Day12 - Scaling applications

02476 Machine Learning Operations

Nicki Skafte Detlefsen, Associate Professor, DTU Compute

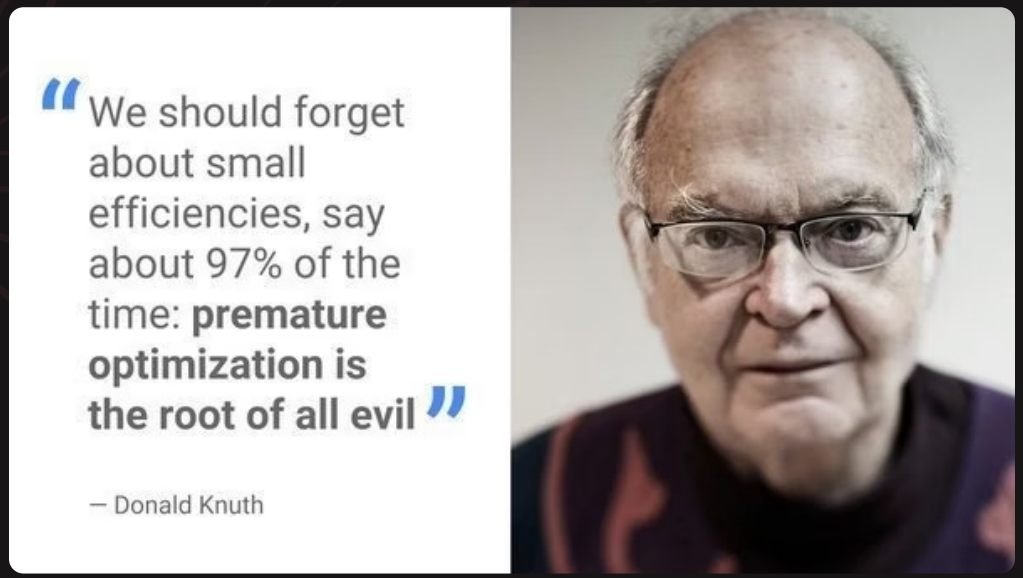
January 2026

When can we start

💡Scaling applications can be important to meet requirements

💡We should only do it when we have a working system

💡Else we run into problems of premature optimization



What is a distributed application

> *Computing on multiple threads/devices/nodes in parallel*

What can run in parallel

💡 Data loading

💡 Training

💡 Inference

The key take away

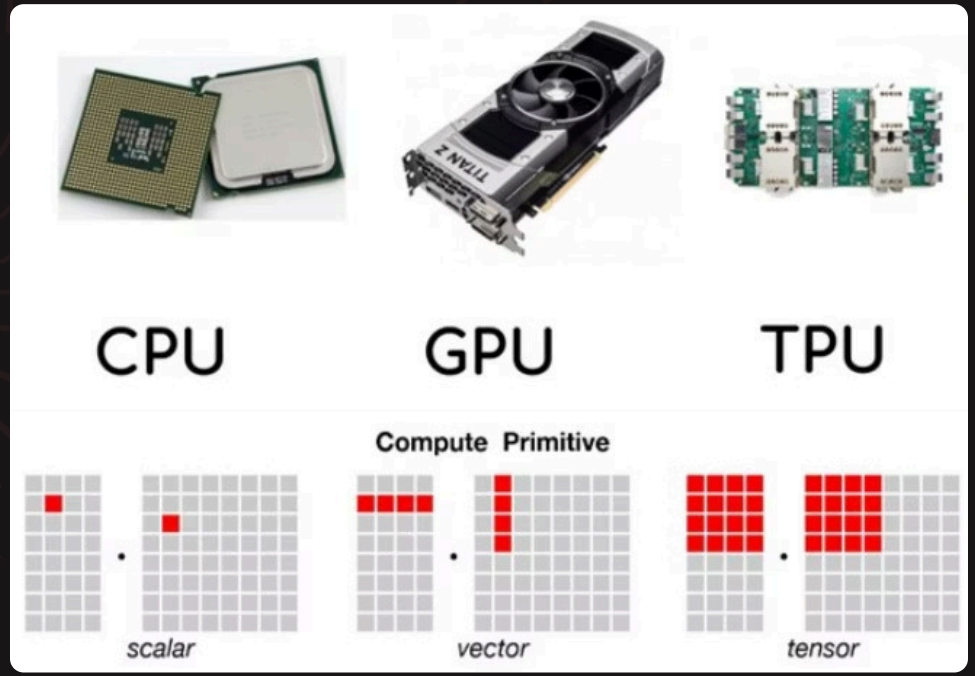
⚠️Distributed computation is not always beneficial, its a trade-off:

Communication overhead

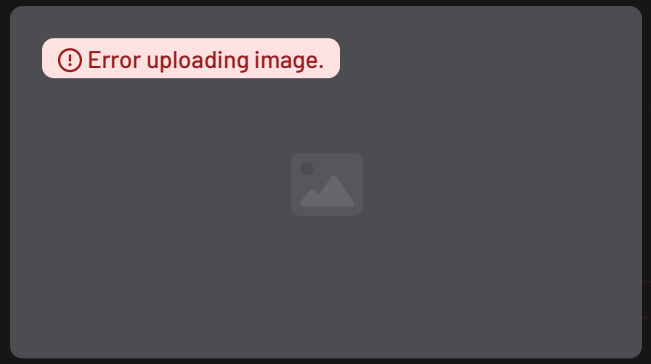
Device utilization

A little side step: Devices

- 💡CPU
  - 🔥General compute unit
  - 🔥2-128 parallel operations
- 💡GPU
  - 🔥Rendering unit
  - 🔥1.000-10.000 parallel operations
- 💡TPU
  - 🔥Specialized unit
  - 🔥32.000 - 128.000 parallel operations



A little side step: Basic communication operations



- 💡 Scatter
- 💡 Gather
- 💡 Reduce
- 💡 Broadcast
- 💡 All-gather
- 💡 All-reduce

Rank 0: main

Rank >0: worker

Device memory

🔥Equally important is the amount of memory you have available

With more memory you get

💡Faster data transfer

💡Possibility of higher data modalities

💡Larger models

	CPU	GPU	TPU
Standard	34-64 GiB	24 GiB	64 GiB
Maximum	2 TiB	140 GiB	32 Tib

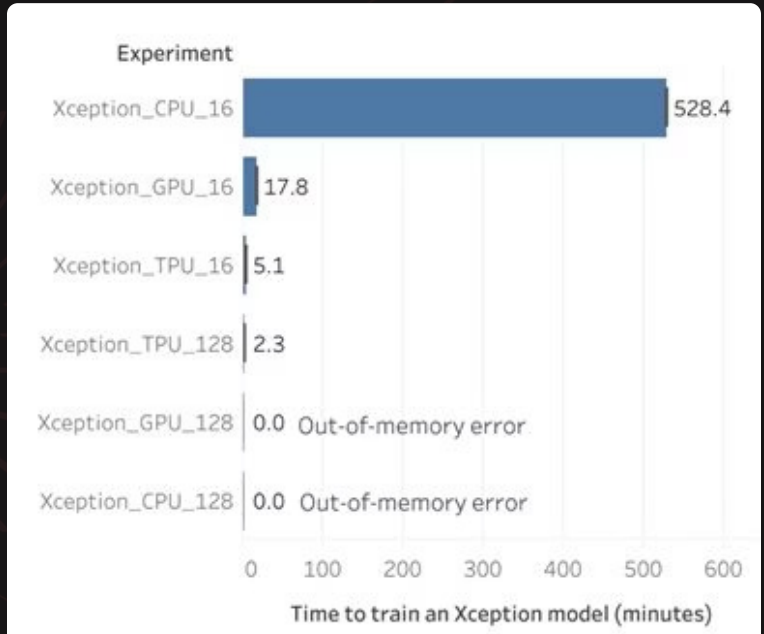
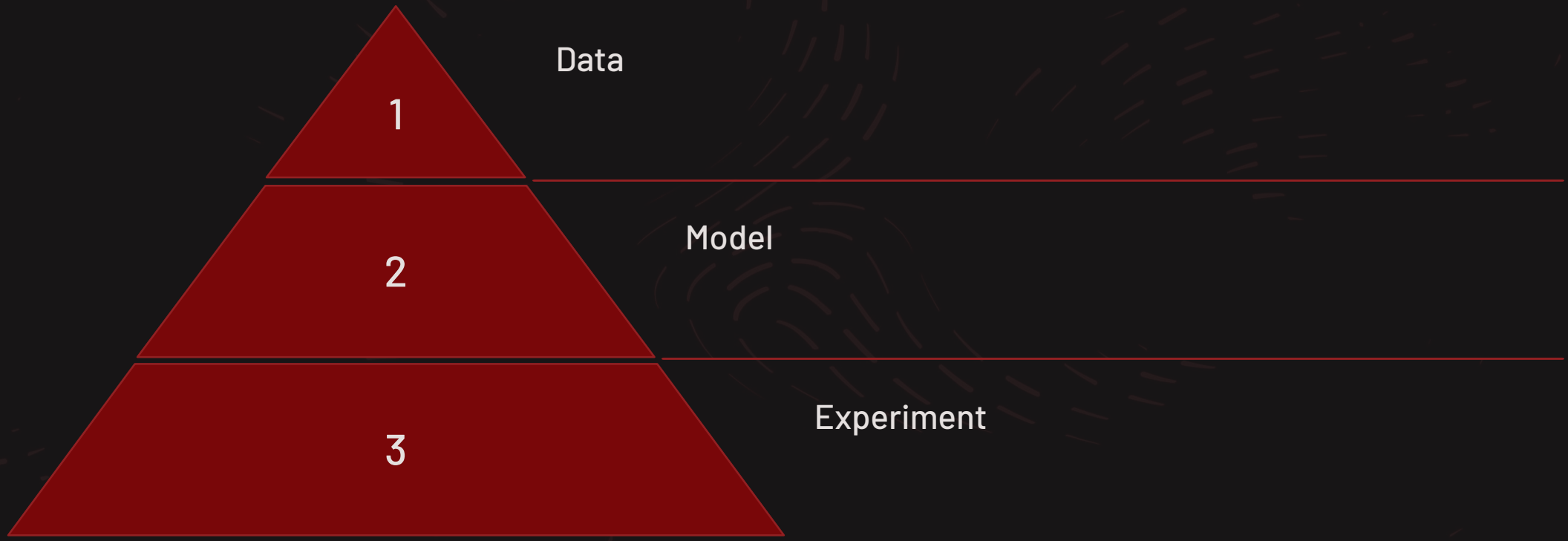


Figure 3: CPUs vs GPUs vs TPUs for training an Xception model for 12 epochs. Y-Axis labels indicate the choice of model, hardware, and batch size for each experiment. Increasing the batch size to 128 for TPUs resulted in an additional ~2x speedup.

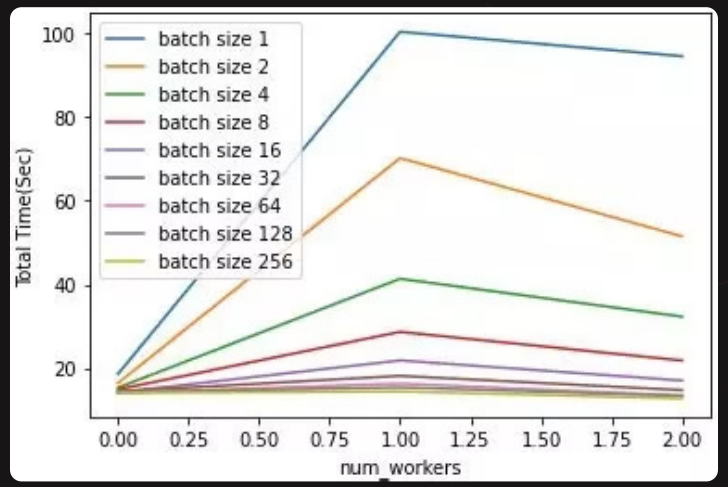
Many layers of distributed computations



Choose one or more to parallelize over

How to parallelize data loading?

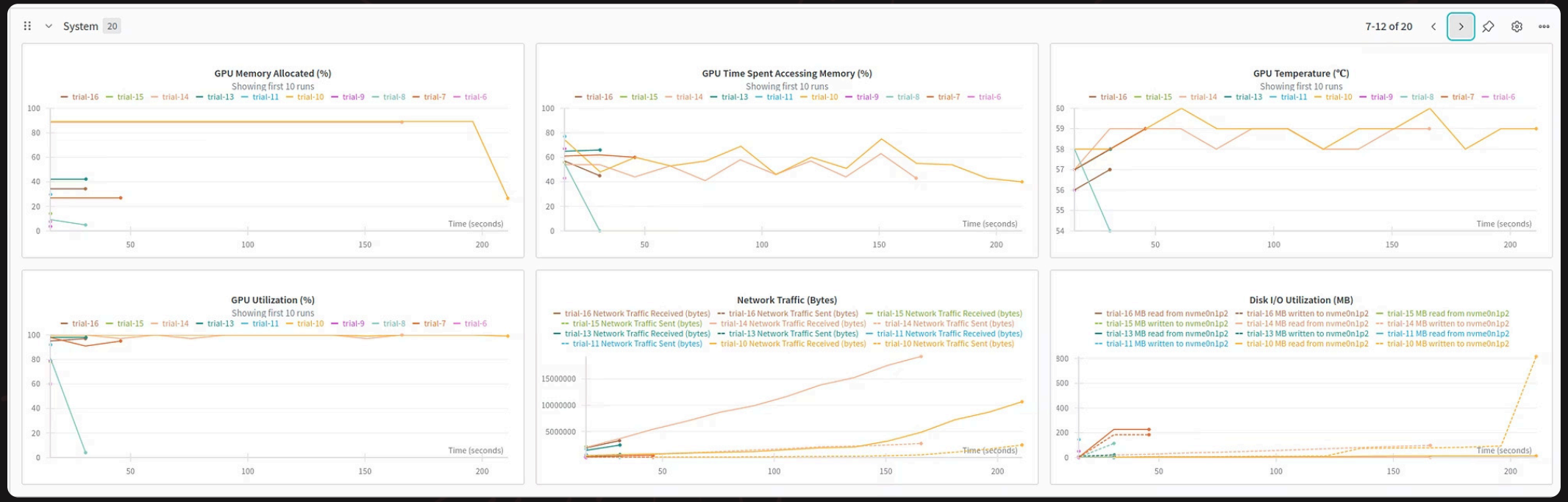
Large batch size + `num_workers>1` in Pytorch




```
from torch.utils.data import Dataset

class SimpleDataset(Dataset):
    def __init__(self, data, labels):
        self.data = data
        self.labels = labels

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        x = self.data[idx]
        y = self.labels[idx]
        return x, y
```

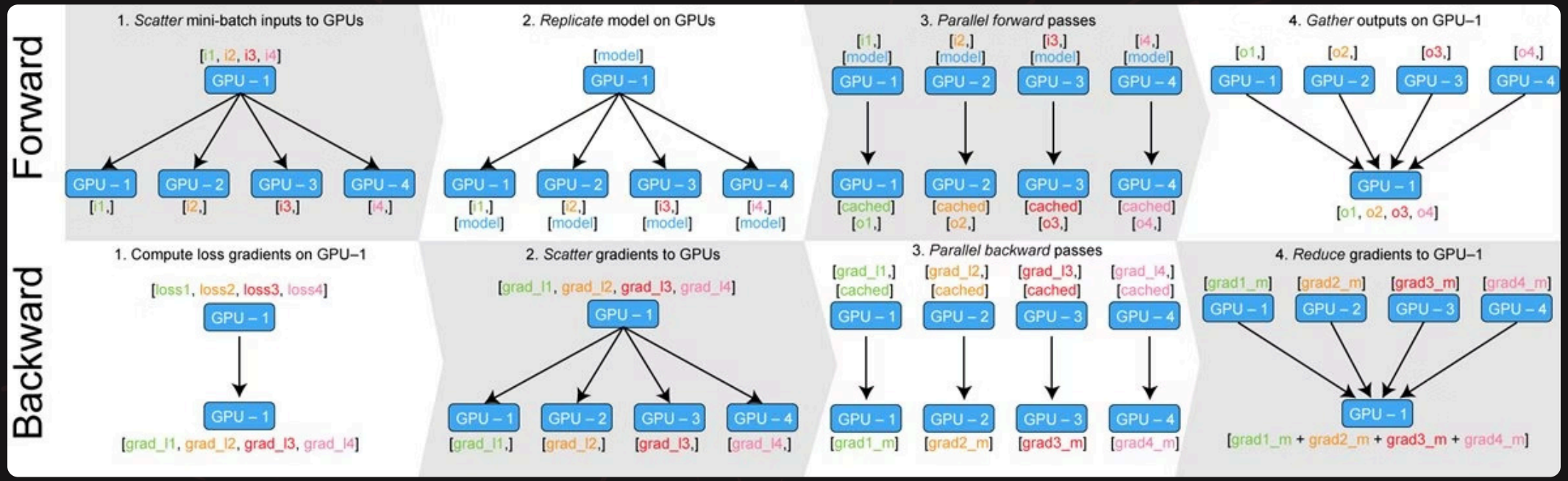
How do I know if I am bottle necked?

If you are using a experiment logger, good chance you are getting this for free :)

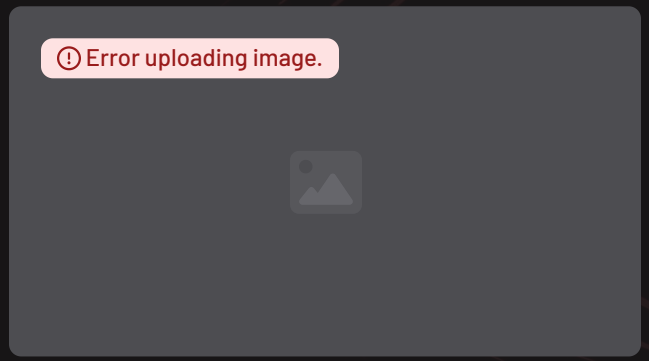


Look for system metrics in wandb, https://docs.wandb.ai/models/ref/python/experiments/system-metrics

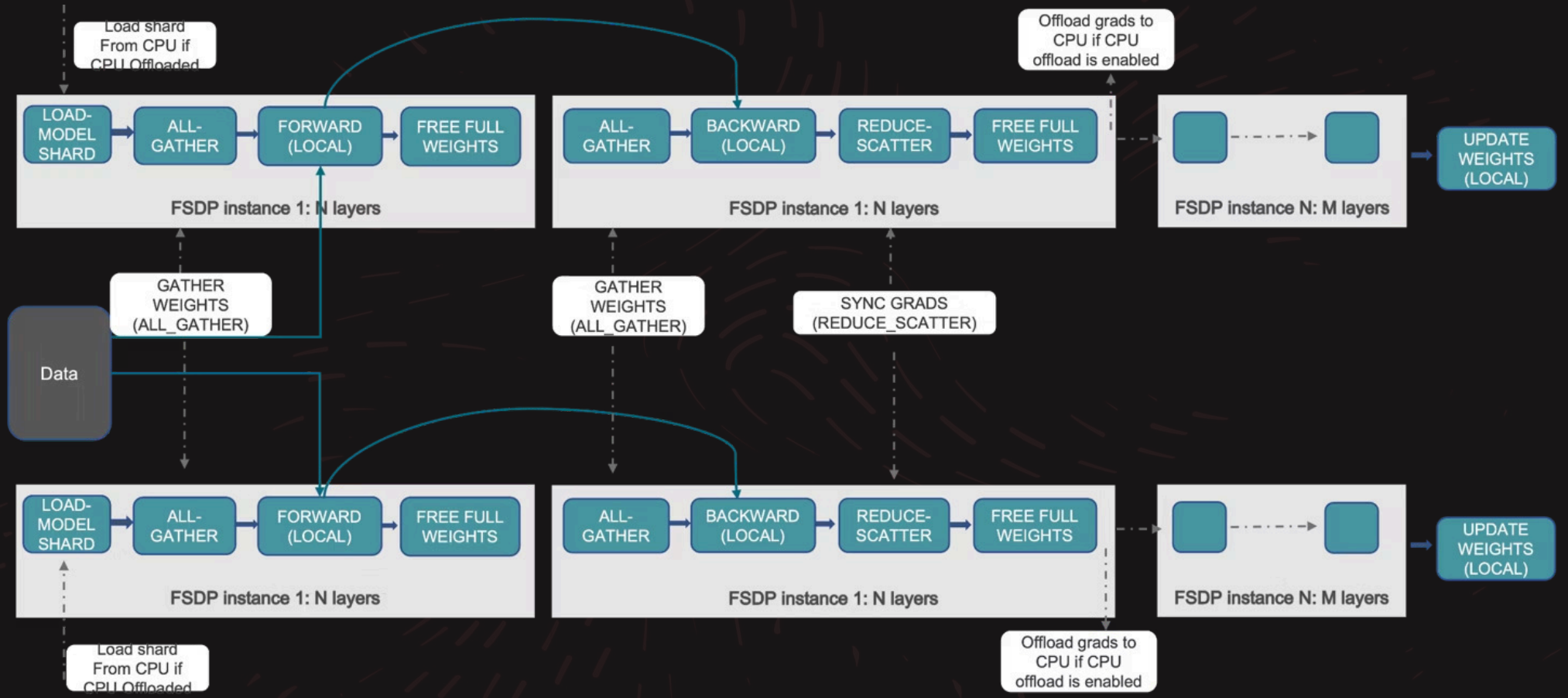
Models: Data parallel



Models: Distributed data parallel



Models: Fully sharded data parallel



How to do this in Pytorch?

💡Dataparallel

- parallel_model = torch.nn.DataParallel(model)

💡Distributed data parallel (DDP)

- Set a environment MASTER_ADDR and MASTER_PORT
- Initialize a process group
- parallel_model = nn.parallel.DistributedDataParallel(model, device_ids=[gpu])
- Use mp.spawn to spawn multiple processes
- ...

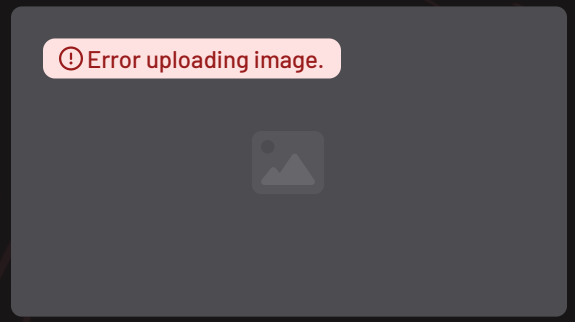
💡Model parallelism

- Just don't

Method	Pros	Cons
Data parallel	Simple to use	Slow due to replicas being destroyed
Distributed data parallel	Fast	High memory requirement
Fully sharded Data Parallel	Large models that other methods	Can be slower than DDP due to high communication cost

Instead use any high level framework

```
# run on cpu, gpu, tpu, ipu
# with no code changes needed
trainer = Trainer(devices=8, accelerator='cpu')
trainer = Trainer(devices=8, accelerator='gpu')
trainer = Trainer(devices=8, accelerator='tpu')
trainer = Trainer(devices=8, accelerator='ipu')

# or just let lightning auto detect
trainer = Trainer(devices=8, accelerator='auto')
```



Optimizing Precision: AMP & Quantization

1

Automatic Mixed Precision (AMP)

- **Goal:** Accelerate training and reduce VRAM usage without sacrificing accuracy.
- **Concept:** Uses 16-bit (FP16/BF16) for most operations while keeping critical values (like gradients/loss) in 32-bit (FP32) to prevent underflow.
- **Implementation:** In PyTorch, use `torch.autocast` and `GradScaler`.

2

Model Quantization

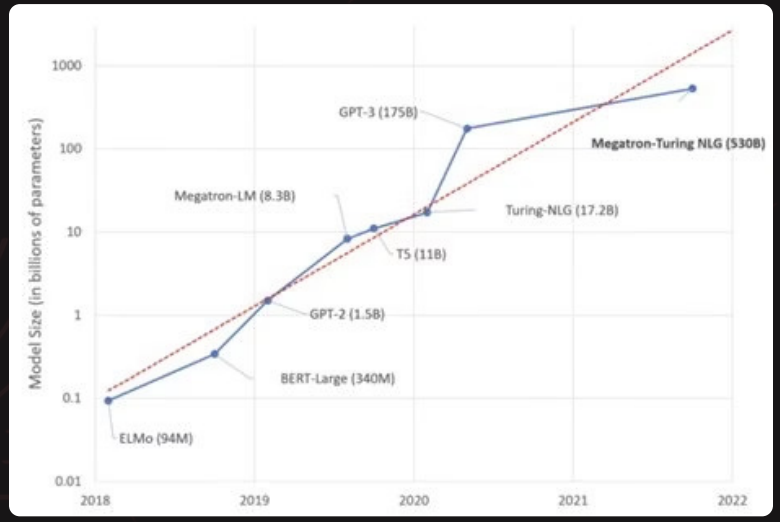
- **Goal:** Drastically reduce model size and latency for deployment/inference.
- **Concept:** Converts weights and activations from 32-bit floating point to lower bit-widths, typically **INT8** (4x smaller).
- **Two Main Strategies:**
  - **Post-Training Quantization (PTQ):** Fast, applies to a pre-trained model with a small calibration set.
  - **Quantization-Aware Training (QAT):** Models the quantization error during training for higher accuracy at low bit-widths.

Above and beyond

Scaling matters in deep learning

Deepspeed is highly optimized for overlapping communication across devices with computation going on

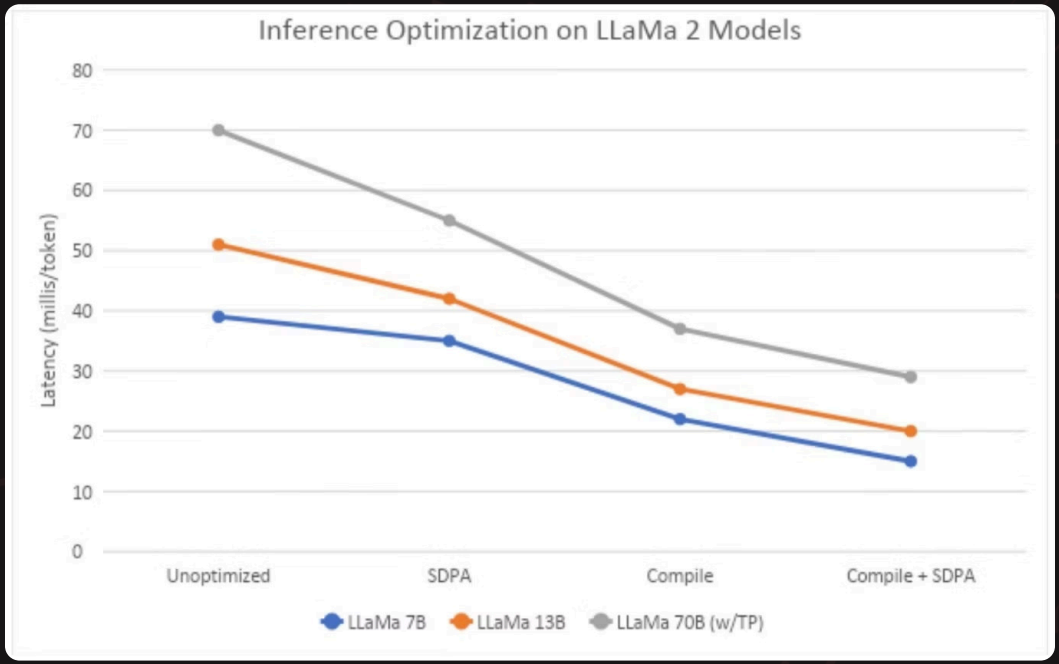
[GitHub - deepspeedai/DeepSpeed: DeepSpeed is a deep learning optimization library that makes distributed training and inference easy, efficient, and effective.]

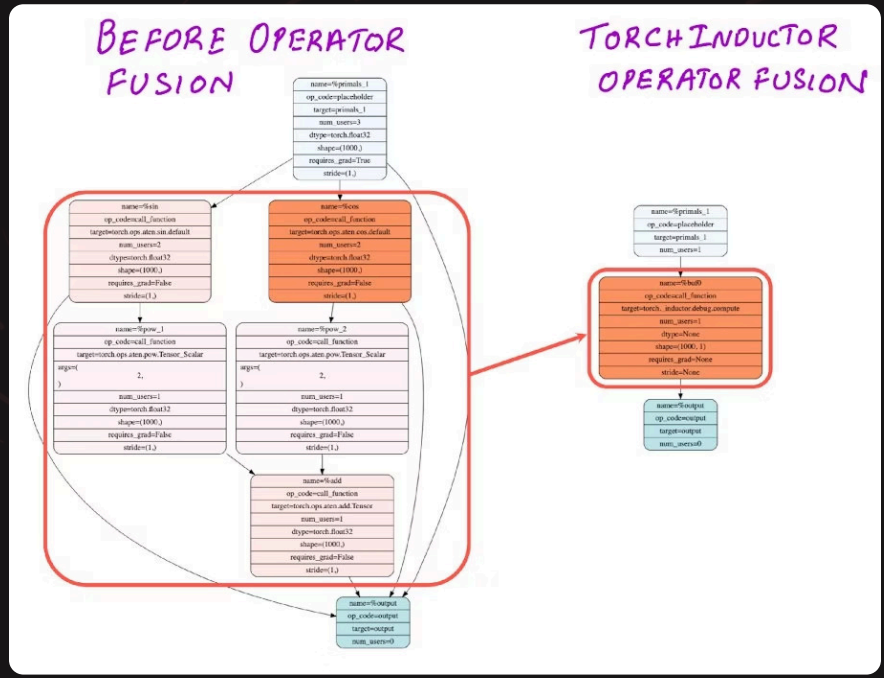



```
# Sharded training using fairscale
trainer = Trainer(devices=4, strategy='ddp_sharded')

# sharded training using deepspeed
trainer = Trainer(devices=4, strategy="deepspeed_stage_1", precision=16)
trainer = Trainer(devices=4, strategy="deepspeed_stage_2", precision=16)
trainer = Trainer(devices=4, strategy="deepspeed_stage_3", precision=16)
```

Remember to compile your model

⚠️In Pytorch use model = torch.compile(model)





Other smaller optimizations

Checkpoints

Frequent checkpointing with asynchronous saving prevents training pauses during disk writes.

GPU Augmentation

Move augmentations to GPU using DALI when CPU is the bottleneck.

Data Formats

Use WebDataset, Parquet, or HDF5 instead of raw images/CSVs to reduce I/O bottlenecks.

Prefetching

Use `prefetch_factor` in DataLoader to fetch batches ahead, keeping GPU from waiting.

What about inference?

⚠️Use batch prediction when possible

```python
from fastapi import FastAPI
from pipeline import model,
                     clean_data,
                     format_data,
                     data_is_valid

app = FastAPI()

@app.post("/predict/")
async def predict(item):

    if not data_is_valid(item):
        return {"message": "data not valid"}

    item = clean_data(item)
    predictions = model.predict(item)
    output = format_data(predictions)

    return output
```

```python
from fastapi import FastAPI
from typing import List
from pipeline import model,
                     clean_data,
                     format_data,
                     data_is_valid

app = FastAPI()

@app.post("/batch-predict/")
async def predict(items: List[str]):

    items = list(set(items)) # <- remove duplicates

    items = [i for i in items
            if data_is_valid(i) == True] # <- leverage list comprehensions

    items = clean_data(items) # <- probably has some numpy or pandas
    predictions = model.predict(items) # <- faster and more efficient than calli
    outputs = format_data(predictions)

    return outputs
```

What about inference?

⚠️Use caching if possible

```python
import functools

@functools.lru_cache(maxsize=128)
def fib(n):
  if n < 2:
    return 1
  return fib(n-1) + fib(n-2)
```

```
$ python3 -m timeit -s 'from fib_test import fib' 'fib(30)'
10 loops, best of 3: 282 msec per loop
$ python3 -m timeit -s 'from fib_test import fib_cache' 'fib_cache(30)'
10000000 loops, best of 3: 0.0791 usec per loop
```

Couple of last notes

- Yes, the material will stay up
- Project report: https://github.com/SkafteNicki/dtu_mlops/tree/main/reports
- Project checklist:
  - No, there is not a specific number I look for to be fulfilled
  - Yes, you need to have some cloud in the project
- The remaining week is reserved for your projects:
  - Tomorrow (Wednesday) the last lecture, a simple summary lecture will take place. Fully online as I will not be at campus.
  - Thursday and Friday no lectures.
  - Thursday and Friday we need to vacate the auditorium due to renovations
- Please remember to do evaluation: https://evaluering.dtu.dk/

Meme of the day

https://skaftenicki.github.io/dtu_mlops/s9_scalable_applications/