with fake_mode():
    m = BlenderbotModel(BlenderbotConfig())

# Check out the model layers and their parameters.
m
#BlenderbotModel(...)
```

*Available in Torchdistx*

<https://github.com/pytorch/torchdistx>

[https://pytorch.org/torchdistx/latest/fake\\_tensor.html](https://pytorch.org/torchdistx/latest/fake_tensor.html)

[https://pytorch.org/torchdistx/latest/fake\\_tensor\\_and\\_deferred\\_init.html](https://pytorch.org/torchdistx/latest/fake_tensor_and_deferred_init.html)





# Fake Tensor

Meta device work great  
for skipping initialization

It might not work in all  
case for materilization  
when working in  
deferred initialization  
context.

some of pytorch  
functions like `zero_like`,  
refers to `src.device`

Fake tensor can solve  
this issue.

```
class MyModule(Module):
    def __init__(self):
        super().__init__()
        self.buf1 = torch.ones([3], device="cpu")
        self.buf2 = torch.zeros_like(self.buf1)

my_module = deferred_init(MyModule)

materialize_tensor(my_module.buf1)
#tensor([1., 1., 1.])
materialize_tensor(my_module.buf2)
#tensor(..., device='meta')
```

*Available in Torchdistx*

<https://github.com/pytorch/torchdistx>

[https://pytorch.org/torchdistx/latest/fake\\_tensor.html](https://pytorch.org/torchdistx/latest/fake_tensor.html)

[https://pytorch.org/torchdistx/latest/fake\\_tensor\\_and\\_deferred\\_init.html](https://pytorch.org/torchdistx/latest/fake_tensor_and_deferred_init.html)



# TorchSnapshot

Torchsnapshot offers highly optimized checkpointing ~2x faster than save()

5x faster than saving a GPU on S3 compared to save()+fsspec

Adaptive to the host memory – avoid OOMs during checkpointing

Can save/load GPU model using temporary memory a fraction of the size of largest tensor in the model

Load 20GB from storage to GPU with 100MB temp memory

Supports S3, GCS

*Available in Torchsnapshot*

<https://pytorch.org/torchsnapshot/>



© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

```
app_state = {"model": model, "optimizer": optimizer}

from torchsnapshot import StateDict

extra_state = StateDict(iterations=0)
app_state = {"model": model, "optimizer": optimizer, "extra_state": extra_state}

from torchsnapshot import Snapshot

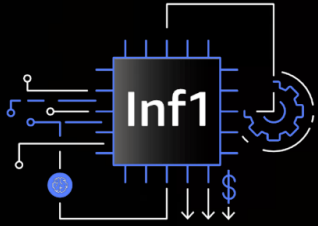
snapshot = Snapshot.take(path="/path/to/my/snapshot", app_state=app_state)

snapshot.restore(app_state=app_state)
```

# Inference at scale



# Amazon EC2 accelerated compute for ML inference



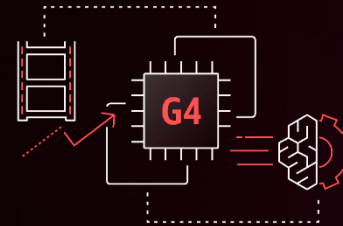
## Inf1: Custom ML acceleration

- Low cost per inference in the cloud
- Up to 2,000 TOPs with AWS-designed Inferentia accelerators
- Designed for high throughput and low latency



## G5: GPU compute instance

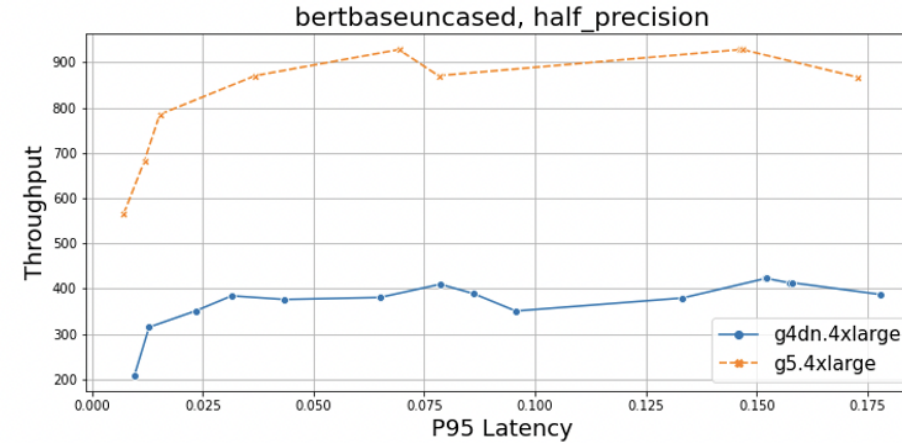
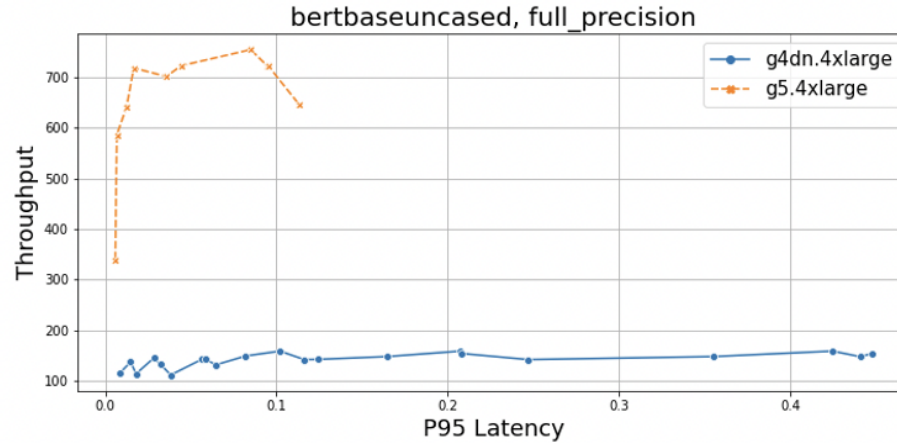
- 3–4x ML inference performance compared to previous generations
- 8x NVIDIA A10G 24 GB, 100 Gb/s



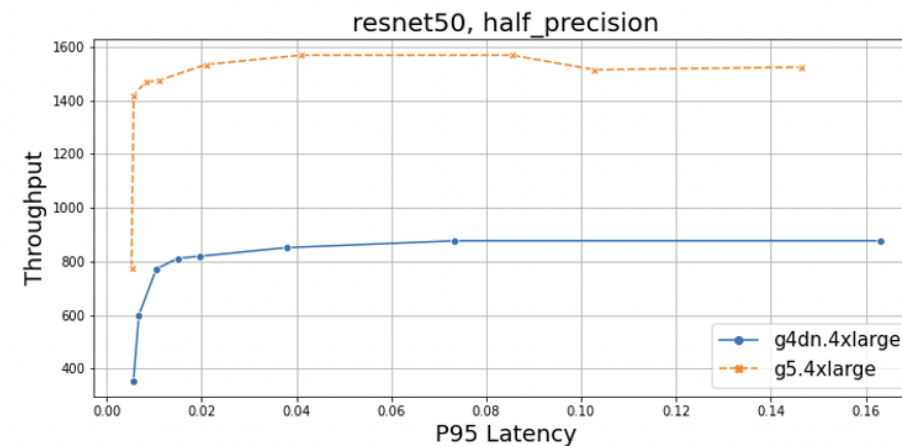
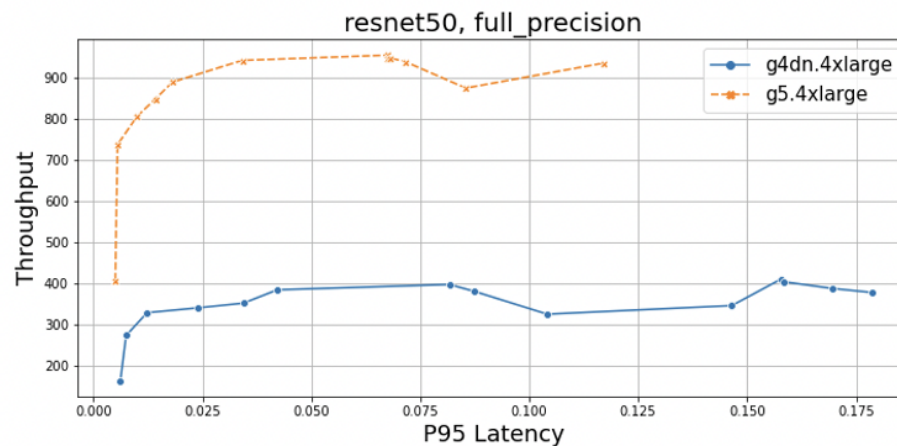
## G4dn: GPU compute instance

- Up to 1,030 TOPs of compute with 8x NVIDIA T4 GPUs
- Low-cost GPU-based instances

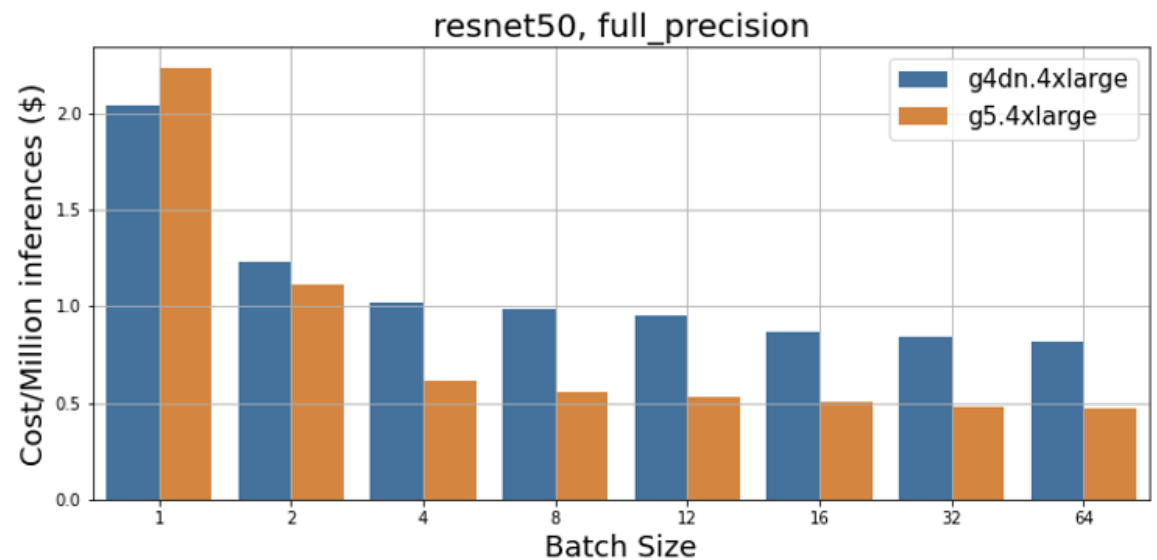
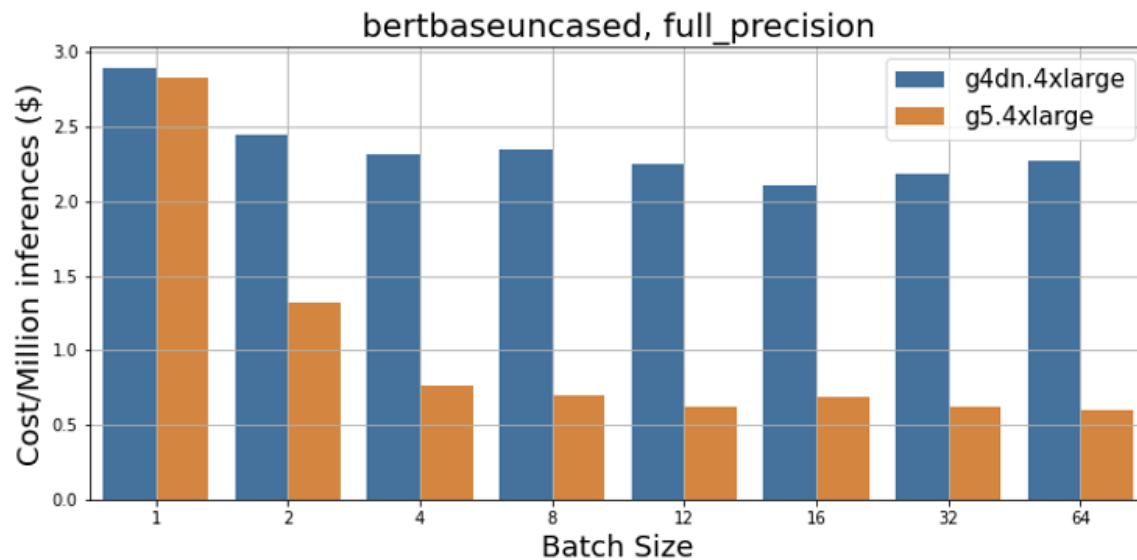
# G5 vs. G4dn



The following graphs compare throughput and P95 latency at full and half precision for ResNet50.



# Cost performance

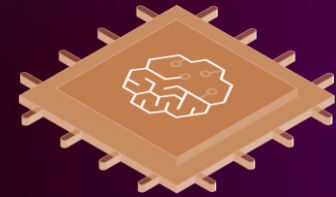


**With G5 instances, you can achieve consistently lower cost-per-inference compared to G4dn instances**

# Amazon EC2 Inf1 instances

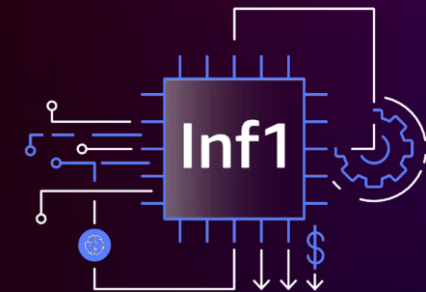
HIGH-PERFORMANCE, LOW-COST MACHINE LEARNING INFERENCE

- Featuring **AWS Inferentia**, the first ML chip designed by Amazon
- **Low cost in the cloud** for running deep-learning models, up to 70% lower cost than GPU instances
- Up to 2.3x higher throughput than GPU-based instances
- Seamless software **integration with ML frameworks** like TensorFlow, PyTorch, and MXNet for getting started quickly and with minimal code changes
- Get started using DLC, DL AMIs, Amazon EKS, Amazon ECS, or Amazon SageMaker



AWS Inferentia

High-performance machine learning inference chip, custom designed by AWS



EC2 Inf1 instances

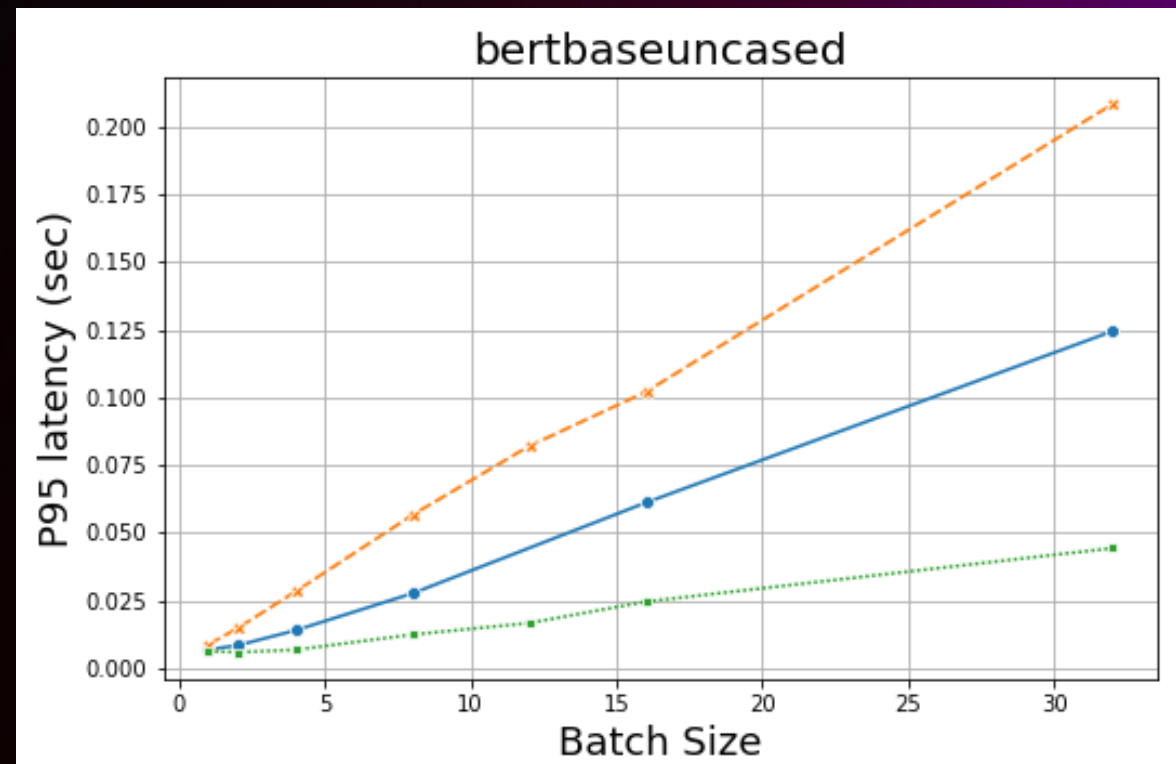
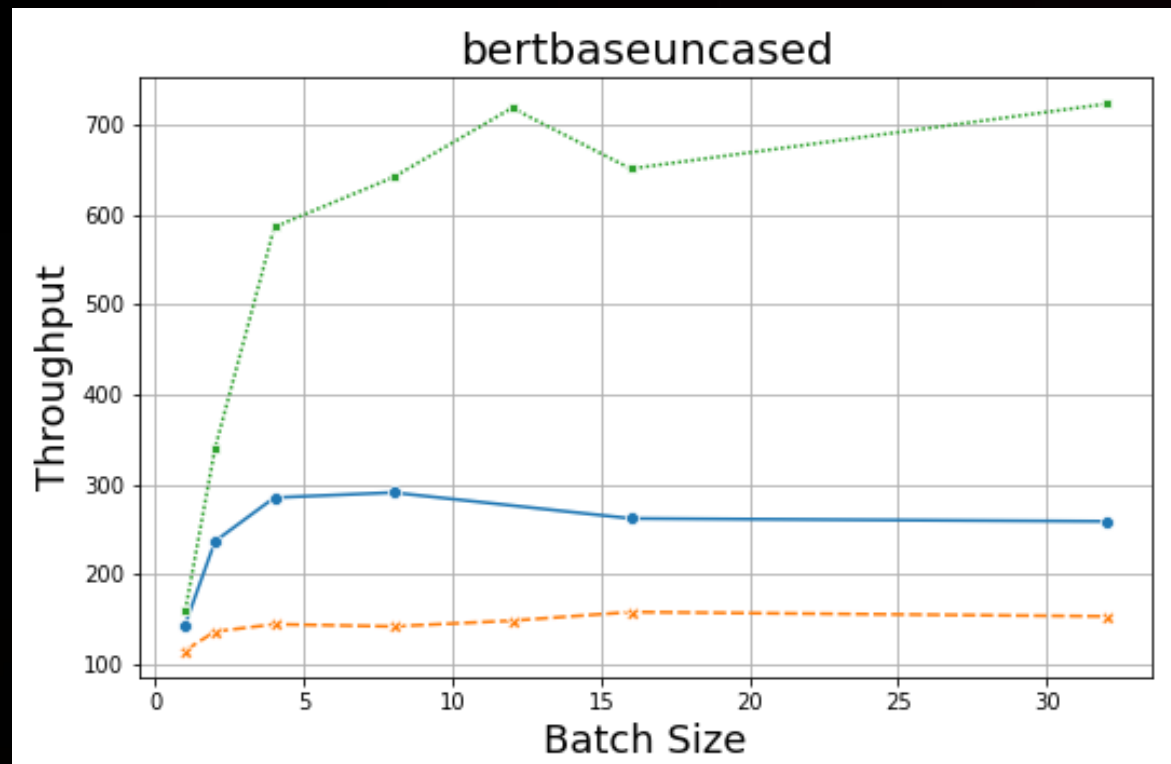
Fast and low-cost inference in the cloud



# Inf1 vs. G5 and G4dn

NATURAL LANGUAGE PROCESSING

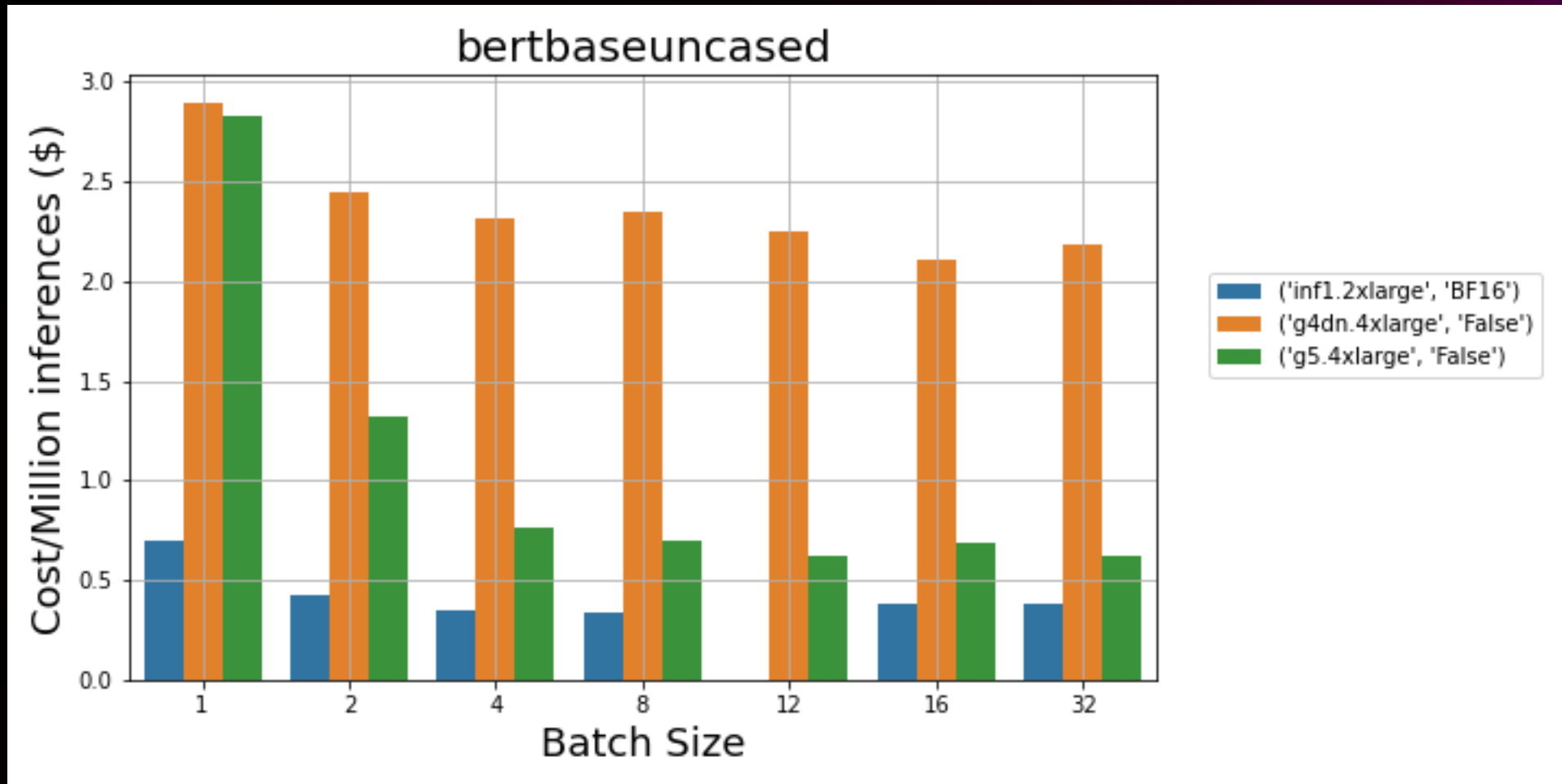
G5.4xlarge  
Inf1.2xlarge  
G4dn.4xlarge



# Inf1 vs. G5 and G4dn

NATURAL LANGUAGE PROCESSING

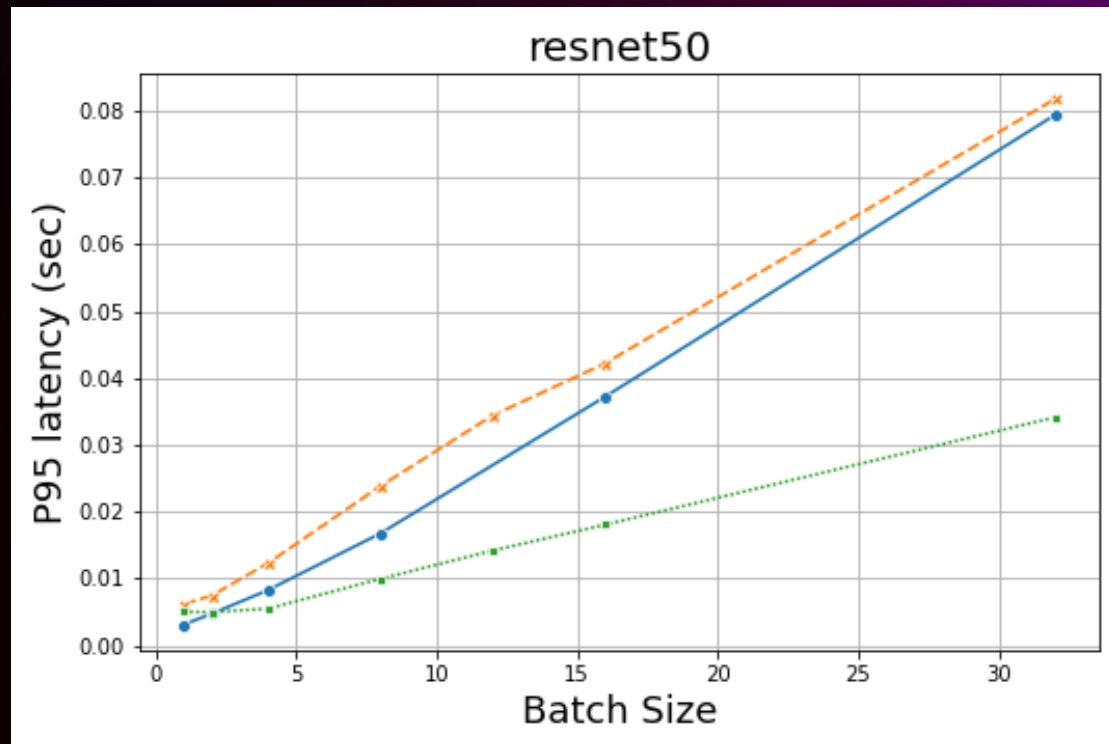
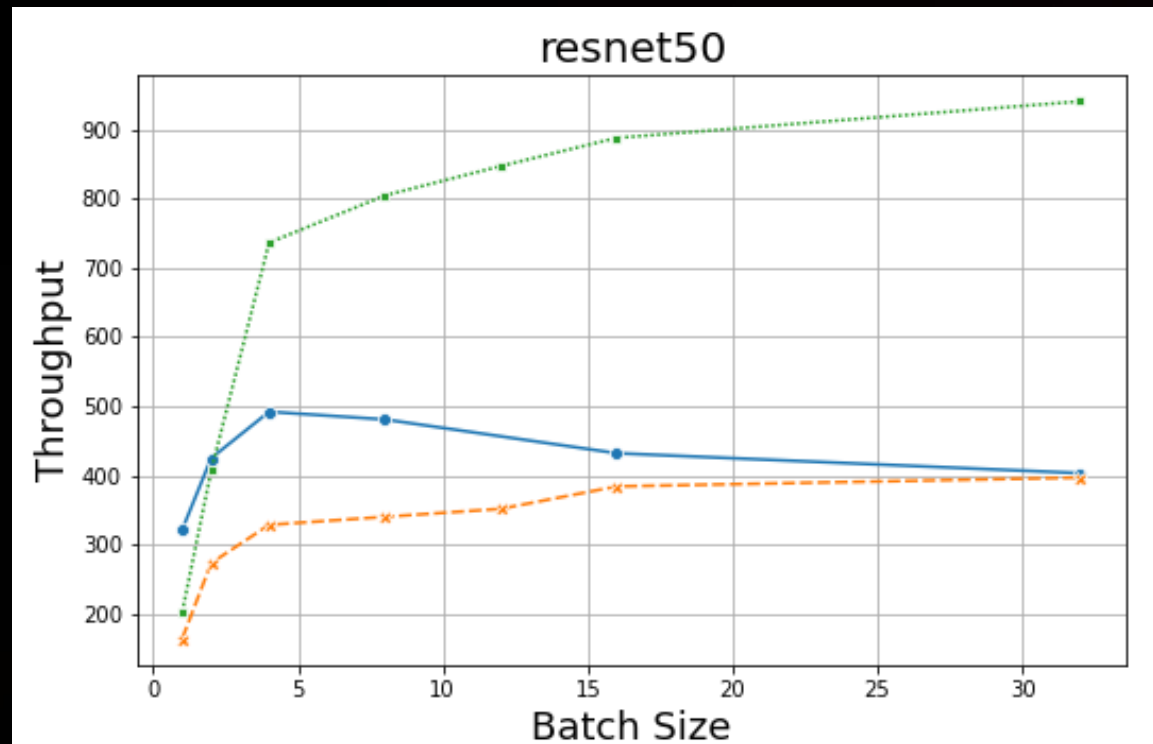
G5.4xlarge  
Inf1.2xlarge  
G4dn.4xlarge



# Inf1 vs. G5 and G4dn

COMPUTER VISION

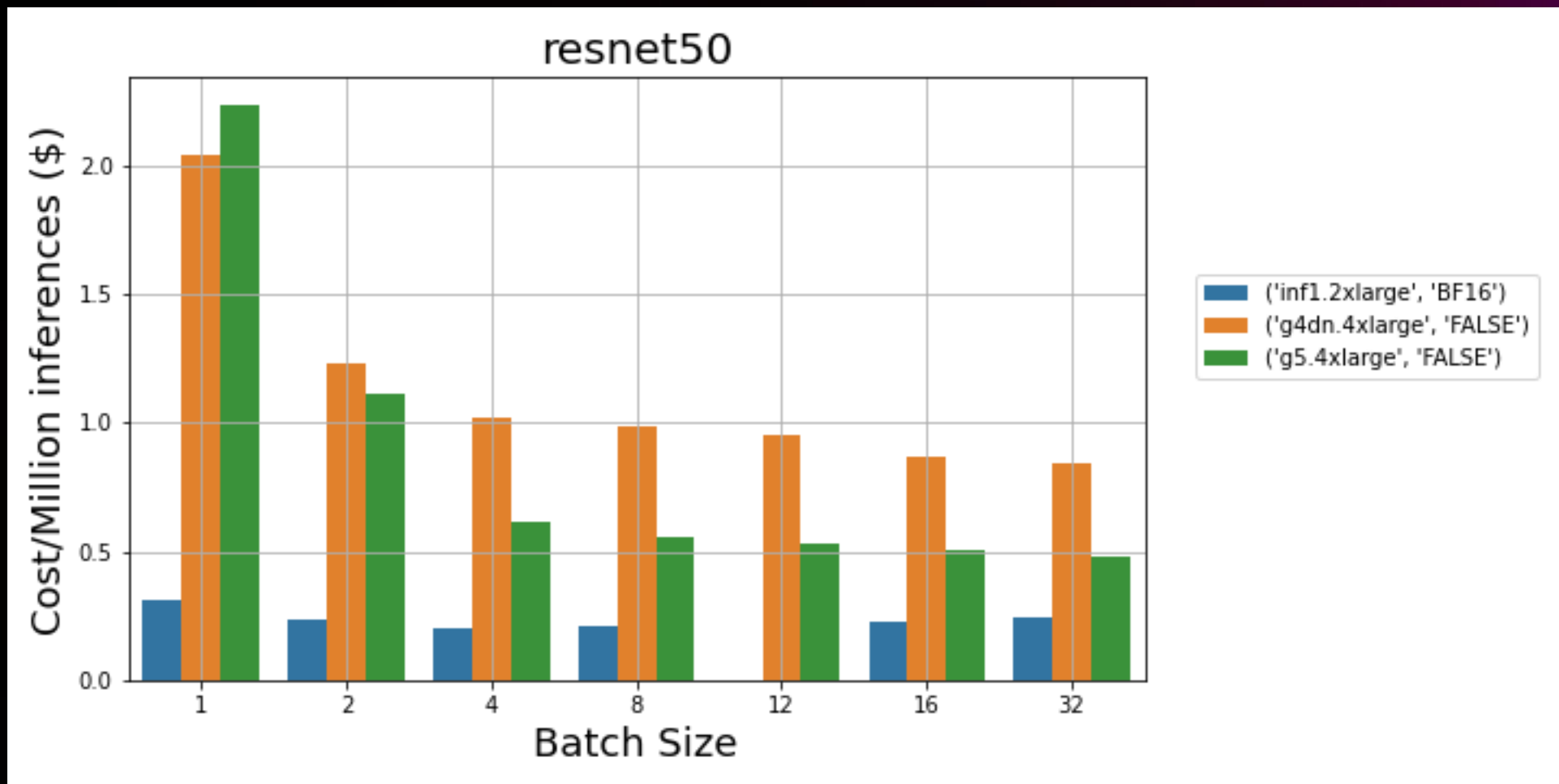
G5.4xlarge  
Inf1.2xlarge  
G4dn.4xlarge



# Inf1 vs. G5 and G4dn

COMPUTER VISION

G5.4xlarge  
Inf1.2xlarge  
G4dn.4xlarge



# Proof of concept

BRINGING THE CREATIVE POTENTIAL OF AI TO EVERYONE

## Startup requirements:

Build personalized creative AI models for mentorship, AMA, interactive NFTs, and more

9–10 models per user

Mostly NLP models

G4dn instances

Need to load, unload, and infer models in parallel

## Challenges:

Low GPU utilization

High cost

High latency due to big models

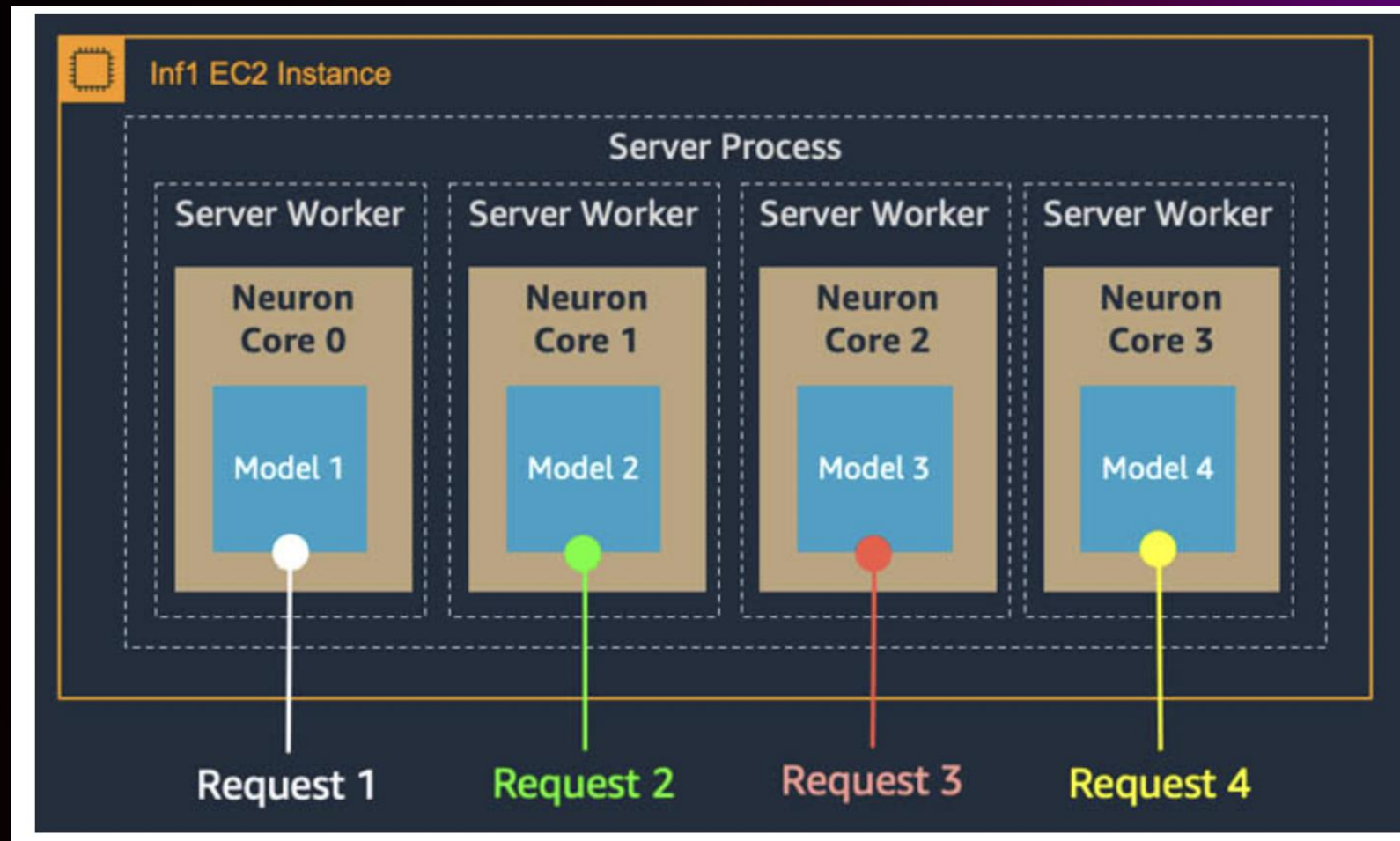
Proposed a solution with AWS Inferentia and FastAPI

## Impact:

Customer is satisfied with the approach, considering implementing in production

# Loading models in parallel

- One python process can be tied to one Neuron Core with `NEURON_RT_VISIBLE_CORES` environment variable.
- This allows one model server per user or a set of users on one Neuron Core.
- With this approach, we can load, unload and infer models in parallel while maintaining high utilization of all 4 cores on an AWS Inferentia chip.



# Amazon EKS



# aws-do-eks framework

Use aws-do-eks framework (<https://github.com/aws-samples/aws-do-eks>)

Set up desired cluster using a conf file

Creates a docker container with all the tools needed to manage the cluster

Automatically sets up necessary IAM roles, VPC, security groups, autoscaling groups, etc.

Provides various tools for setting up persistent volumes, GPU monitoring, launching model training jobs, etc.



# How to set up an Amazon EKS cluster

## Walkthrough config

### Steps:

Fill out config

Build and run in a container

eks-create.sh: Create auto scaling groups

eks-scale.sh: Scale nodes to desired capacity

IAM roles are created for each node group but additional roles, like access to Amazon S3 or Amazon CloudWatch, may need to be added

Spot Instances

Custom AMIs

After a cluster is created:

- New nodegroups can be added after a cluster is created with eks-nodegroup-create.sh
- Nodegroups can be scaled via eks-scale.sh or via console.

### Group size

Specify the size of the Auto Scaling group by changing the desired capacity. You can also specify minimum and maximum capacity limits. Your desired capacity must be within the limit range.

Desired capacity

Minimum capacity

Maximum capacity

Cancel Update

# Shared file system

- Need a shared file system that all nodes in the cluster can access
- File system should be elastic and performant for large data



## Amazon Elastic File System (EFS)

- `efs-create.sh` – Creates security group and a new Amazon EFS volume
- `deploy.sh`
  - Gets existing file system ID
  - Deploys Amazon EFS CSI driver
  - If multiple volumes exist, takes the first one given by `aws efs describe-file-systems`
  - Deploys file system on Amazon EKS



## Amazon FSx for Lustre

- Can be deployed to 1 AZ
- Update `fsx.conf` file
- Run `deploy.sh` – This will create the security group that you need for FSx and also create the storage class
- Create the FSx filesystem from AWS console
- Update `fsx-pvc-static.yaml` with the information from FSx console
- `Kubectl apply fsx-pvc-static.yaml`.

<https://github.com/aws-samples/aws-do-eks/tree/main/Container-Root/eks/deployment/csi>



# Serve 3,000 deep-learning models on Amazon EKS with AWS Inferentia for under \$50 an hour

Exp. #	Instances (num x type)	Models (num)	Sequential Response (ms)	Random Response (ms)	Throughput (req/s)	Latency with Load P90 (ms)	On-Demand Cost (\$/hr)
1	3 x inf1.6xl	144	21 – 32	21 – 30	142	57	3.54
2	5 x inf1.6xl	240	23 – 35	21 – 35	173	56	5.9
3	21 x inf1.6xl	1008	23 – 39	23 – 35	218	24	24.78
4	32 x inf1.6xl	1536	26 – 33	25 – 37	217	23	37.76
5	4 x g4dn.12xl	288	27 – 34	28 – 37	178	30	15.64
6	14 x g4dn.12xl	1008	26 – 35	31 – 41	<b>154</b>	30	54.76
7	32 x inf1.6xl + 21 x g4dn.12xl	3048	27 – 35	24 – 45	248	28	119.91
8	40 x inf1.6xl	3002	24 – 31	25 – 38	<b>1536</b>	33	47.2
9	<b>40 x inf1.6xl</b> <i>(With DNS Caching)</i>	<b>3040</b>	<b>24 – 31</b>	<b>25 – 38</b>	<b>6230</b>	<b>66</b>	<b>47.2</b>

# Setting up for success at AWS

**Mission:** Meet customers where they are, help them adopt and scale ML workloads on AWS with Amazon EC2, key services, and open-source tools and frameworks (PyTorch, TensorFlow)

## **Architecting scalable solutions**

using Amazon EC2, Amazon EKS, Amazon ECS, AWS Batch, Spot, KubeFlow, Ray, PyTorch, TensorFlow, etc.

## **EC2 instance selection**

via benchmarking for specific models and workloads (e.g., P4d for training, G4dn, G5, Inf1 for inference).

## **Develop proof-of-concepts**

to show the art of the possible, unblock technical challenges, and advise on best practices. **Proof of concept credits** available.

## **Guidance on PyTorch**

adoption, scaling, roadmap dives deep in three-way discussions with **Meta** (if needed).

**Areas of interest:** NLP, CV, distributed training, ML inference, MLOPs

“Meta and AWS will jointly help enterprises use PyTorch on AWS to bring deep learning models from research into production faster and easier.”


– [Meta/AWS Press Release, December 2021](#)



# Thank you!

Hamid Shojanazeri

 hamidnazeri@meta.com

 @Nazeri2010

Ankur Srivastava

 awsankur@amazon.com

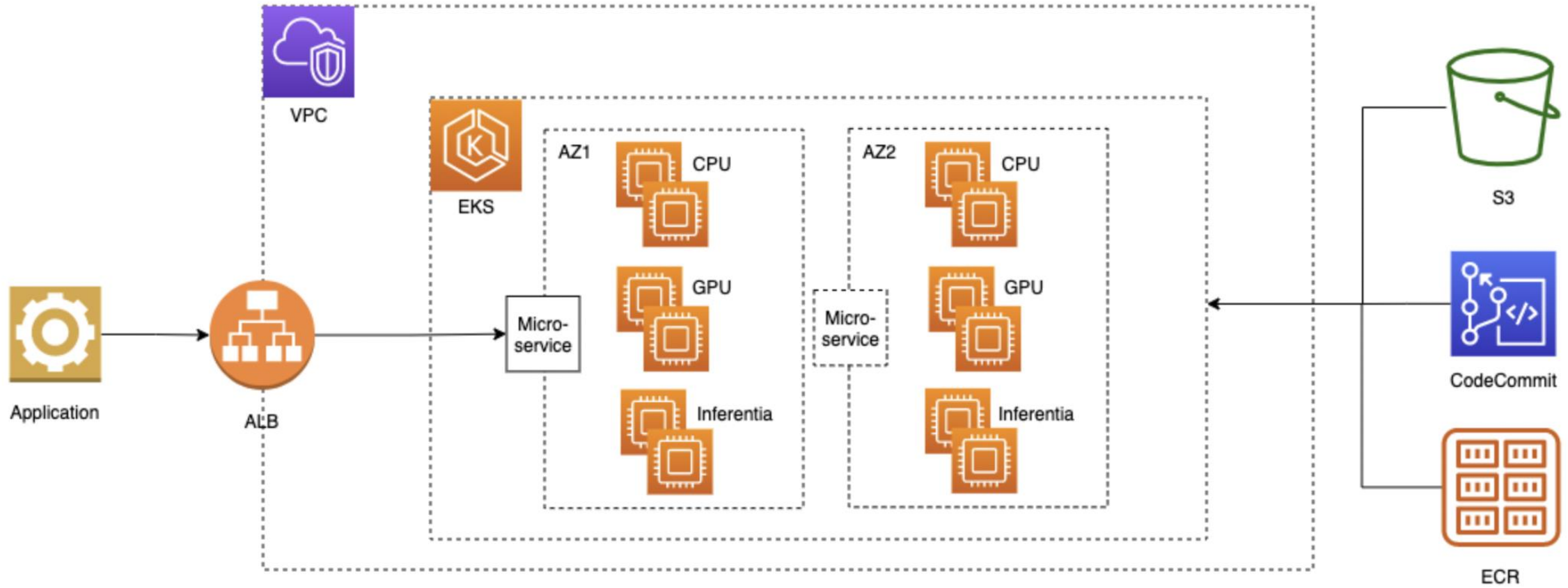
 @ankur\_rice



Please complete the session survey in the **mobile app**



# Architecture



# Moving data from Amazon S3 to Amazon EFS

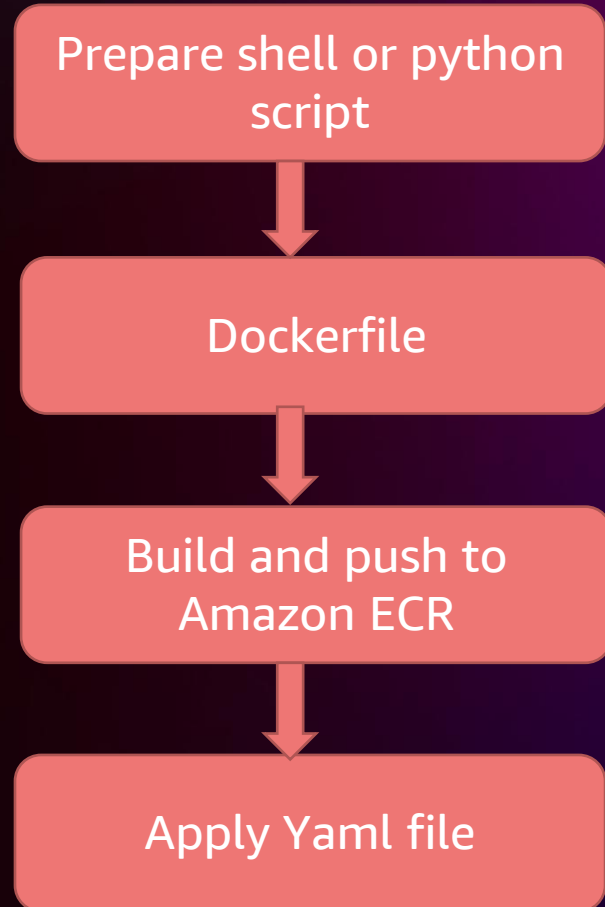
Once a cluster and a file system is created, you might need to:

Move compressed data from Amazon S3 to Amazon EFS

Uncompress data in Amazon EFS

Preprocess data with shell scripts

Or, preprocess data with python code

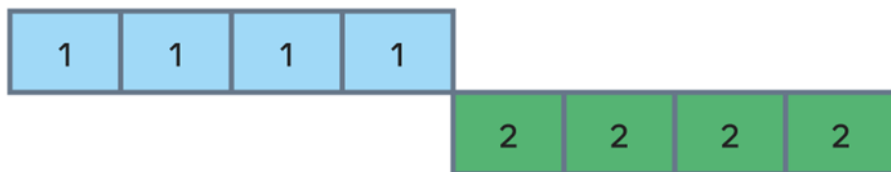


# FSDP Deep Dive: parameter level

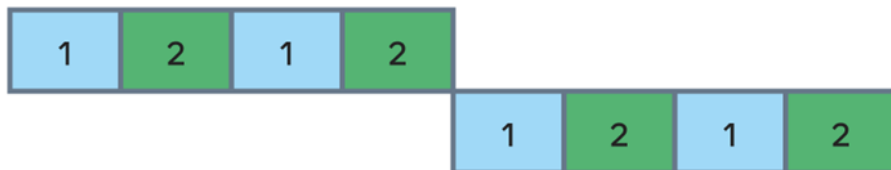
Flattened params from model - 8 total



Typical ownership



Sharding strategy = FSDP Unit of 4 params, with sequential shard ownership by ranks (GPU)

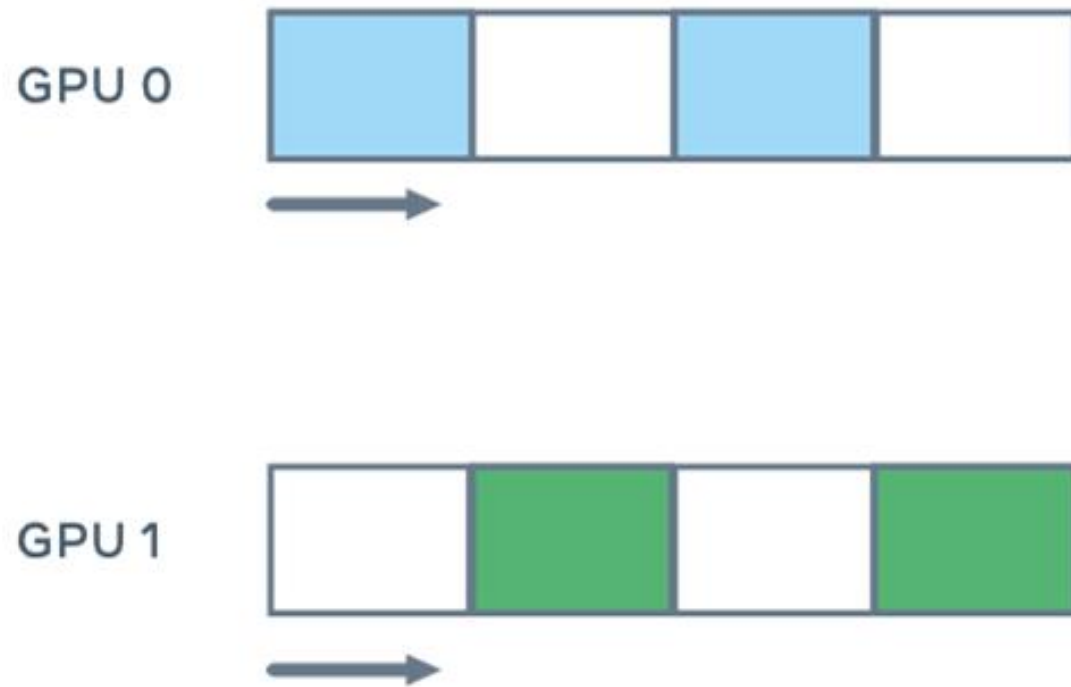




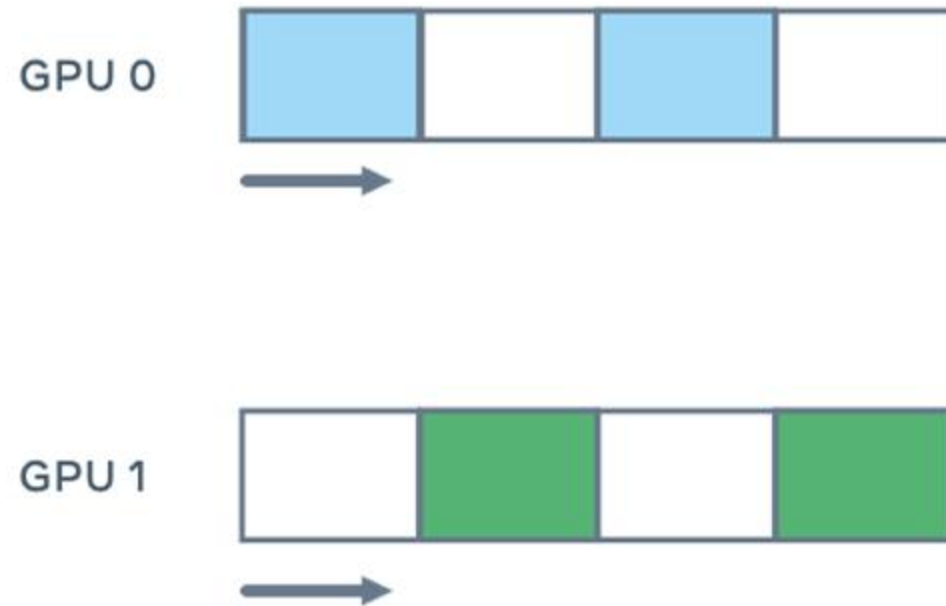
Flattened params from model - 8 total



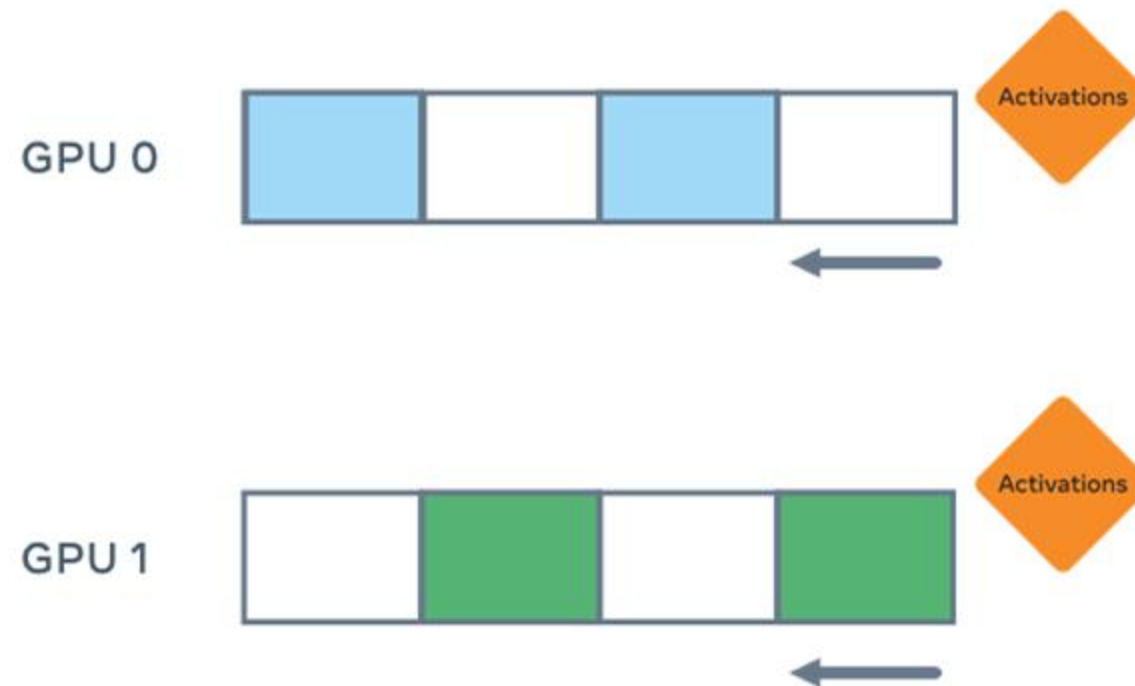
Forward pass on first FSDP Unit. Begin All-Gather.



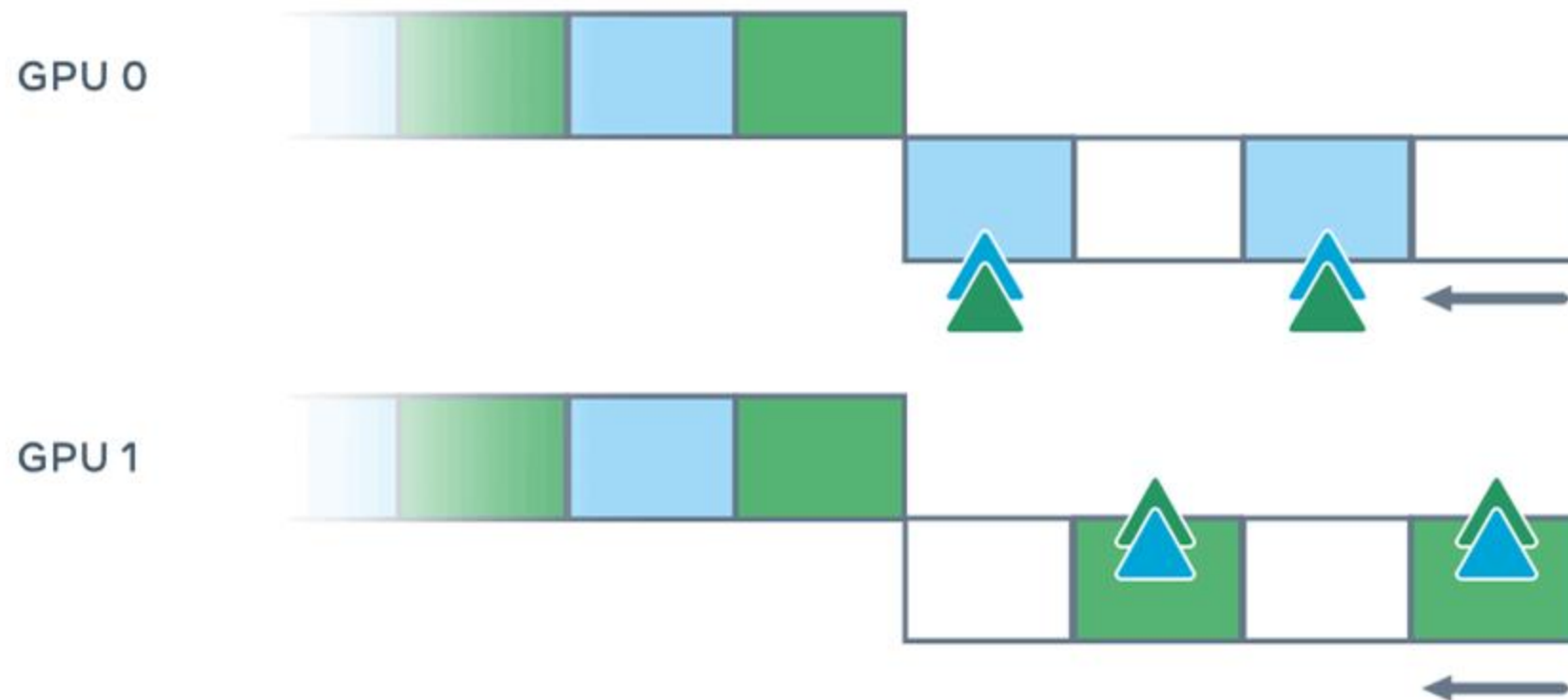
Non locally owned params are dropped to free memory.  
Activations (orange) can be checkpointed. Forward pass continues until we complete model.



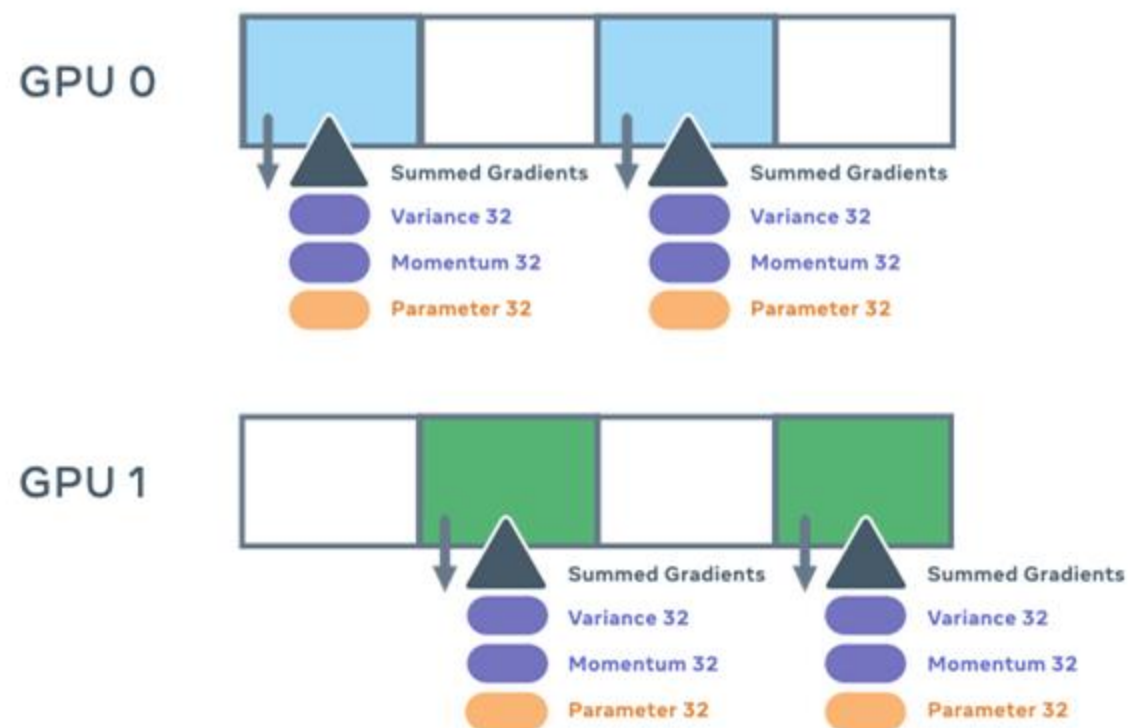
Backward Pass: another All-Gather, compute gradients.



Reduce and aggregate gradients.



Run new gradient in optimizer step.  
Update params (FP32 Master Params).  
Repeat entire cycle until training is complete.



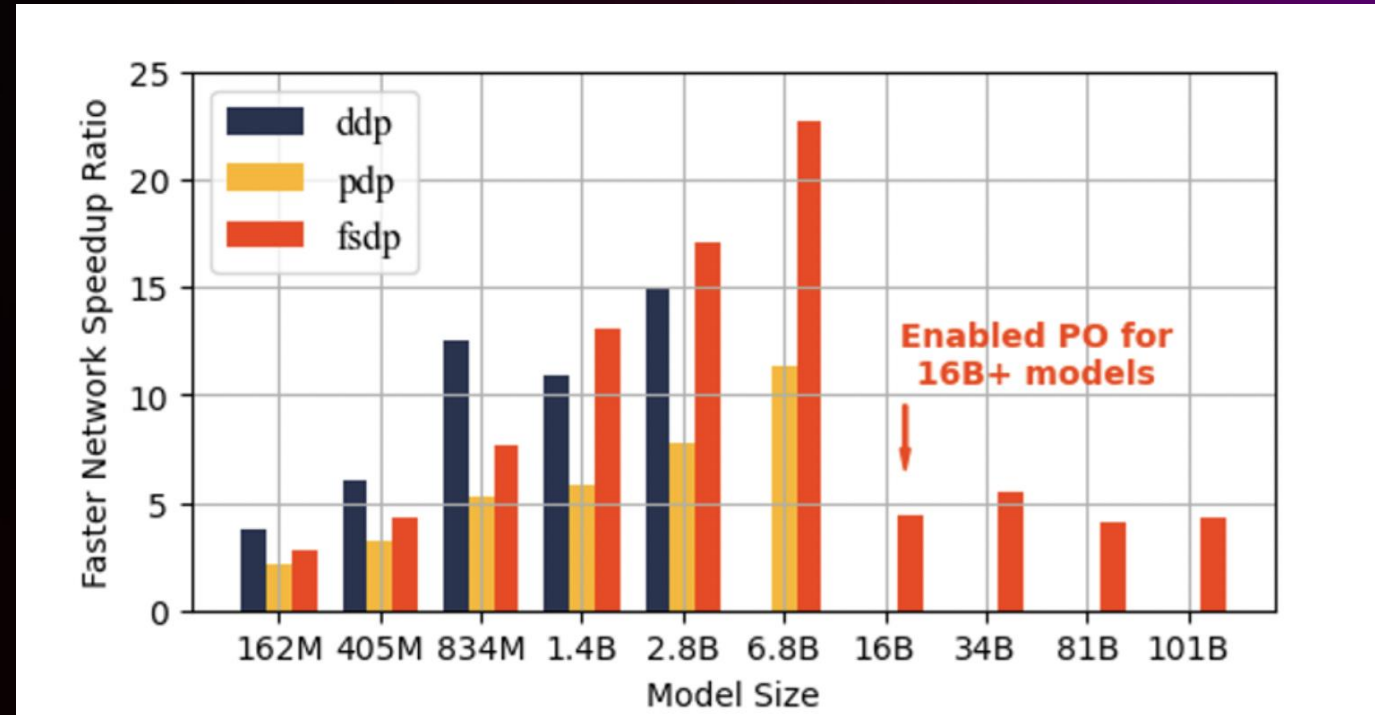
# FSDP - network communication speed is very important

FSDP interleaves communication with computation.

Thus, as communication speed increases, FSDP accelerates.

The Just in Time delivery of parameters in FSDP allows huge model scaling.

EFA on AWS enables higher speed  
(400BG/S)



<https://medium.com/pytorch/pytorch-data-parallel-best-practices-on-google-cloud-6c8da2be180d>



© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# Integrations

FSDP is integrated into HuggingFace



FSDP is integrated into Lightning



# TorchSnapshot

Highly optimized performance from

- Overlapped DtoH copy and storage I/O
- Highly parallelized storage I/O
- 0 copy tensor serialization
- (For data parallel only) evenly partitioning the write workload across all ranks

Our initial benchmark shows

- 2x faster than torch.save when saving GPU model to local FS
- 5x faster than torch.save + fsspec + s3fs when saving GPU model to S3
- For data parallel workload, up to [NUM\_DEVICES]x additional speed up

```
app_state = {"model": model, "optimizer": optimizer}

from torchsnapshot import StateDict

extra_state = StateDict(iterations=0)
app_state = {"model": model, "optimizer": optimizer, "extra_state": extra_state}

from torchsnapshot import Snapshot

snapshot = Snapshot.take(path="/path/to/my/snapshot", app_state=app_state)

snapshot.restore(app_state=app_state)
```

*Available in Torchsnapshot*

<https://pytorch.org/torchsnapshot/>



© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# TorchSnapshot

## Memory usage

- Removes the hard RAM requirement for saving/loading checkpoints. RAM usage is adaptive to available system resource and bound, which greatly reduces the chance of running out of RAM
- When RAM is abundant, TorchSnapshot can further reduce the time spent blocking in checkpoint saving by staging the model weights in RAM and flush the weights to storage in background once training resumes

## Efficient checkpoint content access

- Only read from storage what is needed (e.g. reading a tensor from a checkpoint)

```
app_state = {"model": model, "optimizer": optimizer}

from torchsnapshot import StateDict

extra_state = StateDict(iterations=0)
app_state = {"model": model, "optimizer": optimizer, "extra_state": extra_state}

from torchsnapshot import Snapshot

snapshot = Snapshot.take(path="/path/to/my/snapshot", app_state=app_state)

snapshot.restore(app_state=app_state)
```

Available in Torchsnapshot  
<https://pytorch.org/torchsnapshot/>

# DEBUGGING DISTRIBUTED TRAINING

## DEBUGGING DISTRIBUTED TRAINING



# Monitored Barrier

- Alternative primitive to `torch.distributed.barrier()`
- Mainly meant for debugging
- Option to set a custom timeout
- Rank 0 reports which rank did not enter the barrier within given timeout
- All worker fail if one rank is missing
- Only available for GLOO backend

```
import os
from datetime import timedelta

import torch
import torch.distributed as dist
import torch.multiprocessing as mp

def worker(rank):
    dist.init_process_group("nccl", rank=rank, world_size=2)
    # monitored barrier requires gloo process group to perform host-side sync.
    group_gloo = dist.new_group(backend="gloo")
    if rank not in [1]:
        dist.monitored_barrier(group=group_gloo, timeout=timedelta(seconds=2))

if __name__ == "__main__":
    os.environ["MASTER_ADDR"] = "localhost"
    os.environ["MASTER_PORT"] = "29501"
    mp.spawn(worker, nprocs=2, args=())
```

```
RuntimeError: Rank 1 failed to pass monitoredBarrier in 2000 ms
Original exception:
[gloo/transport/tcp/pair.cc:598] Connection closed by peer
[2401:db00:eef0:1100:3560:0:1c05:25d]:8594
```

# Torch\_DISTRIBUTED\_DEBUG

- Enables additional debugging information
- Two levels of detail
- INFO level
  - Prints info after initialization only
  - Enhanced crash logging due to unused parameter in model
- DETAIL
  - Print timing info during iterations
  - Has impact on training efficiency
  - Additional checks of synchronization and input of collective functions
  - Combine with  
TORCH\_SHOW\_CPP\_STACKTRACES=  
1 for full callstacks on  
desynchronization.

```
I0607 16:10:35.739390 515217 logger.cpp:173] [Rank 0]: DDP Initialized with:  
broadcast_buffers: 1  
bucket_cap_bytes: 26214400  
find_unused_parameters: 0  
gradient_as_bucket_view: 0  
is_multi_device_module: 0  
iteration: 0  
num_parameter_tensors: 2  
output_device: 0  
rank: 0  
total_parameter_size_bytes: 440  
world_size: 2  
backend_name: nccl  
bucket_sizes: 440  
cuda_visible_devices: N/A  
device_ids: 0  
dtypes: float  
master_addr: localhost  
master_port: 29501  
module_name: TwoLinLayerNet  
nccl_async_error_handling: N/A  
nccl_blocking_wait: N/A  
nccl_debug: WARN  
nccl_ib_timeout: N/A  
nccl_nthreads: N/A  
nccl_socket_ifname: N/A  
torch_distributed_debug: INFO
```

```
I0607 16:18:58.085693 544066 logger.cpp:344] [Rank 0 / 2] Training TwoLinLayerNet  
unused_parameter_size=0  
Avg forward compute time: 42850427  
Avg backward compute time: 3885553  
Avg backward comm. time: 2357981  
Avg backward comm/comp overlap time: 2234674
```

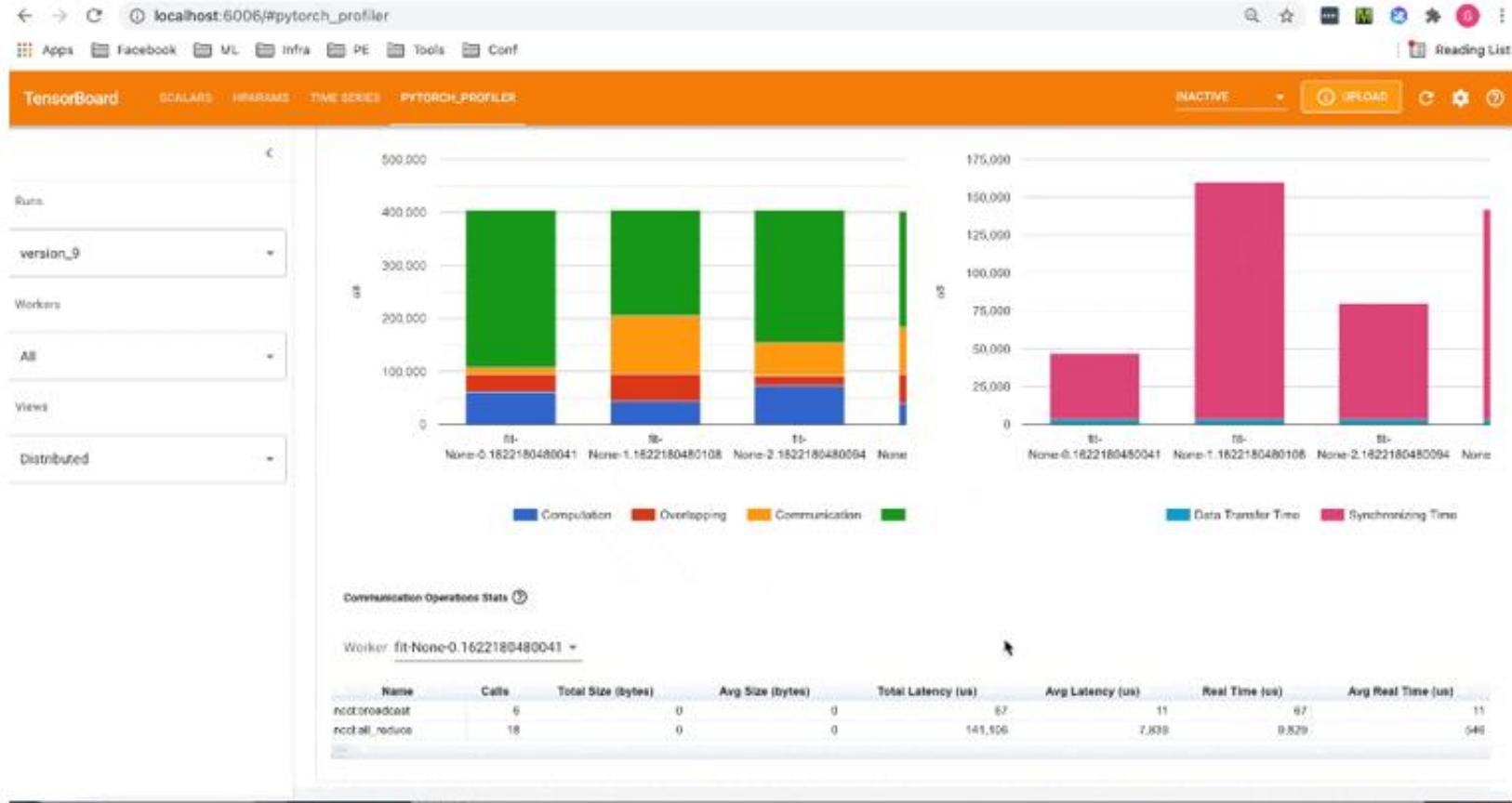


# NCCL\_DEBUG

- Set NCCL\_DEBUG for additional debug information
- VERSION
  - Just print the version at startup
- WARN
  - Lets NCCL print more explicit messages in case of an error
- INFO
  - Highest level
  - Prints excessive debug information
  - Comm channel availability
  - Ring setup
  - Un-/Normal termination

```
ip-172-31-28-185:3669:3669 [0] NCCL INFO Bootstrap : Using ens3:172.31.28.185<0>
ip-172-31-28-185:3669:3669 [0] NCCL INFO NET/Plugin : No plugin found (libnccl-net.so),
ip-172-31-28-185:3669:3669 [0] NCCL INFO NET/IB : No device found.
ip-172-31-28-185:3669:3669 [0] NCCL INFO NET/Socket : Using [0]ens3:172.31.28.185<0>
ip-172-31-28-185:3669:3669 [0] NCCL INFO Using network Socket
NCCL version 2.10.3+cuda11.3
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 00/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 01/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 02/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 03/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 04/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 05/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 06/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 07/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 08/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 09/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 10/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 11/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 12/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 13/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 14/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 15/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 16/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 17/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 18/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 19/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 20/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 21/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 22/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 23/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 24/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 25/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 26/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 27/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 28/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 29/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 30/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Channel 31/32 : 0
ip-172-31-28-185:3669:3679 [0] NCCL INFO Trees [0] -1/-1/-1->0->-1 [1] -1/-1/-1->0->-1
1->0->-1 [16] -1/-1/-1->0->-1 [17] -1/-1/-1->0->-1 [18] -1/-1/-1->0->-1 [19] -1/-1/-1->
ip-172-31-28-185:3669:3679 [0] NCCL INFO Connected all rings
ip-172-31-28-185:3669:3679 [0] NCCL INFO Connected all trees
ip-172-31-28-185:3669:3679 [0] NCCL INFO 32 coll channels, 32 p2p channels, 32 p2p chan
ip-172-31-28-185:3669:3679 [0] NCCL INFO comm 0x7f223c002f70 rank 0 nranks 1 cudaDev 0
```

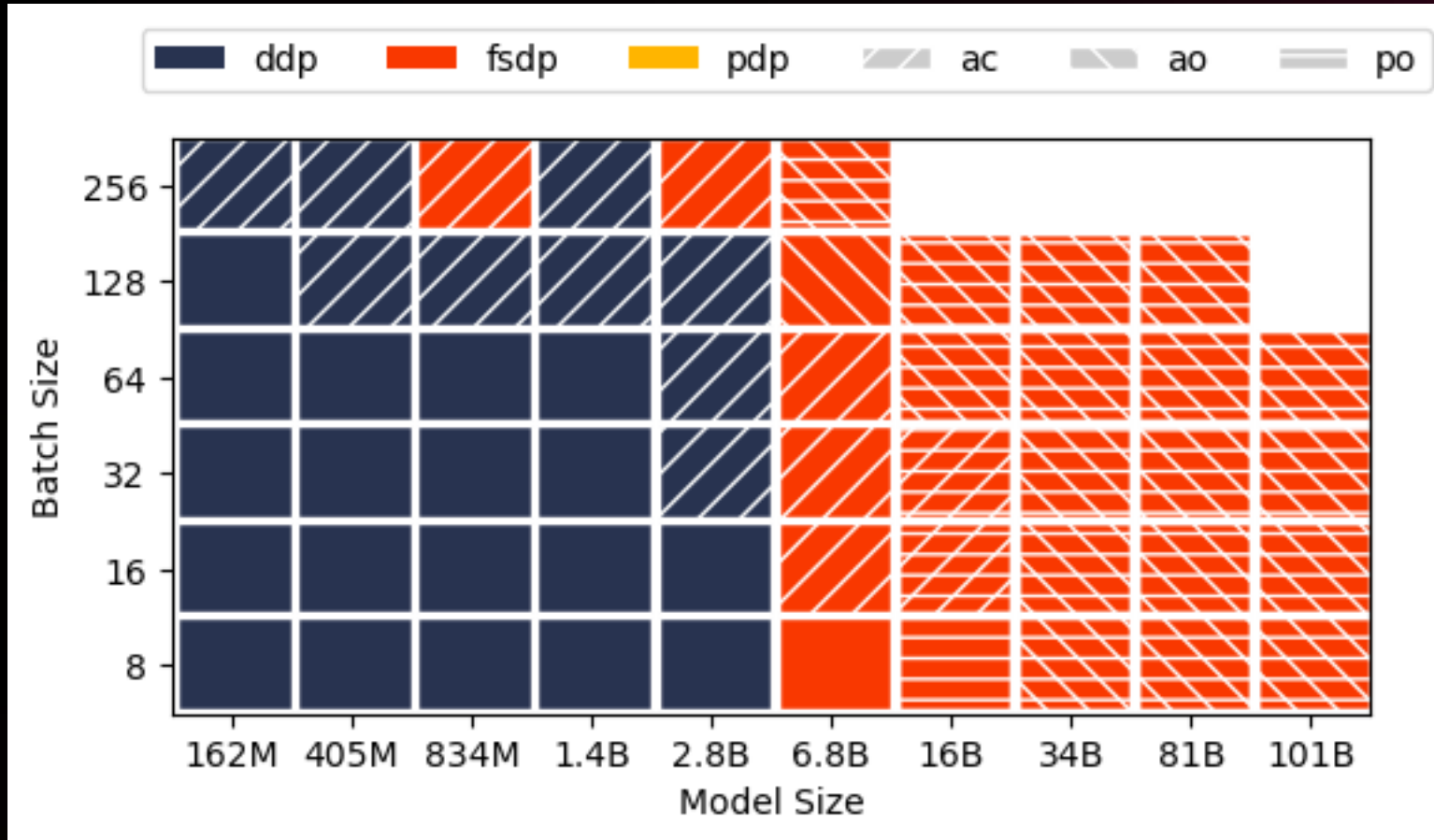
# Pytorch Profiler





# Where does FSDP fit in (vs DDP)

Model scale and recommendations are network speed dependent:  
With NVLink, 100M+ = FSDP! (universal training)





# Speedup over G4dn

Model	Batch size	Precision	Throughput (batch size X requests/sec)		Latency (ms)		\$/million inferences (on demand)		Cost benefit (G5 over G4dn)
			G5	G4dn	G5	G4dn	G5	G4dn	
Bert-base-uncased	32	Full	723	154	44	208	\$0.6	\$2.2	3.7X
		Mixed	870	410	37	79	\$0.5	\$0.8	1.6X
	16	Full	651	158	25	102	\$0.7	\$2.1	3.0X
		Mixed	762	376	21	43	\$0.6	\$0.9	1.5X
	8	Full	642	142	13	57	\$0.7	\$2.3	3.3X
		Mixed	681	350	12	23	\$0.7	\$1.0	1.4X
	1	Full	160	116	6	9	\$2.8	\$2.9	1.0X
		Mixed	137	102	7	10	\$3.3	\$3.3	1.0X
ResNet50	32	Full	941	397	34	82	\$0.5	\$0.8	1.6X
		Mixed	1533	851	21	38	\$0.3	\$0.4	1.3X
	16	Full	888	384	18	42	\$0.5	\$0.9	1.8X
		Mixed	1474	819	11	20	\$0.3	\$0.4	1.3X
	8	Full	805	340	10	24	\$0.6	\$1.0	1.7X
		Mixed	1419	772	6	10	\$0.3	\$0.4	1.3X
	1	Full	202	164	5	6	\$2.2	\$2	0.9X
		Mixed	196	180	5	6	\$2.3	\$1.9	0.8X

# Amazon EC2 **G5** instances

Using NVIDIA A10G GPU with 24GB GPU memory

- 3–4x higher graphics and ML performance
- 80 2<sup>nd</sup>-gen RT cores



AMD Rome CPUs

- Up to 3.3GHz core frequency

Instance size	GPU	vCPU	Memory (GB)	GPU memory (GB)	Storage (GB)	Network bandwidth (Gb/s)	EBS bandwidth (Gb/s)
<b>G5.xlarge</b>	1	4	16	24	250	Up to 10	Up to 3.5
<b>G5.2xlarge</b>	1	8	32	24	450	Up to 10	Up to 3.5
<b>G5.4xlarge</b>	1	16	64	24	600	Up to 25	8
<b>G5.8xlarge</b>	1	32	128	24	900	25	16
<b>G5.16xlarge</b>	1	64	256	24	1900	25	16
<b>G5.12xlarge</b>	4	48	192	96	3800	40	16
<b>G5.24xlarge</b>	4	96	384	96	3800	50	19
<b>G5.48xlarge</b>	8	192	768	192	7600	100	19

# Technical details

- AWS Inferentia is 45% cheaper than g4dn instances and can achieve 3.5 times the throughput for large NLP models.
- Compiled models have 5–6 times smaller memory footprint.
- Benchmarked throughput of models while loading and unloading in parallel on a GPU, low GPU utilization because load and unload API calls given preference than inference API.
- Similar default behavior with AWS Inferentia Neuron Cores; models get loaded sequentially on Neuron Cores.