# SCALING MODEL TRAINING WITH MORE COMPUTE, HOW DO THEY DO IT?

# WHO AM I?

- Zachary Mueller
- Technical Lead for the 🤗 Accelerate project
- API design geek

# UNDERSTANDING GPU USAGE

- We can somewhat estimate the memory usage in vanilla full-fine-tuning of models
- Requires certain assumptions (that I'll be covering):
  - Adam optimizer
  - Batch size of 1

# UNDERSTANDING GPU USAGE

General estimate (`bert-base-cased`, 108M params):

- Each parameter is 4 bytes
- Backward ~= 2x the model size
- The optimizer step ~= 4x the model size (1x model, 1x gradients, 2x optimizer):

| dtype | Model | Gradients | Backward pass | Optimizer step | Highest |
|---|---|---|---|---|---|
| float32 | 413.18 MB | 413.18 MB | 826.36 MB | 1.61 GB | 1.61 GB |
| float16 | 413.18 MB* | 619.77 MB | 826.36 MB | 826.36 MB | 826.36 MB |

*All estimations were based off the Model Estimator Tool

# UNDERSTANDING GPU USAGE

This works fine for small models, we have cards with anywhere from 12-24GB of GPU memory (on the GPU-poor side).

But what happens as we scale?

Here's `llama-3-8B` (8.03B parameters)

| dtype | Model | Gradients | Backward pass | Optimizer step | Highest |
|---|---|---|---|---|---|
| float32 | 28.21 GB | 28.21 GB | 56.43 GB | 112.84 GB | 112.84 GB |
| float16 | 28.21 GB* | 42.32 GB | 56.43 GB | 56.43 GB | 56.43 GB |

Well, *I* don't have 56GB of GPU memory in a single card, let alone 112GB.

What can we do?

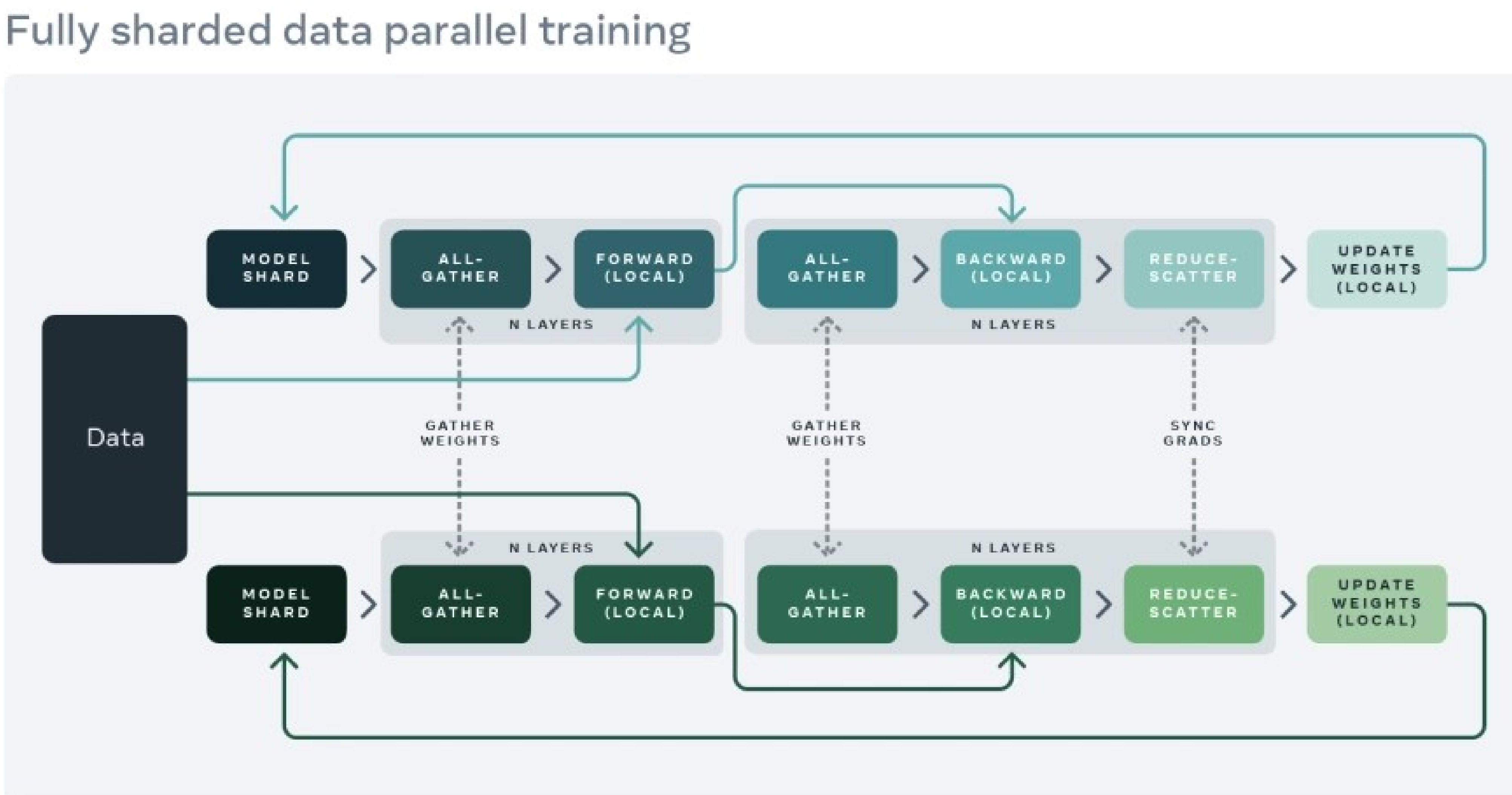
# DISTRIBUTED TRAINING

# KINDS OF TRAINING

- Single GPU:
  - No distributed techniques at play
- Distributed Data Parallelism (DDP):
  - A full copy of the model exists on each device, but data is chunked between each GPU
- Fully Sharded Data Parallelism (FSDP) & DeepSpeed (DS):
  - Split chunks of the model and optimizer states across GPUs, allowing for training bigger models on smaller (multiple) GPUs

# FULLY SHARDED DATA PARALLELISM

# FULLY SHARDED DATA PARALLELISM



# FSDP: GETTING PARAMETER SPECIFIC

- Different parameters can dicatate how much memory is needed for total GPU training across multiple GPUs
- These include how model weights are sharded, gradients, and more.
- I’ll cover some important ones I needed when doing a Full-Fine-Tune of Llama-3-8B *without PEFT* on 2x4090’s

# `sharding_strategy`

- Dictates the level of divving resources to perform
  - `FULL_SHARD`: Includes optimizer states, gradients, and parameters
  - `SHARD_GRAD_OP`: Includes optimizer states and gradients
  - `NO_SHARD`: Normal DDP
  - `HYBRID_SHARD`: Includes optimizer states, gradients, and parameters but each node has the full model

## `auto_wrap_policy`:

- How the model should be split
- Can be either `TRANSFORMER_BASED_WRAP` or `SIZE_BASED_WRAP`
- `TRANSFORMER`/`fsdp_transformers_layer_cls_to_wrap`:
  - Need to declare the layer
  - Generally `transformers` has good defaults
- `SIZE`/`fsdp_min_num_param`:
  - Number of total parameters in a shard

## `offload_params`:

- Offloads the parameters and gradients to the CPU if they can't fit into memory
- Allows you to train much larger models locally, but will be much slower

> Case: FFT of Llama-3-8B with `fsdp_offload_params` on 2x4090 GPUs was 72hrs, vs ~an hour or two when using 1xH100

# `cpu_ram_efficient_loading` AND `sync_module_states`

- Uses the idea behind big model inference/the `meta` device to load in the model to the GPU in a low-ram scenario
- Rather than needing `model_size` * `n_gpus` RAM, we can load the model on a single node and then send the weights directly to each shard when the time is right via `sync_module_states`

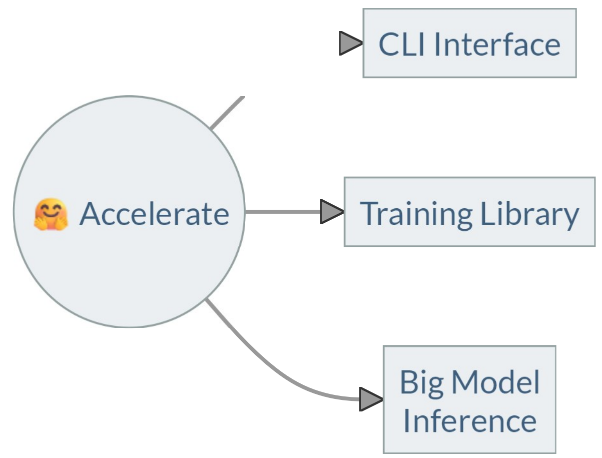
# TYING THIS TO 🤗 ACCELERATE

# TYING THIS TO 🤗 ACCELERATE

- So far we've covered the theory, but how do we put it into practice
- By using a library that's at the heart of the entire open-source ecosystem
    - Nearly all of 🤗
    - `axolotl`
    - `fastai`
    - `FastChat`
    - `lucidrains`
    - `kornia`

Are you using it and you don't even know?

# WHAT IS 🤗 ACCELERATE?



# A CLI INTERFACE

- `accelerate config`
  - Configure the environment
- `accelerate estimate-memory`
  - How to guess vRAM requirements
- `accelerate launch`
  - How to run your script

# LAUNCHING DISTRIBUTED TRAINING IS HARD

- ```
  python script.py
  ```
- ```
  torchrun --nnodes=1 --nproc_per_node=2 script.py
  ```
- ```
  deepspeed --num_gpus=2 script.py
  ```

How can we make this better?
  ```

# accelerate launch

```
accelerate launch script.py
```

# `accelerate config`

- Rely on `config.yaml` files
- Choose to either running `accelerate config` or write your own:

ddp_config.yaml

```yaml
compute_environment: LOCAL_MACHINE
distributed_type: MULTI_GPU
main_training_function: main
mixed_precision: bf16
num_machines: 1
num_processes: 8
```

fsdp_config.yaml

```yaml
compute_environment: LOCAL_MACHINE
distributed_type: FSDP
fsdp_config:
  fsdp_auto_wrap_policy: TRANSFORMER_BASED_WRAP
  fsdp_backward_prefetch: BACKWARD_PRE
  fsdp_cpu_ram_efficient_loading: true
  fsdp_forward_prefetch: false
  fsdp_offload_params: false
  fsdp_sharding_strategy: FULL_SHARD
  fsdp_state_dict_type: SHARDED_STATE_DICT
  fsdp_sync_module_states: true
  fsdp_use_orig_params: false
main_training_function: main
mixed_precision: bf16
num_machines: 1
num_processes: 8
```

# A TRAINING LIBRARY

# A TRAINING LIBRARY: THE CODE

```python
# For alignment purposes
for batch in dataloader:
    optimizer.zero_grad()
    inputs, targets = batch
    inputs = inputs.to(device)
    targets = targets.to(device)
    outputs = model(inputs)
    loss = loss_function(outputs, targets)
    loss.backward()
    optimizer.step()
    scheduler.step()
```

```python
from accelerate import Accelerator
accelerator = Accelerator()
dataloader, model, optimizer scheduler = (
    accelerator.prepare(
        dataloader, model, optimizer, scheduler
    )
)

for batch in dataloader:
    optimizer.zero_grad()
    inputs, targets = batch
    # inputs = inputs.to(device)
    # targets = targets.to(device)
    outputs = model(inputs)
    loss = loss_function(outputs, targets)
    accelerator.backward(loss) # loss.backward()
    optimizer.step()
    scheduler.step()
```

# A TRAINING LIBRARY: HOW SCALING WORKS

- Accelerate's DataLoaders and schedulers work off of a sharding mindset
- Rather than repeating the same data across n nodes, we instead split it
- Speeds up training linearly
- Given a batch size of 16 on a single GPU, to recreate this across 8 GPUs you would use a batch size of 2
- This also means the scheduler will be stepped n GPUs at a time per "global step"

# A TRAINING LIBRARY: MIXED PRECISION

- This may be a bit different than your “normal” idea of mixed precision.
- We do **not** convert the model weights to BF16/FP16
- Instead we **wrap the forward pass** with `autocast` to convert the gradients automatically
- This preserves the original precision of the weights, which leads to stable training and better fine-tuning later on.
- **If you use `.bf16()` weights, you are STUCK in bf16 perminantly**

# A TRAINING LIBRARY: MIXED PRECISION

- Let's tie that back up to the model estimator with neat tools like NVIDIA's TransformerEngine

| Optimization Level | Computation (GEMM) | Comm | Weight | Master Weight | Weight Gradient | Optimizer States |
|---|---|---|---|---|---|---|
| FP16 AMP | FP16 | FP32 | FP32 | N/A | FP32 | FP32+FP32 |
| Nvidia TE | FP8 | FP32 | FP32 | N/A | FP32 | FP32+FP32 |
| MS-AMP O1 | FP8 | FP8 | FP16 | N/A | FP8 | FP32+FP32 |
| MS-AMP O2 | FP8 | FP8 | FP16 | N/A | FP8 | FP8+FP16 |
| MS-AMP O3 | FP8 | FP8 | FP8 | FP16 | FP8 | FP8+FP16 |

# DEEPSPEED VS FULLY SHARDED DATA PARALLELISM

- Extremely similar, however mostly used different naming conventions for items and slight tweaks in the implementation

| Framework | Model Loading (`torch_dtype`) | Mixed Precision | Preparation (Local) | Training | Optimizer (Local) |
|---|---|---|---|---|---|
| FSDP | bf16 | default (none) | bf16 | bf16 | bf16 |
| FSDP | bf16 | bf16 | fp32 | bf16 | fp32 |
| DeepSpeed | bf16 | bf16 | fp32 | bf16 | fp32 |

To learn more, check out the documentation or join my office hours

# KEY TAKEAWAYS:

- You can scale out training with `accelerate`, FSDP, and DeepSpeed across multiple GPUs to train bigger models
- Techniques like `FP8` can help speed up training some and reduce computational overhead
- Comes at a cost of end-precision and locking model weights for futher fine-tunes if not careful

# SOME HANDY RESOURCES

- 🤗 Accelerate documentation
- Launching distributed code
- Distributed code and Jupyter Notebooks
- Migrating to 🤗 Accelerate easily
- Big Model Inference tutorial
- DeepSpeed and 🤗 Accelerate
- Fully Sharded Data Parallelism and 🤗 Accelerate
- FSDP vs DeepSpeed In-Depth