DTU

02476 Machine Learning Operations

Nicki Skafte Detlefsen

Scaling applications

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

> “We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**”
>
> — Donald Knuth

What is a distributed application

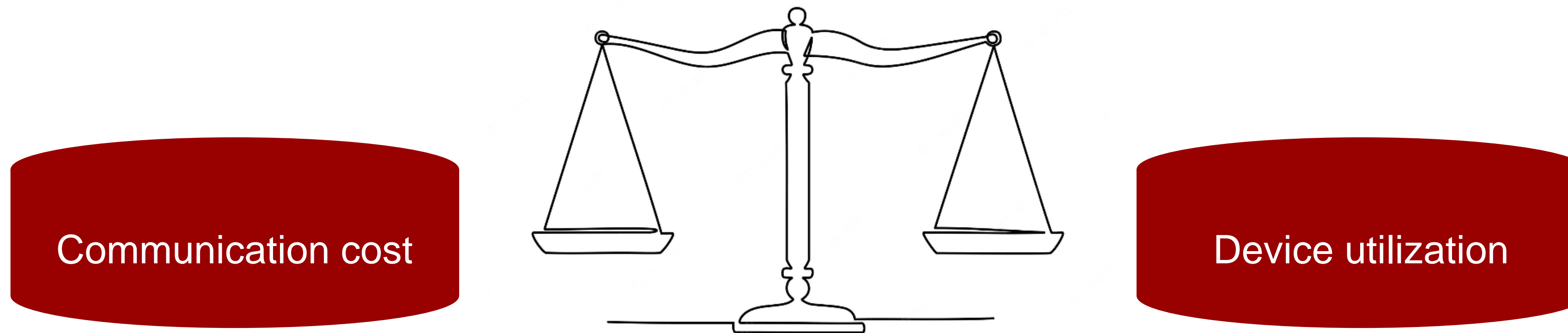
Computing on multiple threads/devices/nodes in parallel

What can run in parallel

- 💡 Data loading
- 💡 Training
- 💡 Inference

The key take away

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

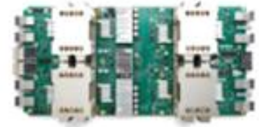


Lets take a look at training

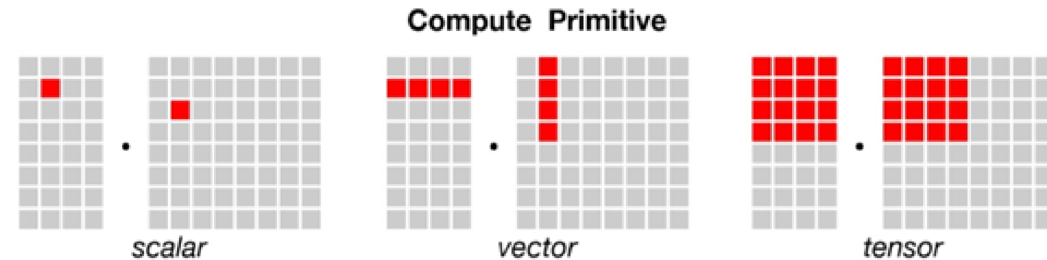
Devices

Three common types of 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



CPU GPU TPU



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	12 GiB	64 GiB
Maximum	2 TiB	80 GiB	32 Tib

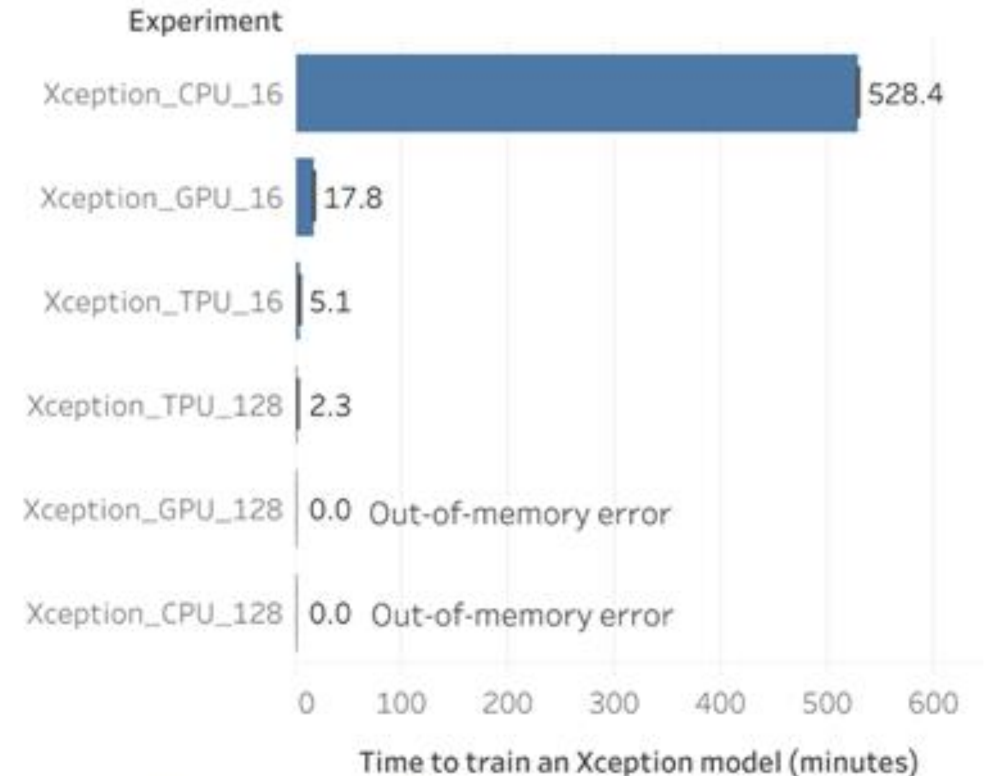
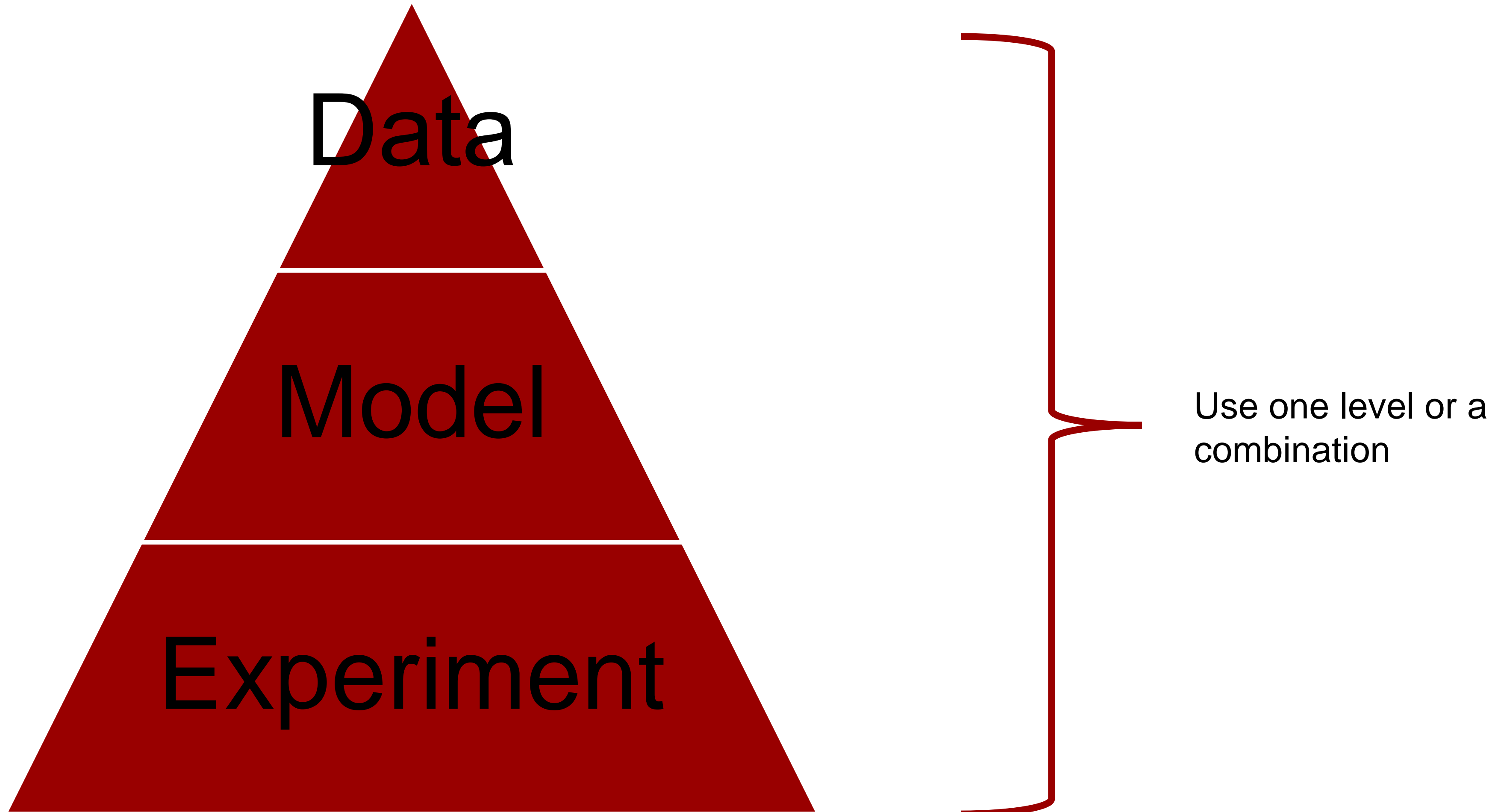


Figure 3: CPUs vs GPUs vs TPUs for training an Xception model for 12 epochs. Y-Axis labels indicate the choice of model, hardware, and batch size for each experiment. Increasing the batch size to 128 for TPUs resulted in an additional ~2x speedup.

Many layers of distributed computations



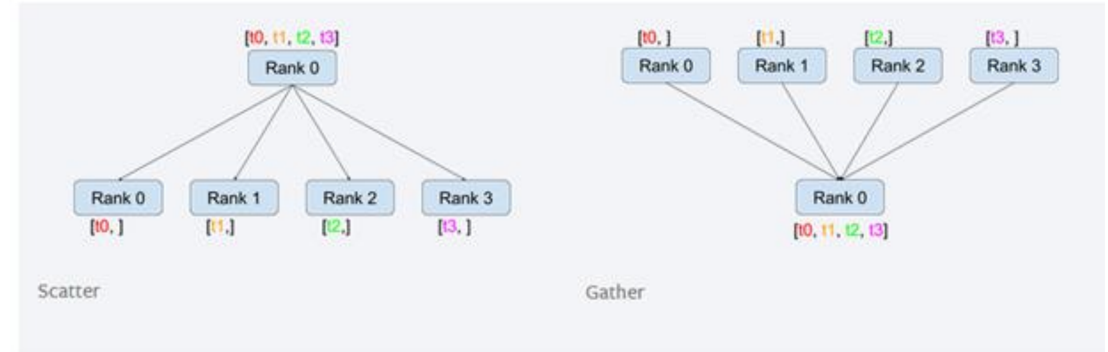
DTU

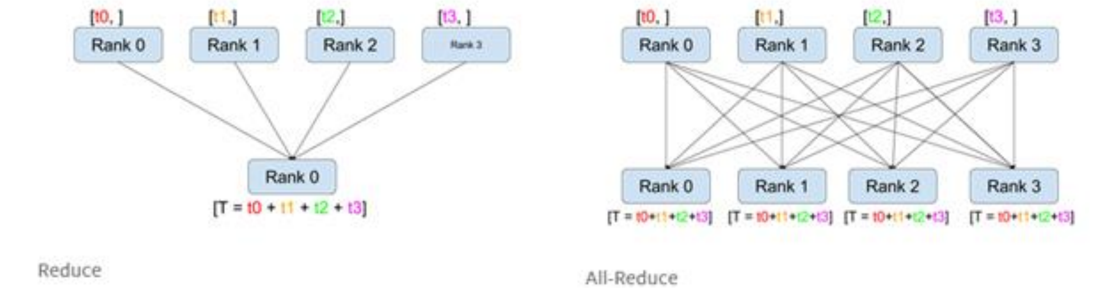
Basic communication operations

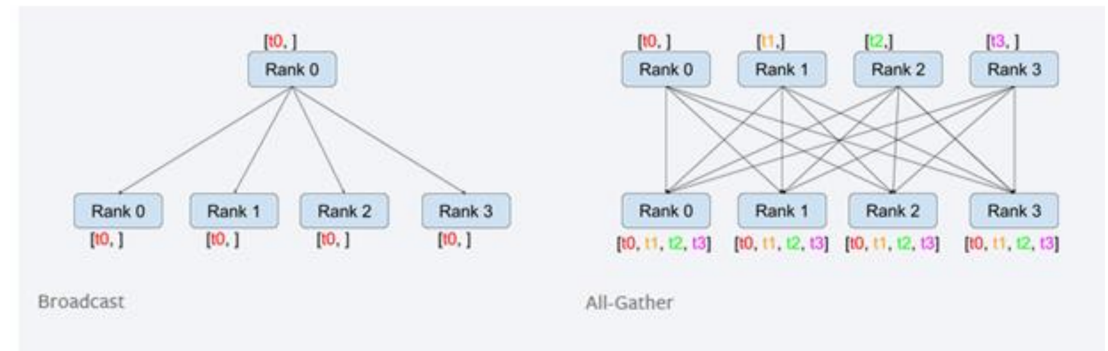
- Scatter
- Gather
- Reduce
- Broadcast
- All-gather
- All-reduce

Rank 0: main

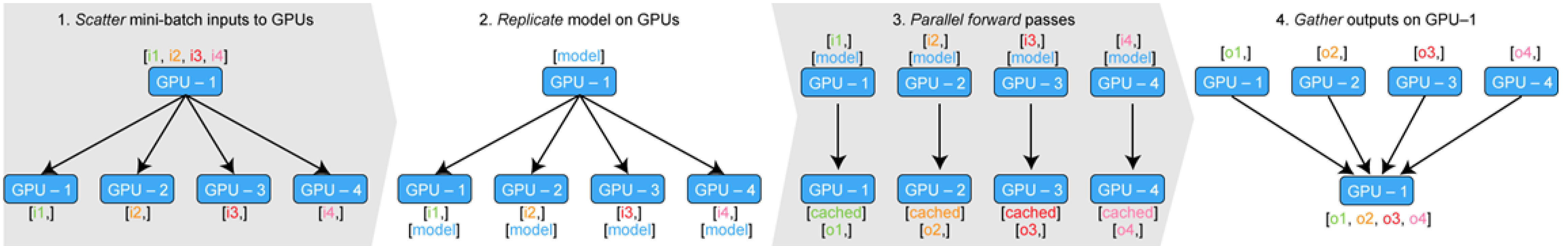
Rank >0: worker

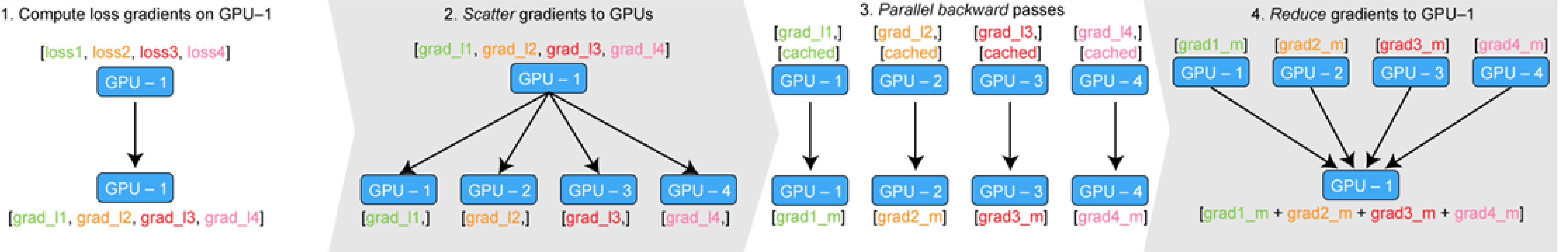






Data parallel





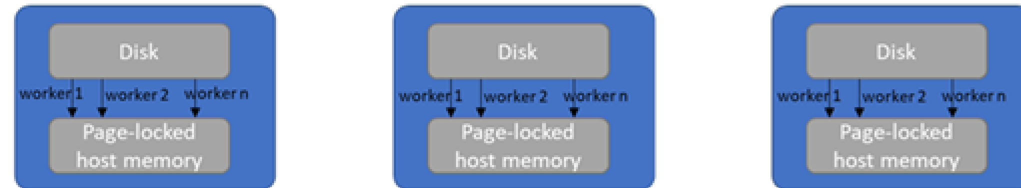
Distributed data parallel

Distributed Data Parallel

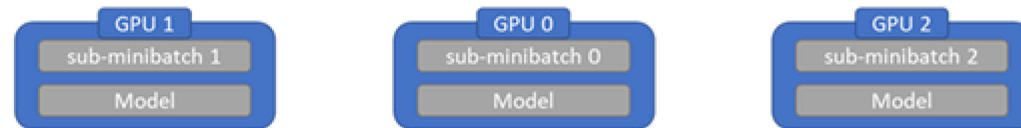
No master GPUs

Implemented in PyTorch DistributedDataParallel module

1. Load data from disk into page-locked memory on the host. Use multiple worker processes to parallelize data load. Distributed minibatch sampler ensures that each process loads non-overlapping data

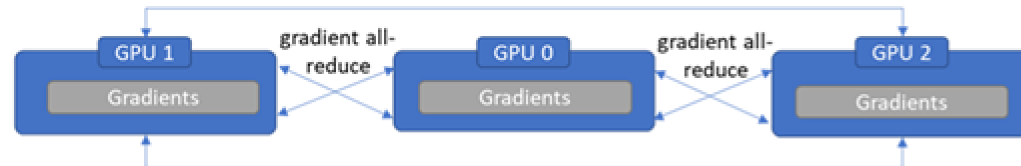


2. Transfer minibatch data from page-locked memory to each GPU concurrently. No data broadcast is needed. Each GPU has an identical copy of the model and no model broadcast is needed either



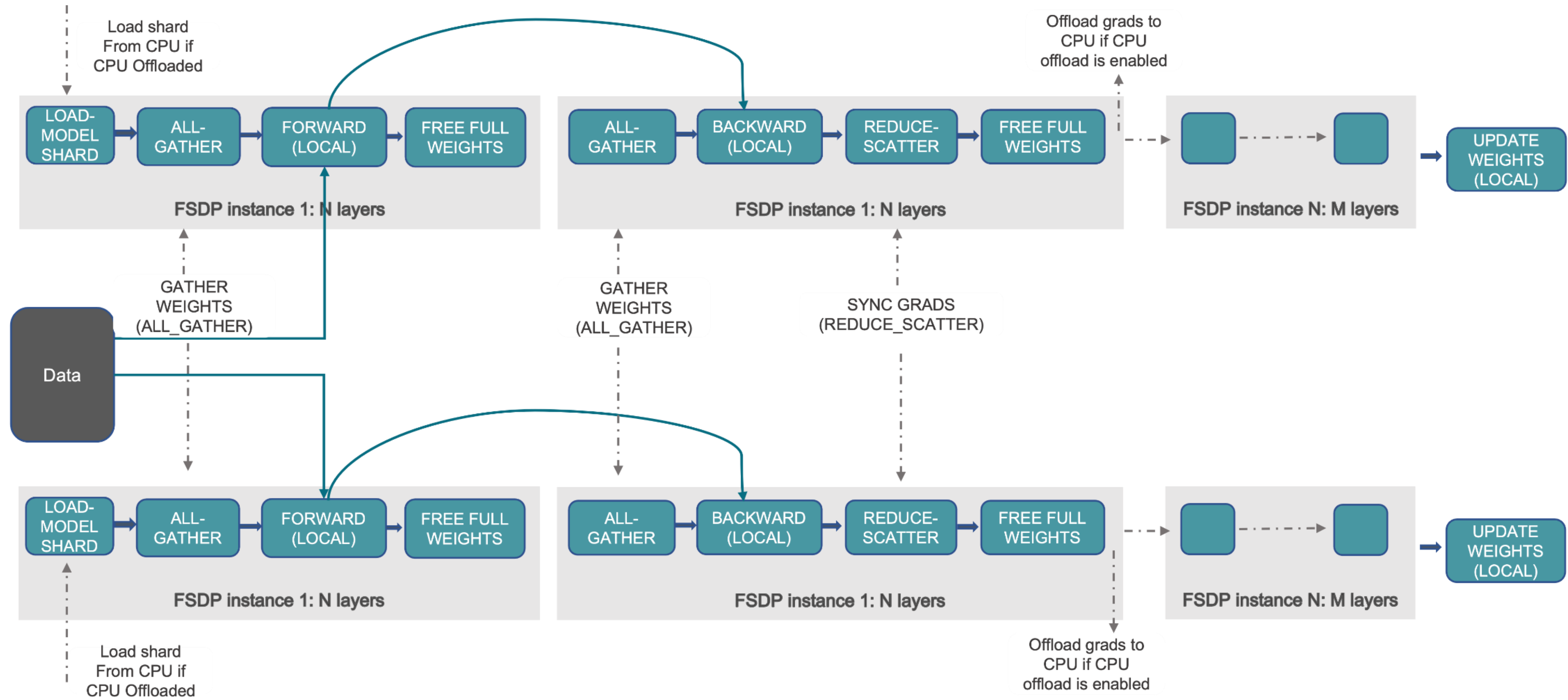
3. Run forward pass on each GPU, compute output

4. Compute loss, run backward pass to compute gradients. Perform gradient all-reduce in parallel with gradient computation



5. Update Model parameters. Because each GPU started with an identical copy of the model and gradients were all-reduced, weights updates on all GPUs are identical. Thus no model sync is required

Fully sharded Data Parallel



Comparison

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

How to do it 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

Instead use any high level framework

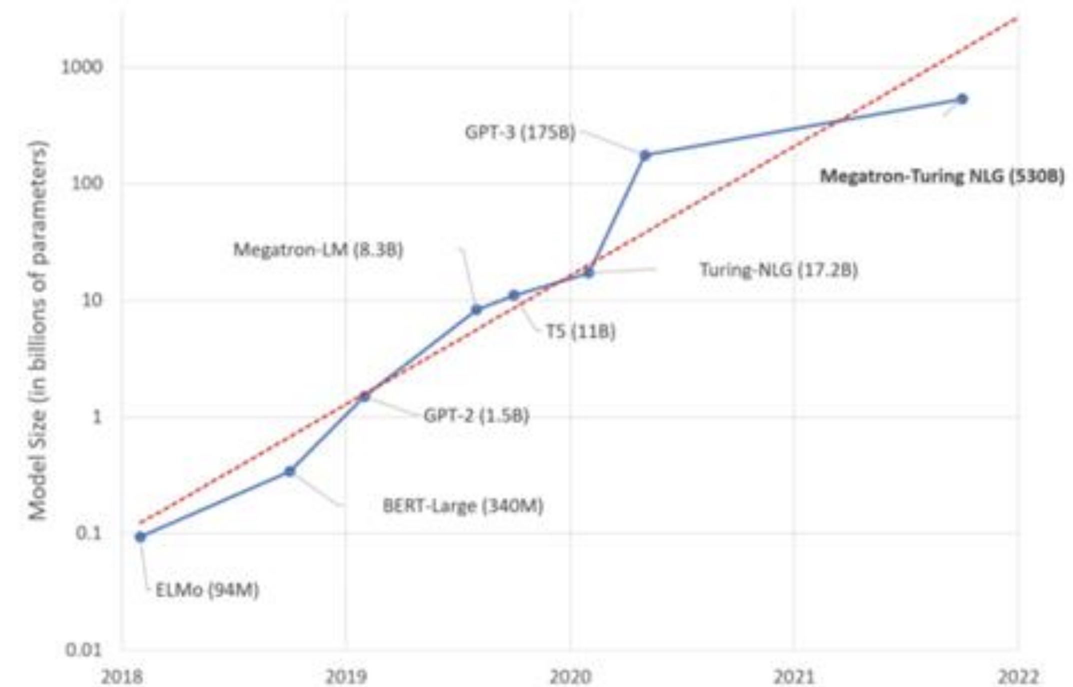
```
# run on cpu, gpu, tpu, ipu
# with no code changes needed
trainer = Trainer(devices=8, accelerator='cpu')
trainer = Trainer(devices=8, accelerator='gpu')
trainer = Trainer(devices=8, accelerator='tpu')
trainer = Trainer(devices=8, accelerator='ipu')

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

```
# for gpu, you can also do multiple nodes
# 32 nodes * 8 gpus per node = 256 gpus!
trainer = Trainer(devices=8, accelerator='gpu', num_nodes=32)
```

Above and beyond

Scaling matters in deep learning

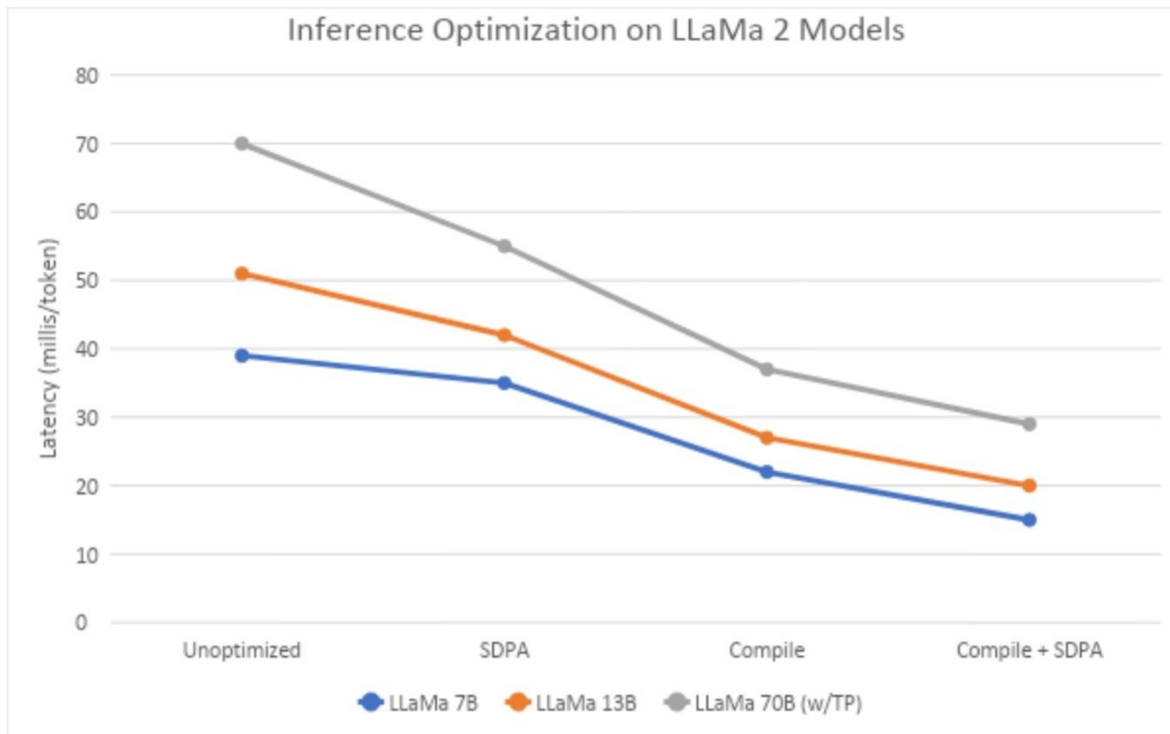


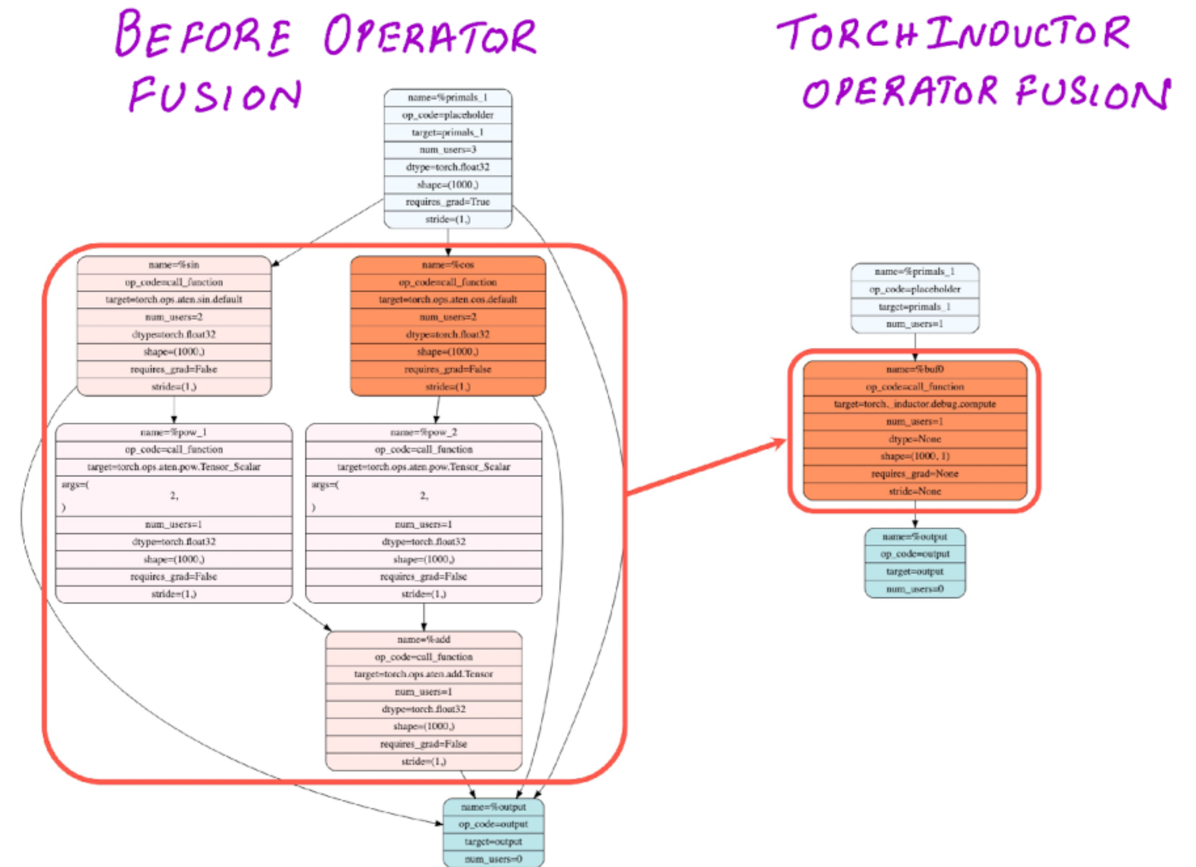

```
# Sharded training using fairscale
trainer = Trainer(devices=4, strategy='ddp_sharded')

# 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)`





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
```

Meme of the day