OpenBioML PyTorch Lightning workshop (2/2)

Adrian Wälchli, Research Engineer
Luca Antiga, CTO

OpenBioML PyTorch Lightning workshop

Session 1: Thu 23 Feb, 3pm ET

Session 2: Thu 2 Mar, 3pm ET

https://harvard.zoom.us/j/97375262666

OpenBioML PyTorch Lightning workshop

Session 1

- Intro to PyTorch Lightning + Fabric
- Hands-on: raw PyTorch -> Fabric -> PyTorch Lightning Trainer
- A look into OpenFold

By the end of Session 1 we saw how to build a model with PyTorch Lightning and Fabric and train it. Distributed. On a SLURM cluster.

OpenBioML PyTorch Lightning workshop

Session 2

- Intro to core distributed concepts and what's new in PyTorch 2.0
- Hands-on: how to debug and optimize performance, single node and distributed, running benchmarks
- More on OpenFold

By the end of Session 2 you will know how to make sure you are setting up your training correctly and verify you are leveraging your hardware the best.

Join us here

discord	discord.gg/MWAEvnC5fU
forums	lightning.ai/forums
twitter	@LightningAI

https://linktr.ee/lightningai

PyTorch Lightning

You do the science, we do the engineering

33 million+
DOWNLOADS

780+
CONTRIBUTORS

pip install lightning 21,512

13,000+
PROJECTS USING LIGHTNING

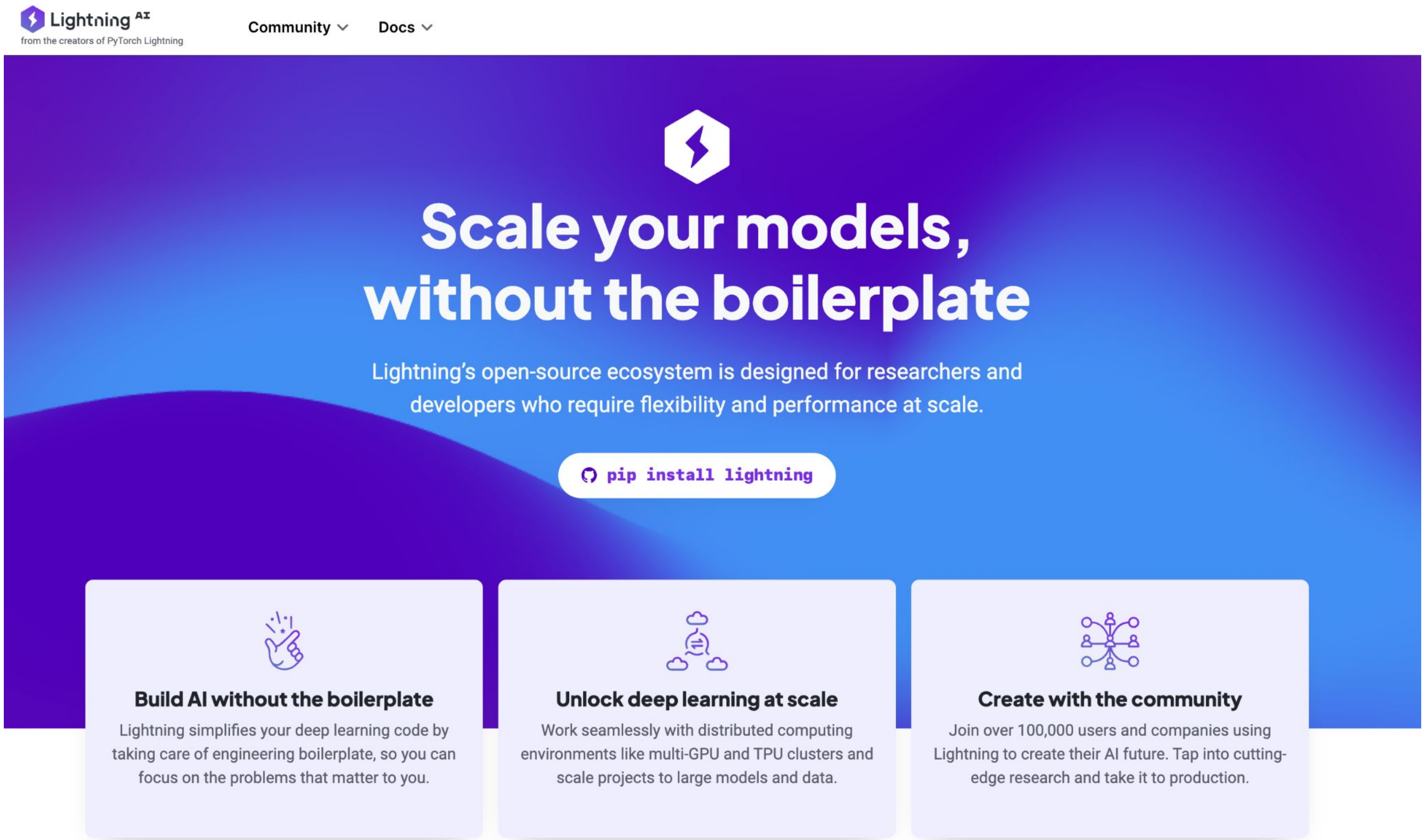
6,000+
SLACK MEMBERS

10,000+ ORGANIZATIONS BUILD WITH LIGHTNING

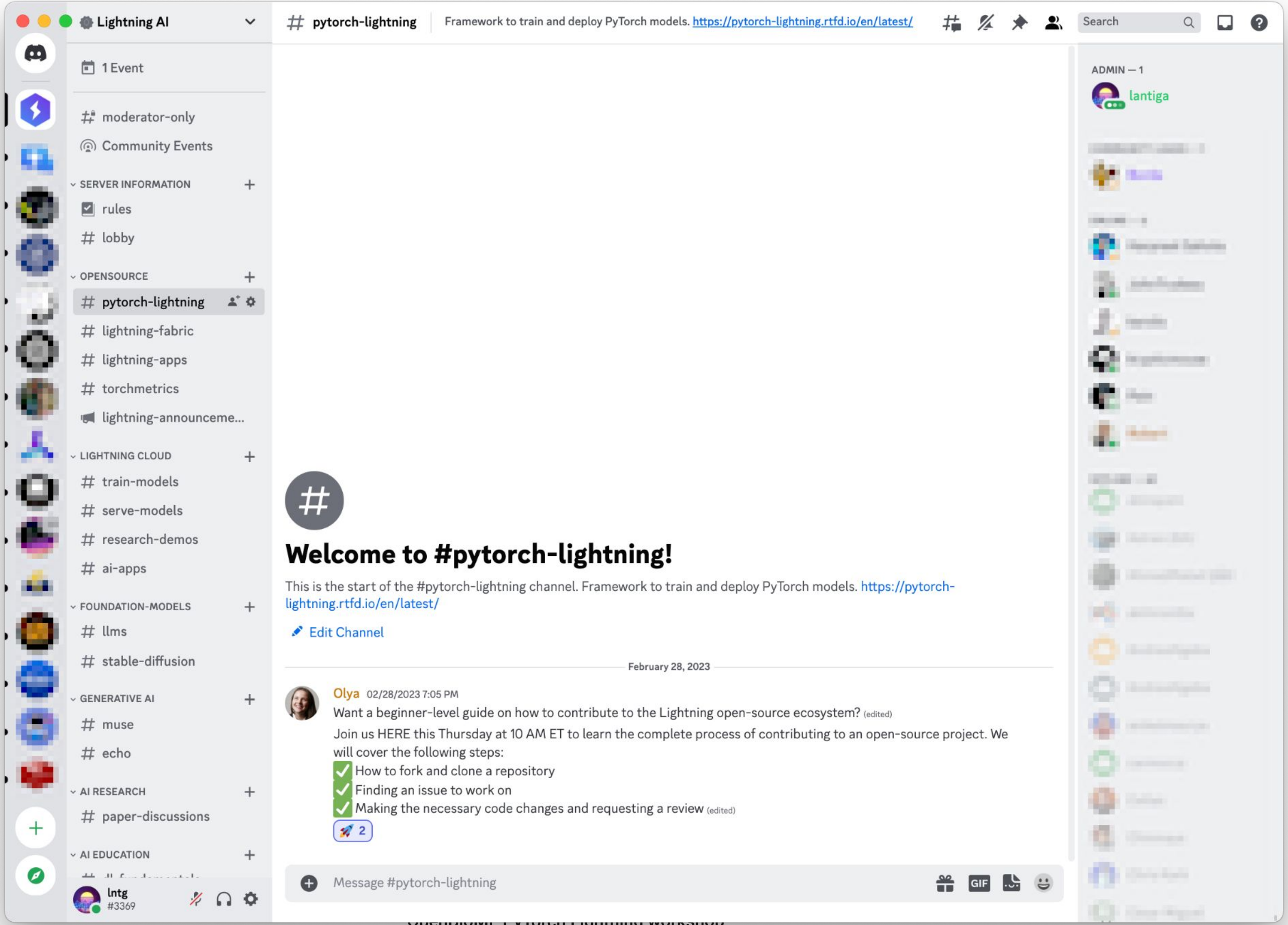
amazon NVIDIA Meta Microsoft habana An Intel Company MARS

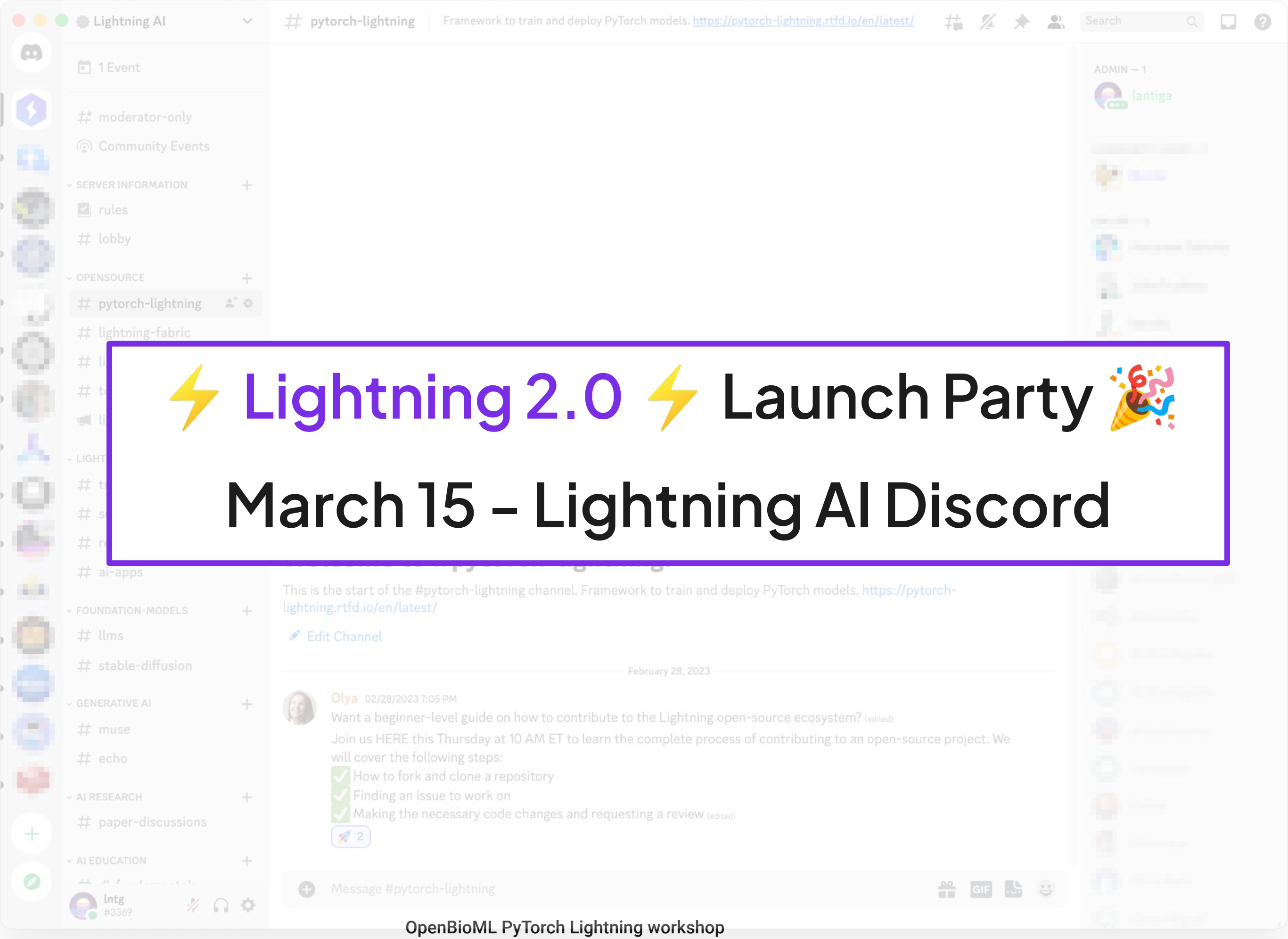
https://lightning.ai

lightning.ai/pages/open-source/



discord.gg/gxd2Abx5





⚡ Lightning 2.0 ⚡ Launch Party 🎉

March 15 – Lightning AI Discord

Intro: Scaling out and going fast

pytorch-lightning.readthedocs.io

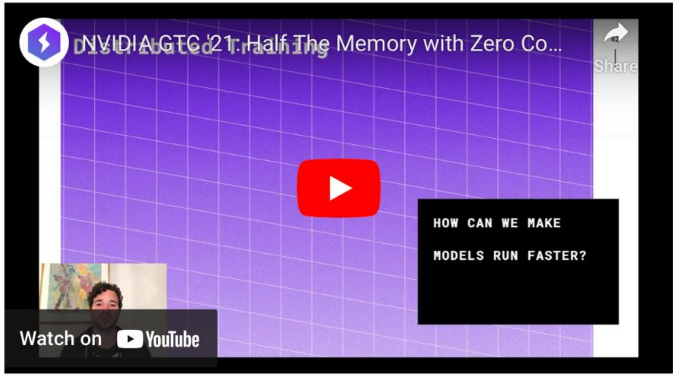
TRAIN 1 TRILLION+ PARAMETER MODELS

When training large models, fitting larger batch sizes, or trying to increase throughput using multi-GPU compute, Lightning provides advanced optimized distributed training strategies to support these cases and offer substantial improvements in memory usage.

Note that some of the extreme memory saving configurations will affect the speed of training. This Speed/Memory trade-off in most cases can be adjusted.

Some of these memory-efficient strategies rely on offloading onto other forms of memory, such as CPU RAM or NVMe. This means you can even see memory benefits on a **single GPU**, using a strategy such as DeepSpeed ZeRO Stage 3 Offload.

Check out this amazing video explaining model parallelism and how it works behind the scenes:



Choosing an Advanced Distributed GPU Strategy

Lightning AI

PyTorch Lightning

Organized PyTorch

LightningModule

Trainer

```
import lightning as L
from torch import n, optim

encoder = nn.Sequential(nn.Linear(28 * 28, 64), nn.ReLU(), nn.Linear(64, 3))
decoder = nn.Sequential(nn.Linear(3, 64), n.ReLU() , nn. Linear (64, 28 * 28))

class LitAutoEncoder(L.LightningModule):
    def _init__(self, encoder, decoder):
        super()._init__()
        self.encoder = encoder
        self.decoder = decoder

    def training_step(self, batch, batch_idx):
        X, y = batch
        X = x. view(x.size(0), -1)
        z = self.encoder(x)
        x_hat = self.decoder (z)
        loss = nn. functional.mse_loss(x_hat, x)
        self.log("train_loss", loss) return loss

    def configure_optimizers( self):
        optimizer = optim.Adam(self.parameters(), lr=le-3)
        return optimizer

autoencoder = LitAutoEncoder(encoder, decoder)
dataset = MIST(os.getcwd(), download=True, transform=ToTensor())

train_loader = utils.data.DataLoader (dataset)

trainer = L.Trainer(limit_train_batches=100, max_epochs=1)
trainer.fit(model=autoencoder, train_dataloaders=train_loader)
```

PyTorch Lightning

Accelerators

GPU, TPU, HPU, IPU, MPS

Strategies

DDP, FSDP, DeepSpeed, Colossal AI

Precision

Callbacks

```python
# train on 4 GPUs
trainer = Trainer(
     devices=4,
     accelerator="gpu"
)

# train 1B+ parameter models with Deepspeed/fsdp
trainer = Trainer(
    devices=4,
    accelerator="gpu",
    strategy="deepspeed_stage_2",
    precision=16
)

# 20+ helpful flags for rapid idea iteration
trainer = Trainer(
    max_epochs=10,
    min_epochs=5,
    overfit_batches=1
)

# access the latest state of the art techniques
trainer = Trainer(callbacks=[StochasticWeightAveraging(...)])
```

Fabric

3 Your code

1 Fabric object

2 Setup model, optimizer, dataloader

```
import lightning as L

def train(fabric, model, optimizer, dataloader):
    # Training loop
    model.train()
    for epoch in range(num_epochs):
        for i, batch in enumerate(dataloader):
            ...

def main():
    # (Optional) Parse command line options
    args = parse_args()

    # Configure Fabric
    fabric = L.Fabric(..., strategy="deepspeed")

    # Instantiate objects
    model = ...
    optimizer = ...
    train dataloader = ...

    # Set up objects
    model, optimizer = fabric.setup(model, optimizer)
    train_dataloader = fabric.setup_dataloaders(train_dataloader)

    # Run training loop
    train(fabric, model, optimizer, train_dataloader)

if __name__ == "__main__":
    main()
```


Lightning AI

Scaling out

```
fabric = Fabric(accelerator="gpu", strategy="ddp", devices=8, num_nodes=4)
```

```
trainer = Trainer(accelerator="gpu", strategy="ddp", devices=8, num_nodes=4)
```

Scaling out

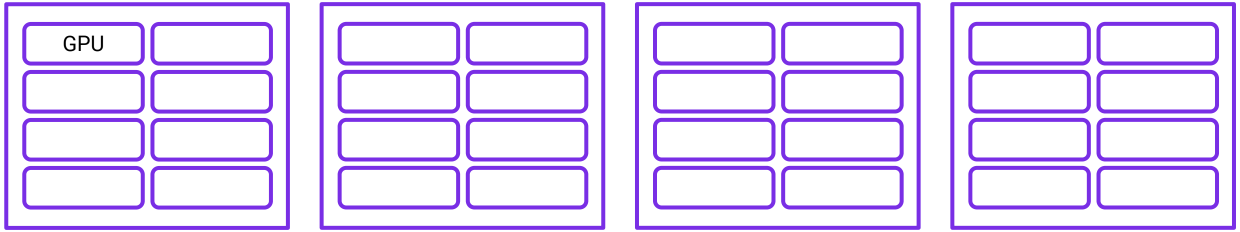
```
fabric = Fabric(accelerator="gpu", strategy="ddp", devices=8, num_nodes=4)
```

```
trainer = Trainer(accelerator="gpu", strategy="ddp", devices=8, num_nodes=4)
```

Scaling out

```
fabric = Fabric(accelerator="gpu", strategy="ddp", devices=8, num_nodes=4)
```

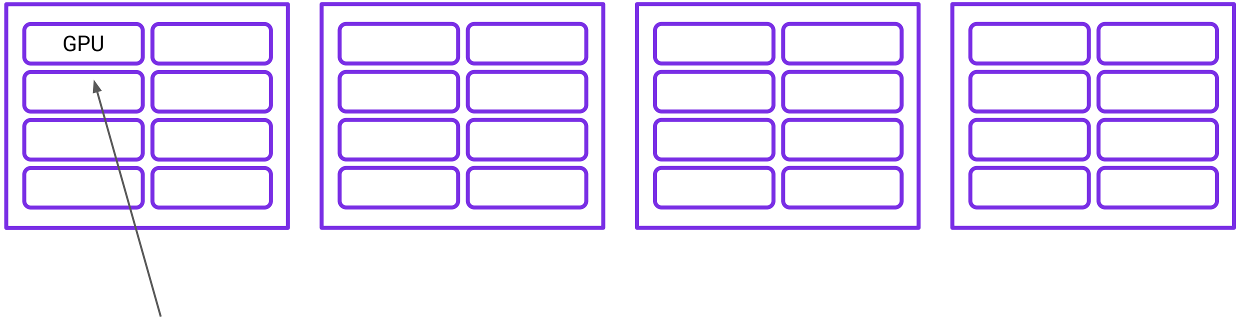
```
trainer = Trainer(accelerator="gpu", strategy="ddp", devices=8, num_nodes=4)
```



Scaling out

```
fabric = Fabric(accelerator="gpu", strategy="ddp", devices=8, num_nodes=4)
```

```
trainer = Trainer(accelerator="gpu", strategy="ddp", devices=8, num_nodes=4)
```

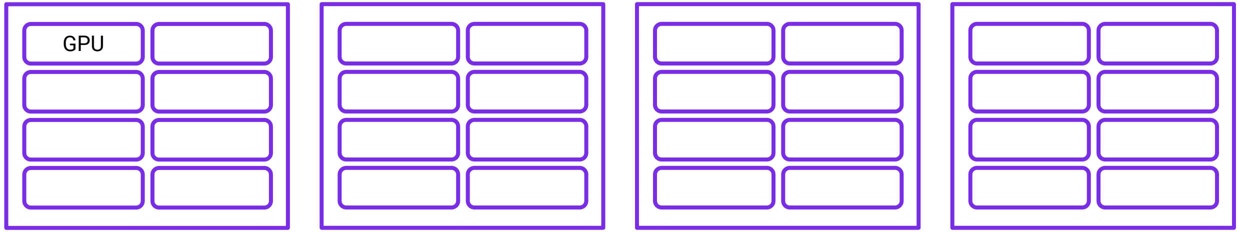


Think one process per accelerator

Under the hood

```
MASTER_PORT: free port on machine with NODE_RANK 0
MASTER_ADDR: address of NODE_RANK 0 node
WORLD_SIZE: the total number of GPUs/processes
NODE_RANK: id of the node in the cluster
```

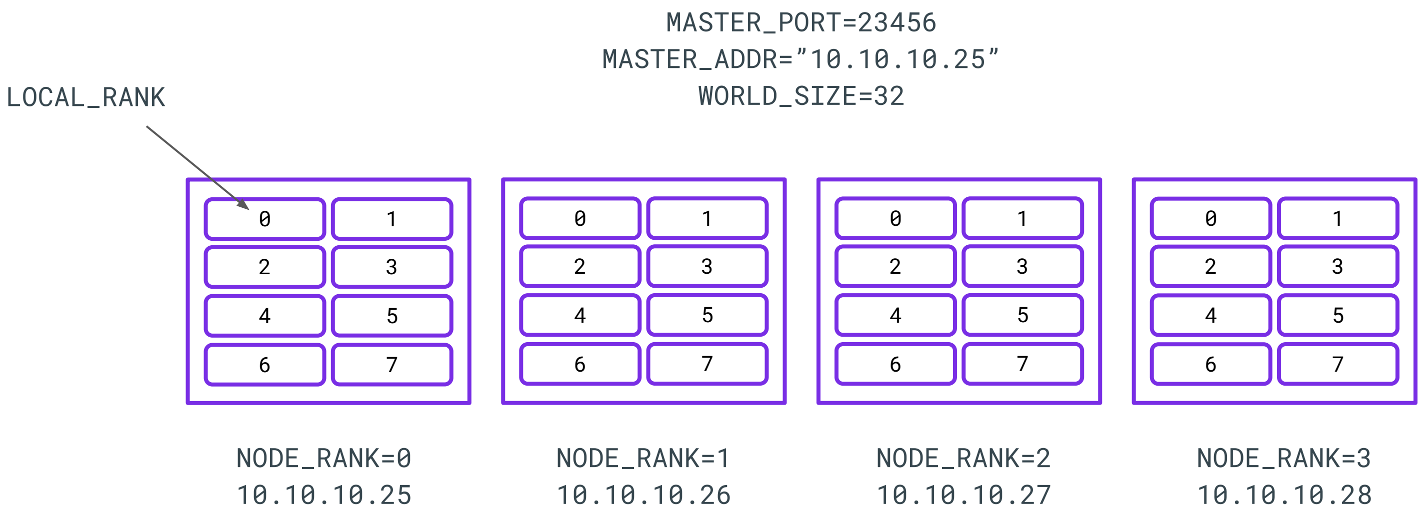
MASTER_PORT=23456
MASTER_ADDR=”10.10.10.25”
WORLD_SIZE=32



NODE_RANK=0	NODE_RANK=1	NODE_RANK=2	NODE_RANK=3
10.10.10.25	10.10.10.26	10.10.10.27	10.10.10.28

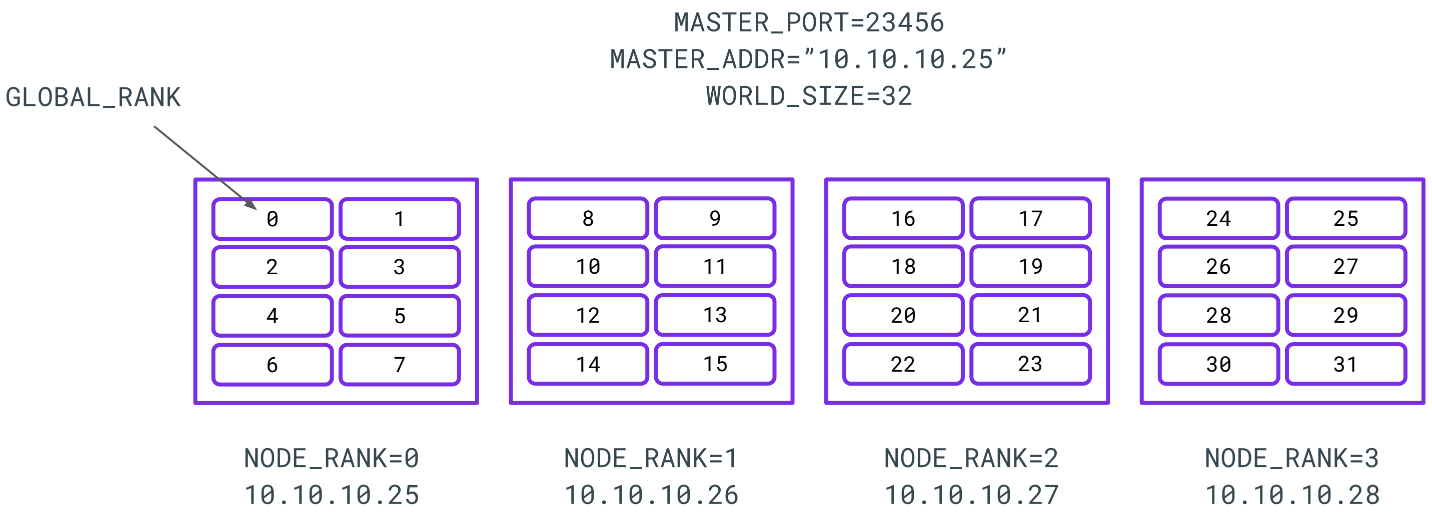
Under the hood

```
WORLD_SIZE: the total number of GPUs/processes
NODE_RANK: id of the node
LOCAL_RANK: id of the process in each node
GLOBAL_RANK: unique id of the process
```



Under the hood

```
WORLD_SIZE: the total number of GPUs/processes
NODE_RANK: id of the node
LOCAL_RANK: id of the process in each node
GLOBAL_RANK: unique id of the process
```

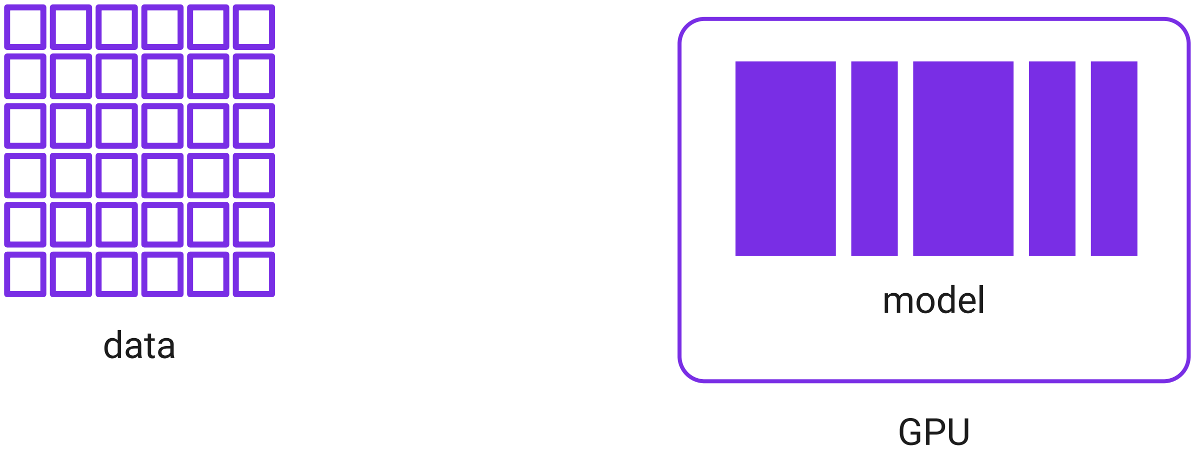


In Lightning / Fabric

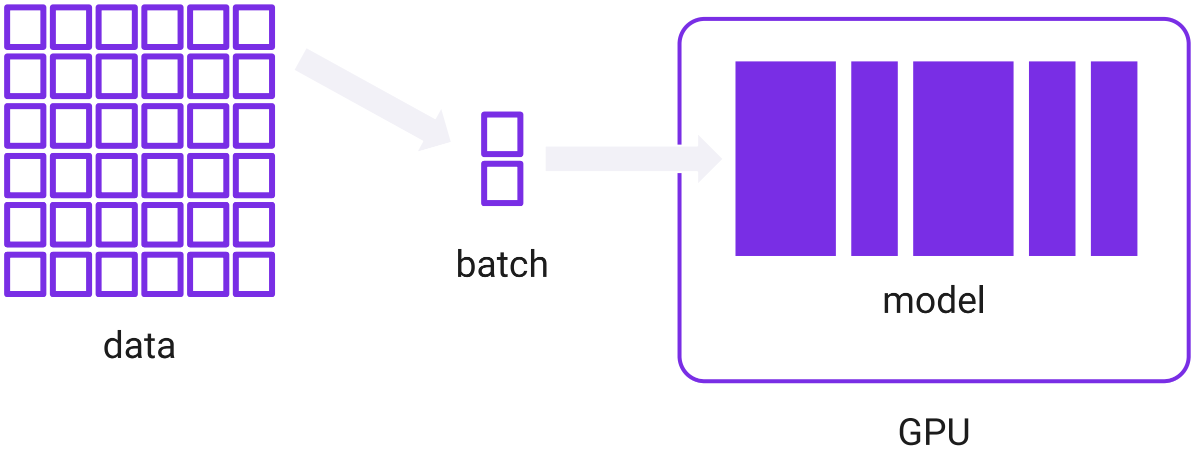
```python
from lightning.fabric import Fabric

# Devices and num nodes determine how many processes there are
fabric = Fabric(devices=8, num_nodes=4)
fabric.launch()
```

```python
# The total number of processes running across all devices and nodes
fabric.world_ size  # 4 * 8 = 32

# The global index of the current process across all devices and nodes
fabric.global_rank  # -> {0, 1, 2, 3, 4, ..., 31}

# The index of the current process among the processes running on the
local node
fabric.local_rank   # -> {0, 1, 2, 3, 4, 5, 6, 7}

# The index of the current node
fabric.node_rank    # -> {0, 1, 2, 3}

# Do something only on rank 0
if fabric.global_rank == 0:
    ...
```

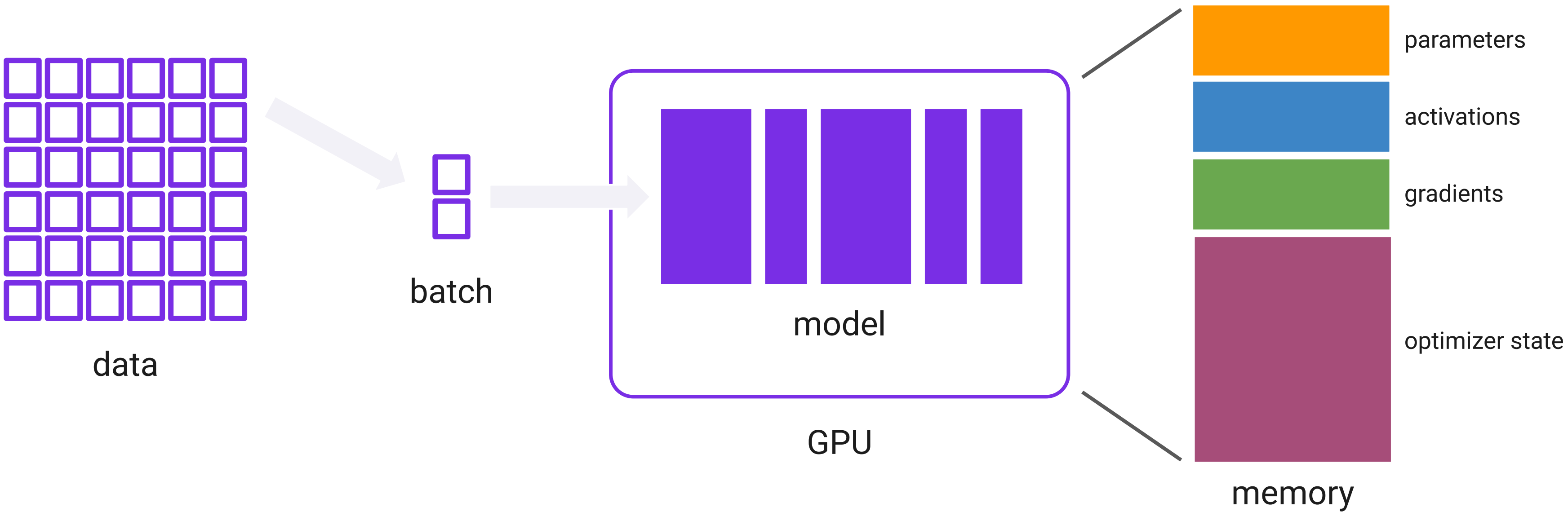
Scaling out



Scaling out

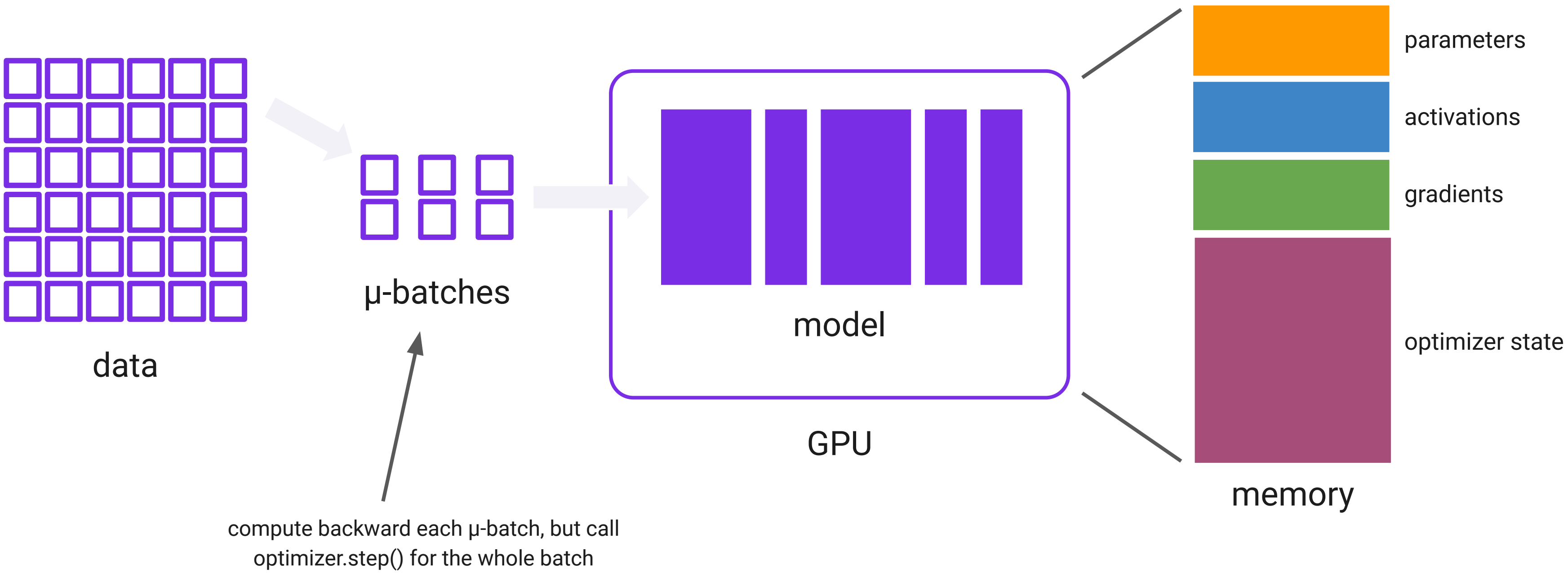


Scaling out

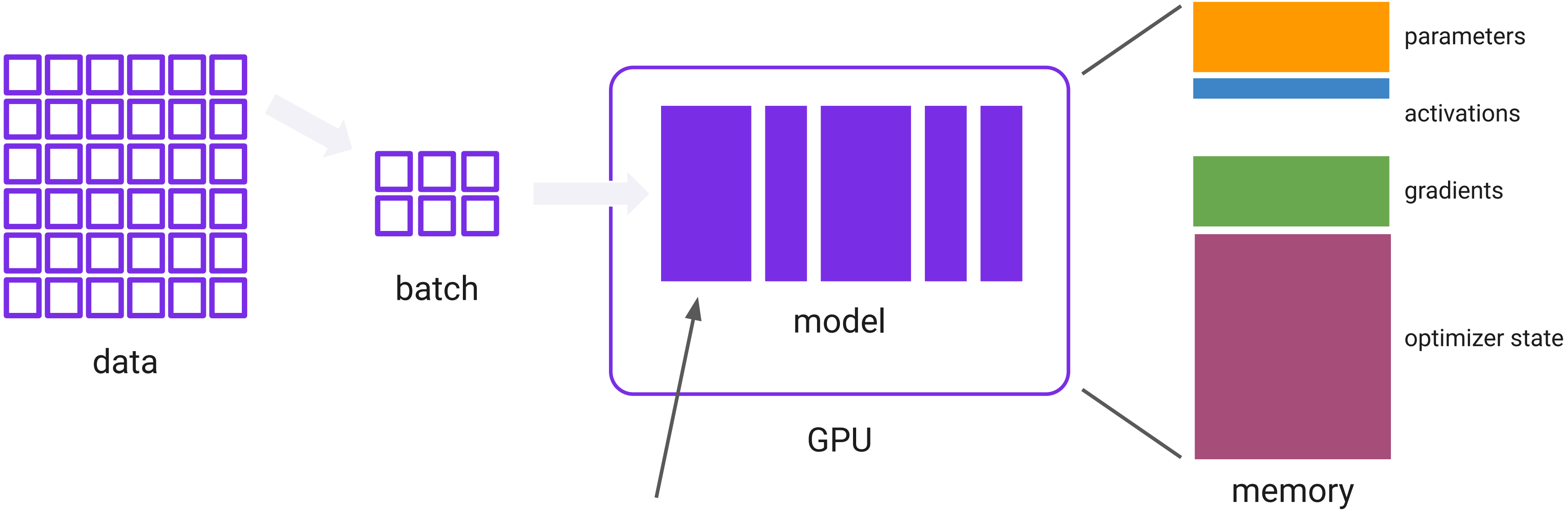


data
batch
model
GPU
parameters
activations
gradients
optimizer state
memory

Gradient accumulation

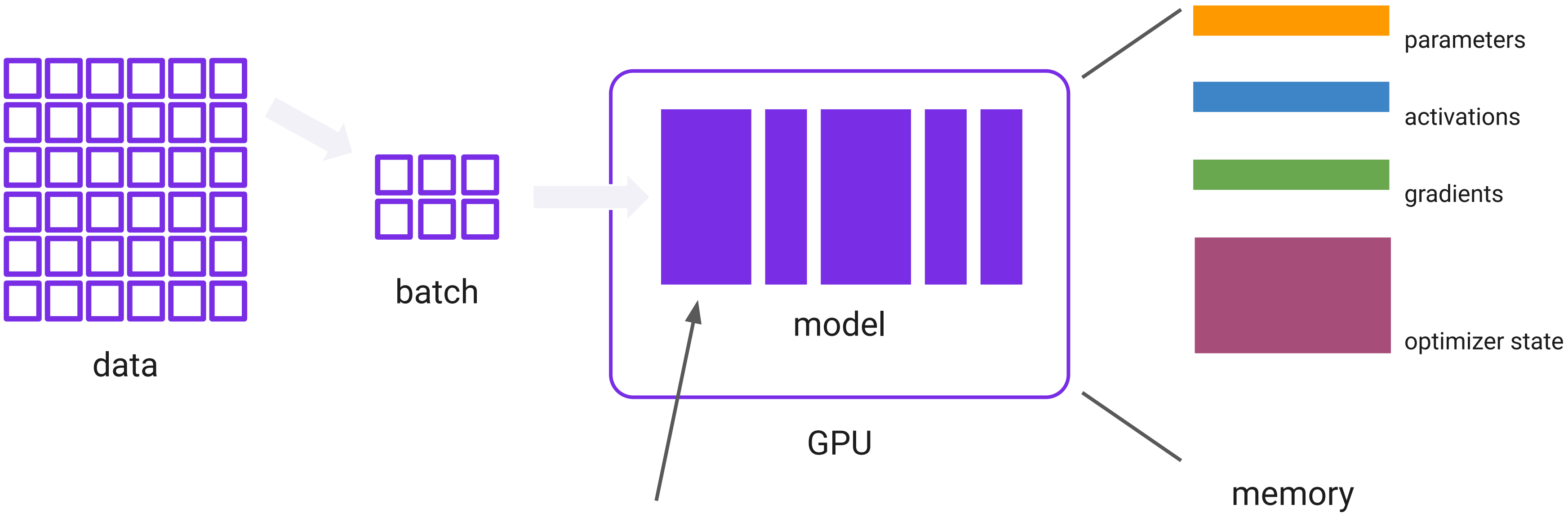


Activation/Gradient checkpointing



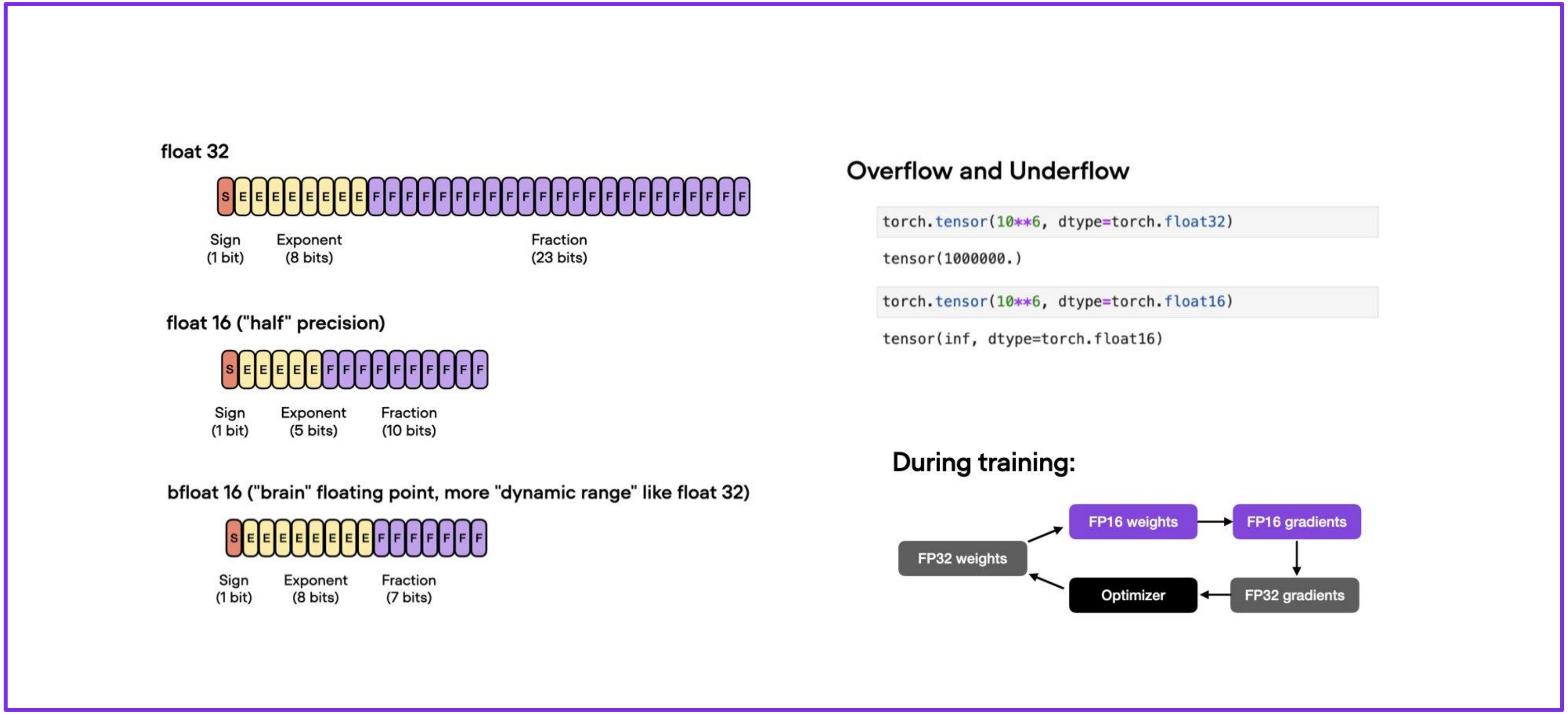
avoid storing activations during forward, only save (checkpoint) a few of them in-between to avoid recomputing too much

Mixed precision



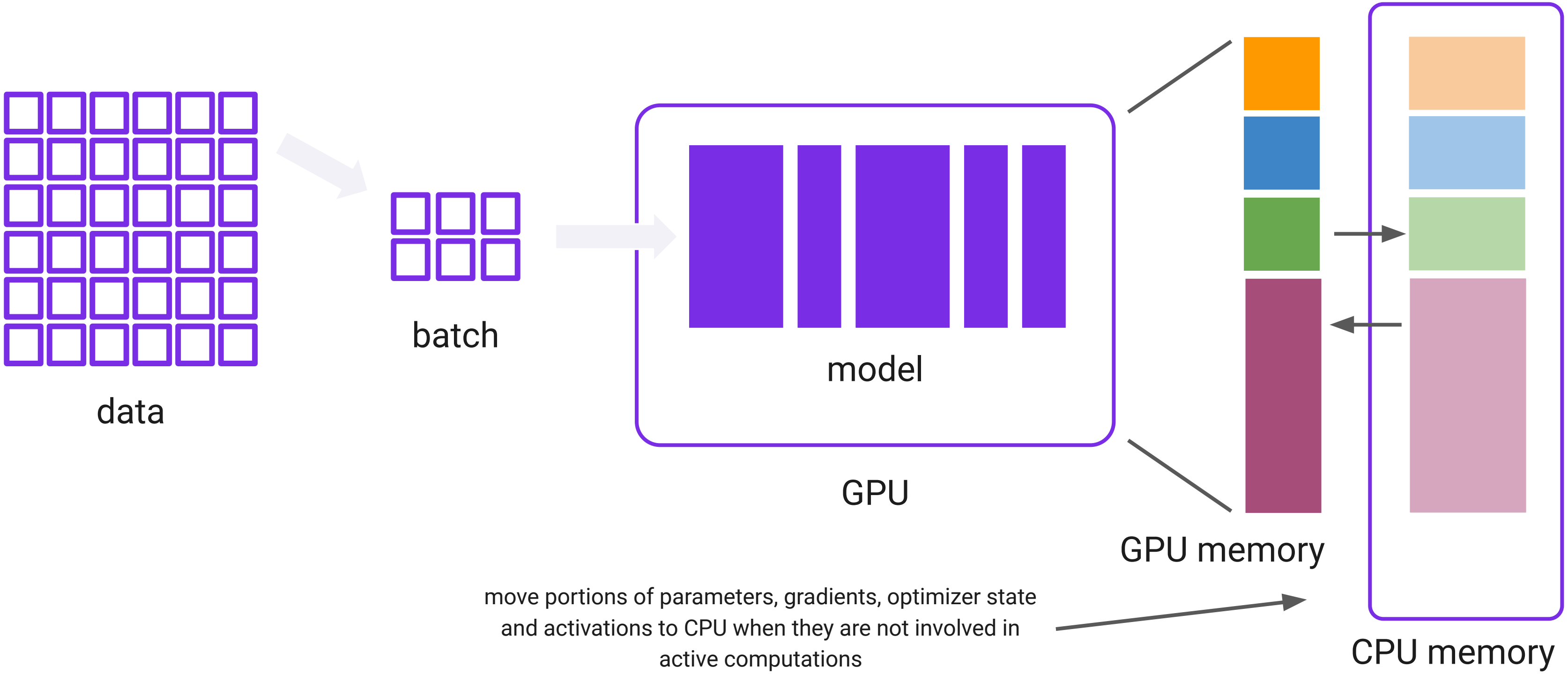
reduce numerical precision (32 -> 16-bit) so that memory footprint is reduced; normally full precision is automatically restored at computation time

Mixed precision



Sebastian Rashka, https://lightning.ai/pages/courses/deep-learning-fundamentals/

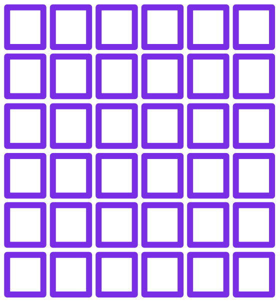
Offloading



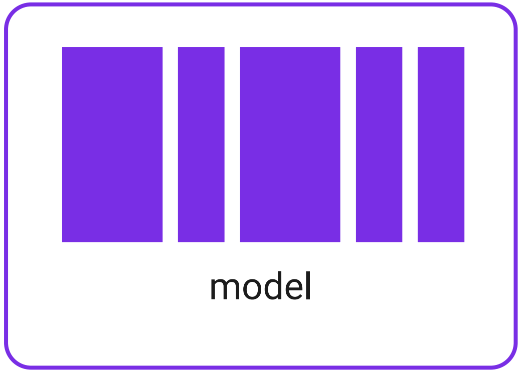
move portions of parameters, gradients, optimizer state and activations to CPU when they are not involved in active computations

Lightning AI

Scaling out

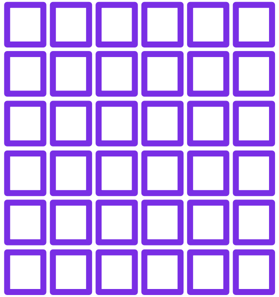


data



GPU

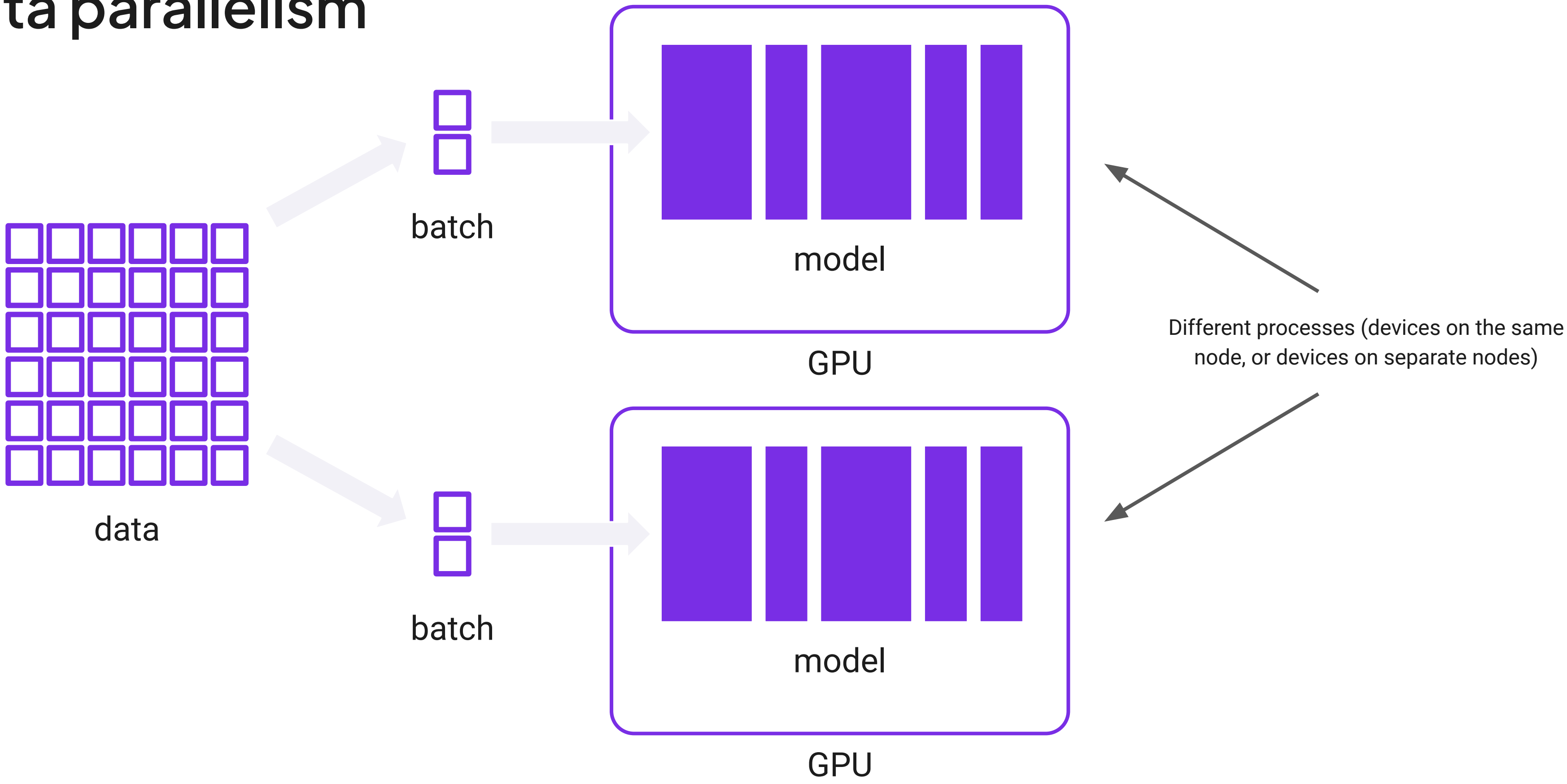
Scaling out



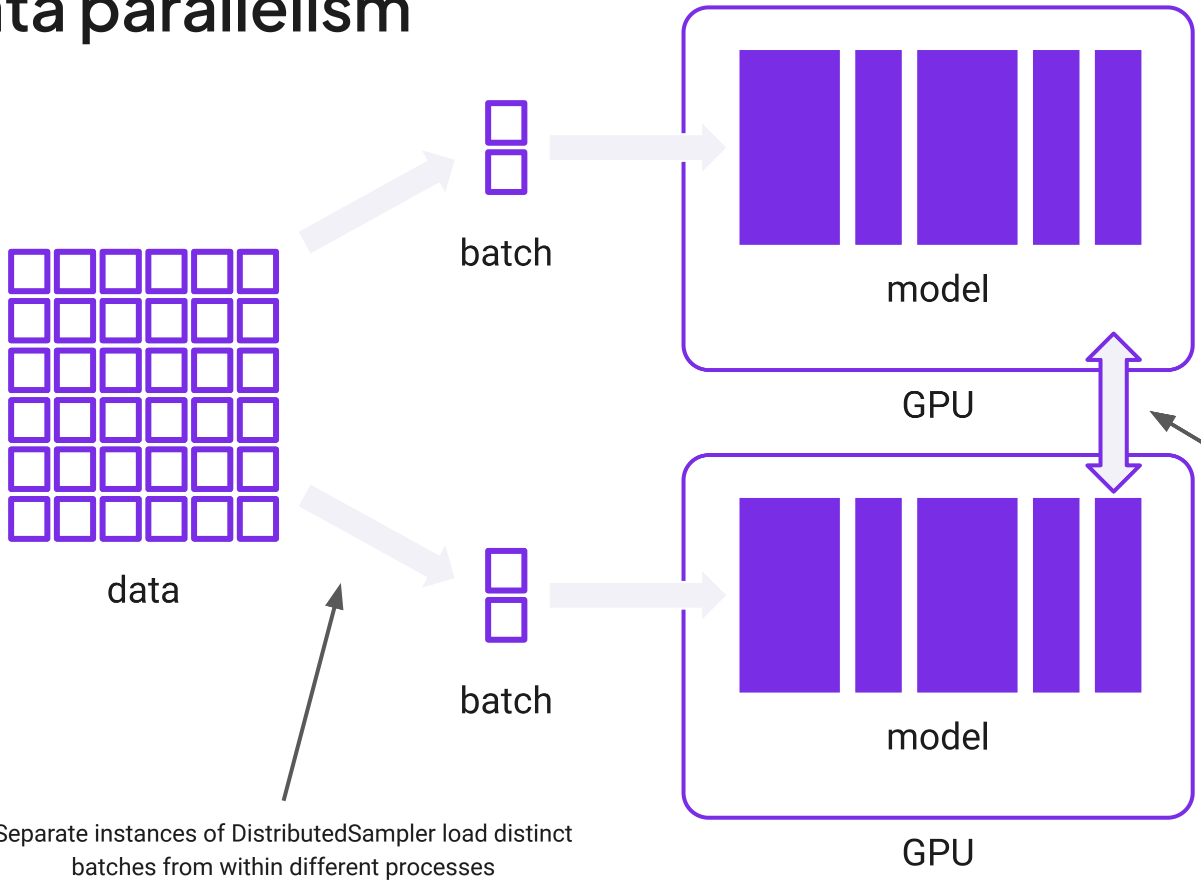
data

GPU

Data parallelism



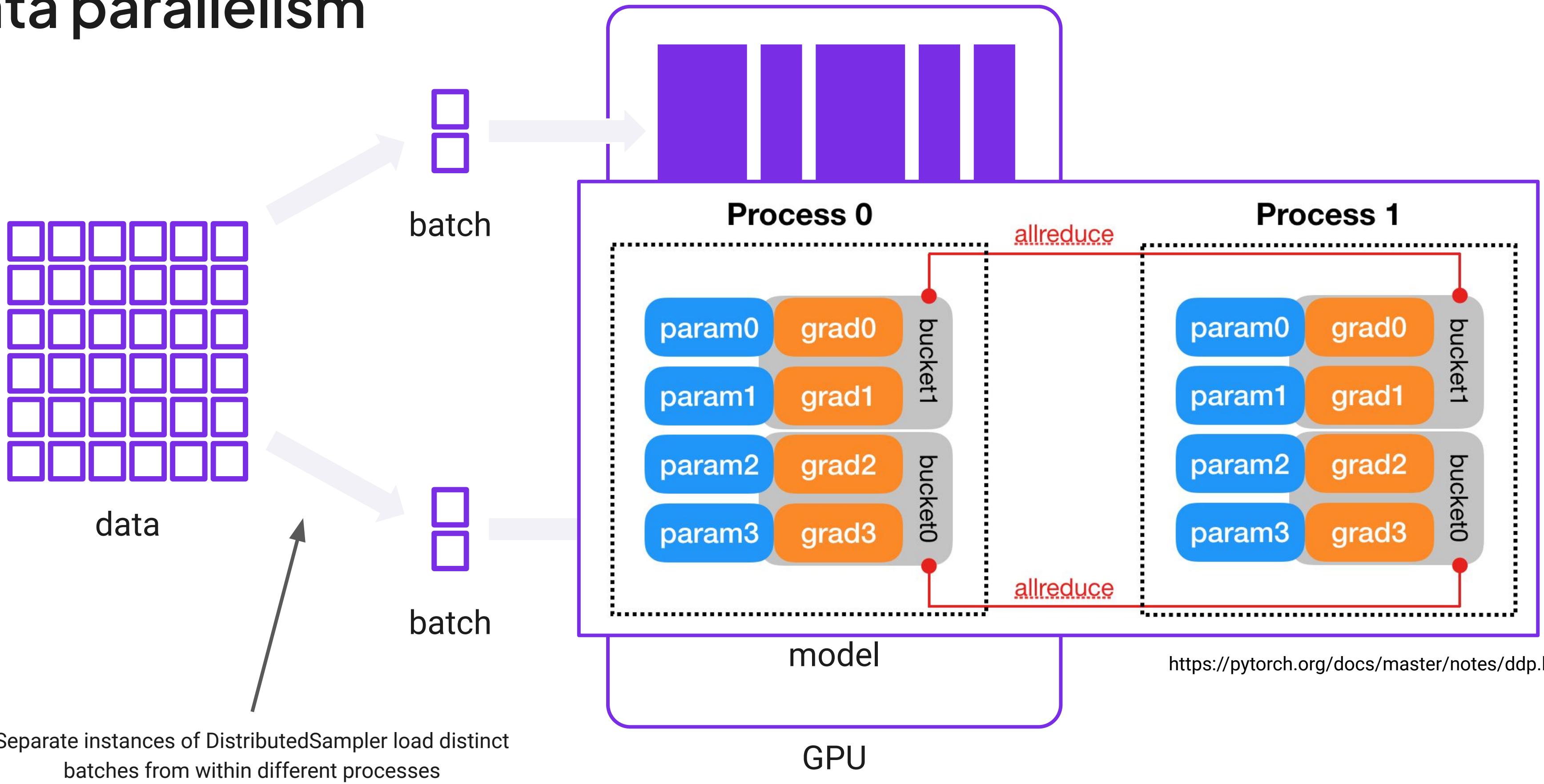
Data parallelism



Gradients need to be synchronized across processes during the backward pass

Separate instances of DistributedSampler load distinct batches from within different processes

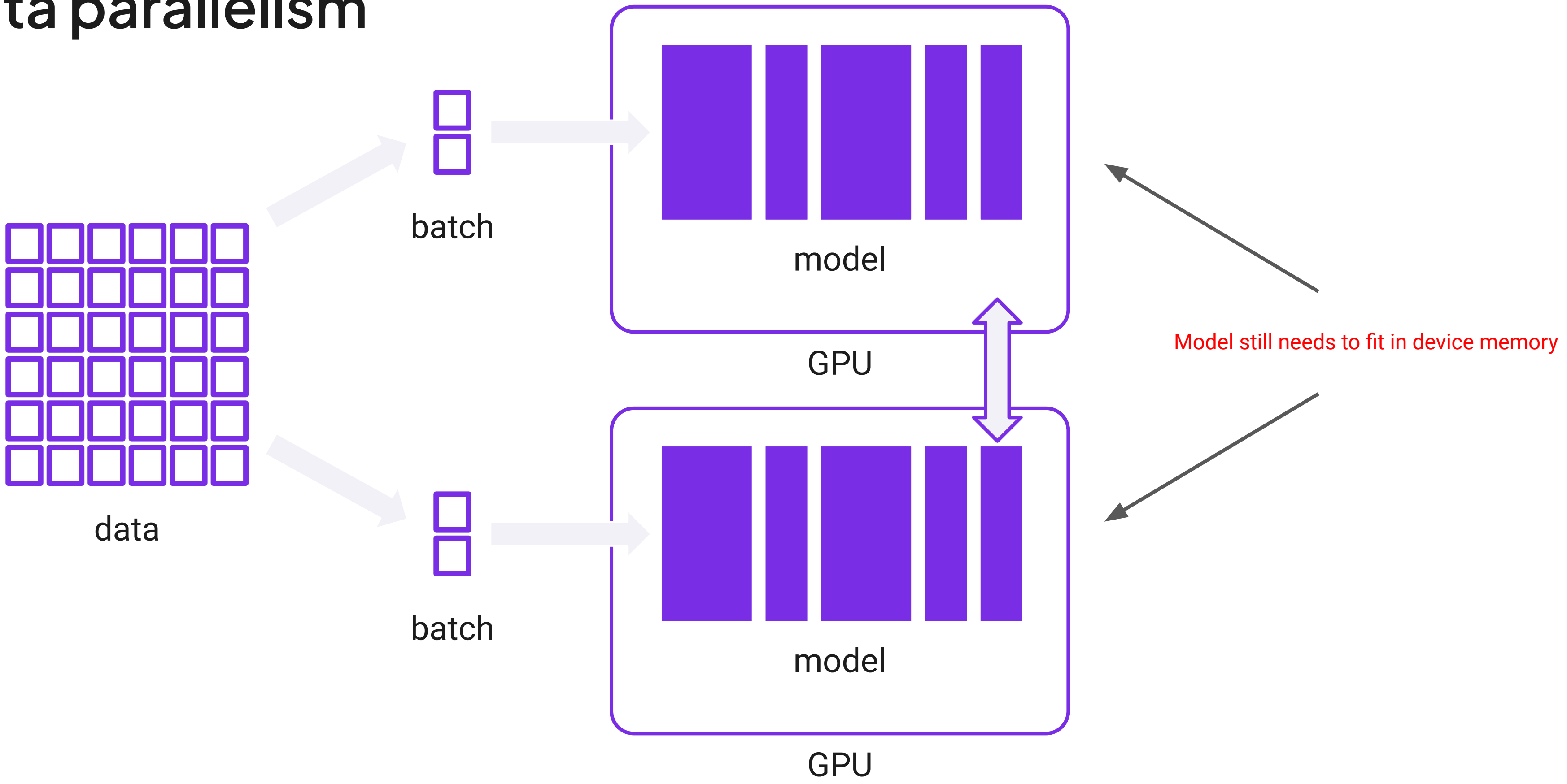
Data parallelism



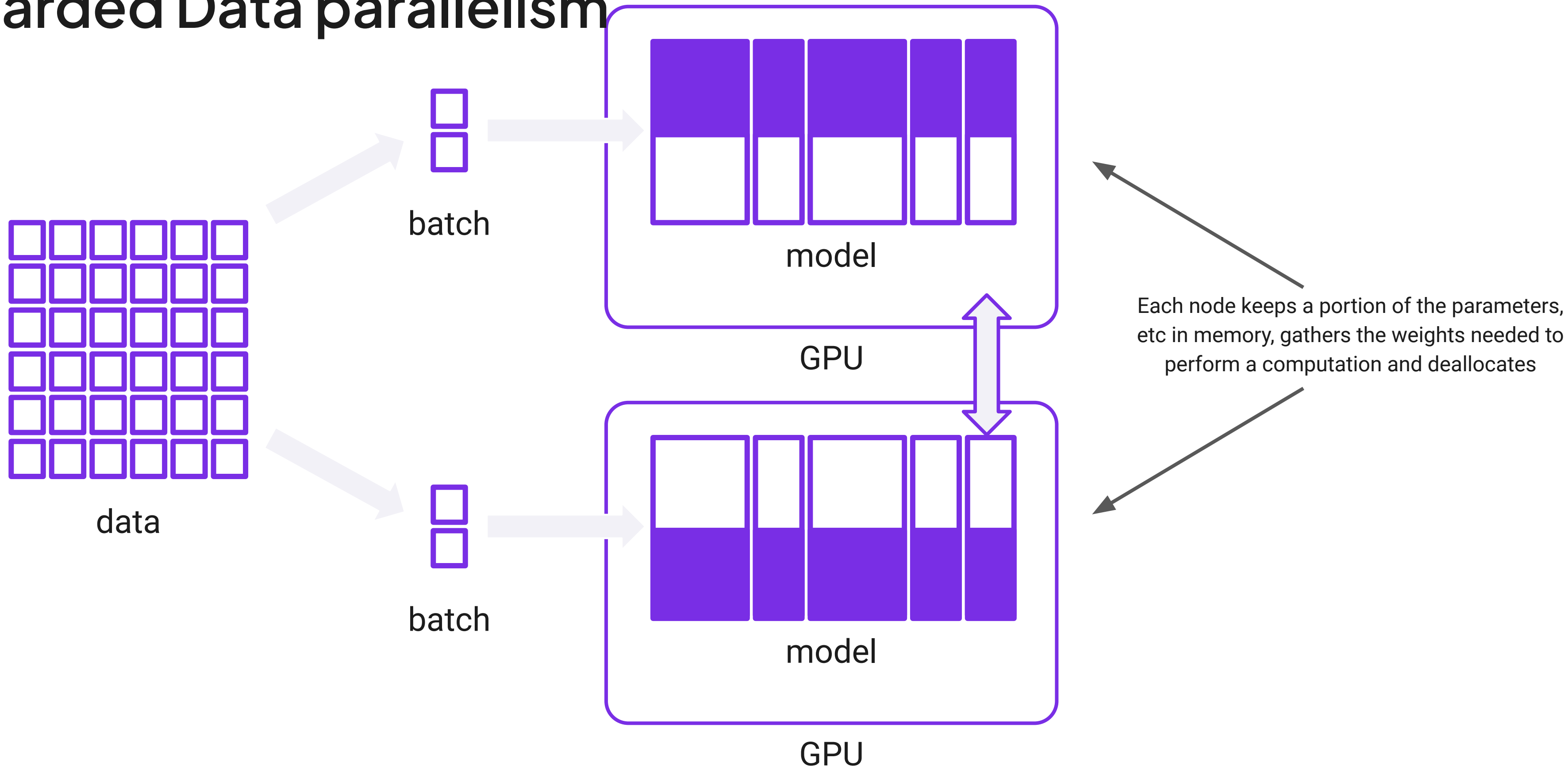
https://pytorch.org/docs/master/notes/ddp.html

Separate instances of DistributedSampler load distinct batches from within different processes

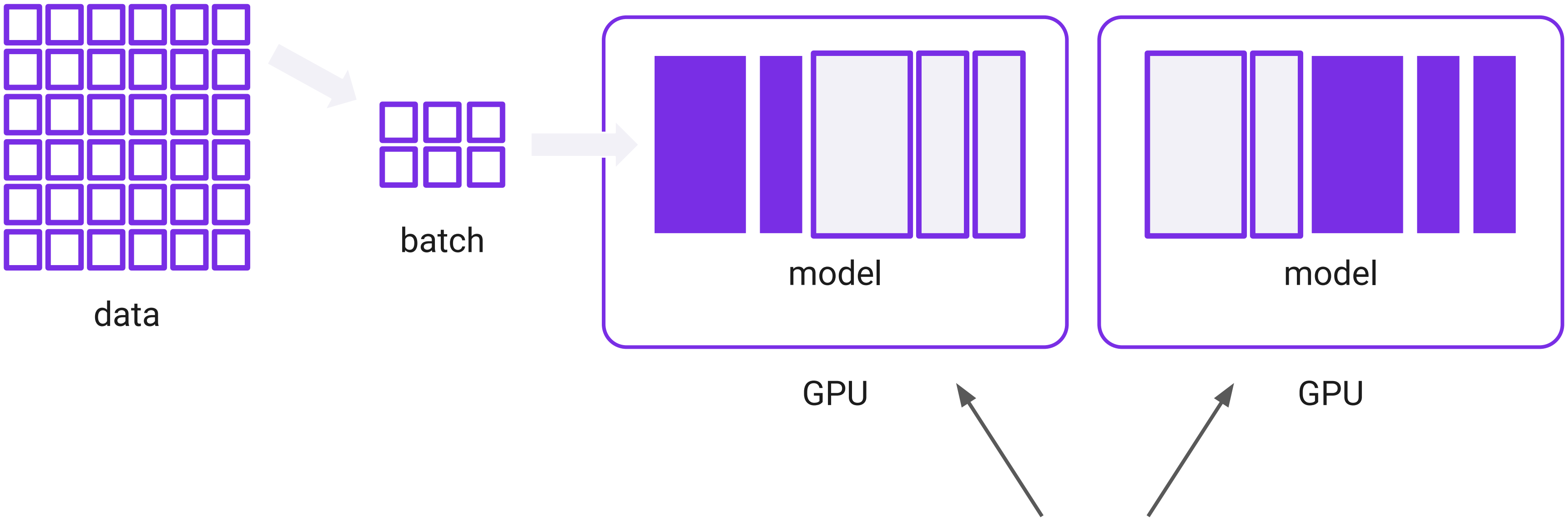
Data parallelism



Sharded Data parallelism



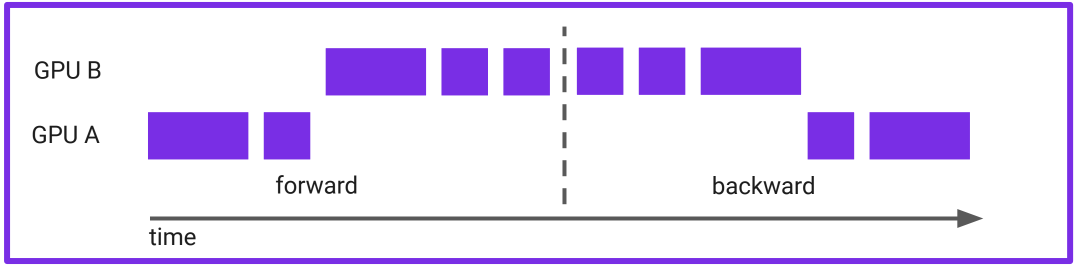
Model parallelism

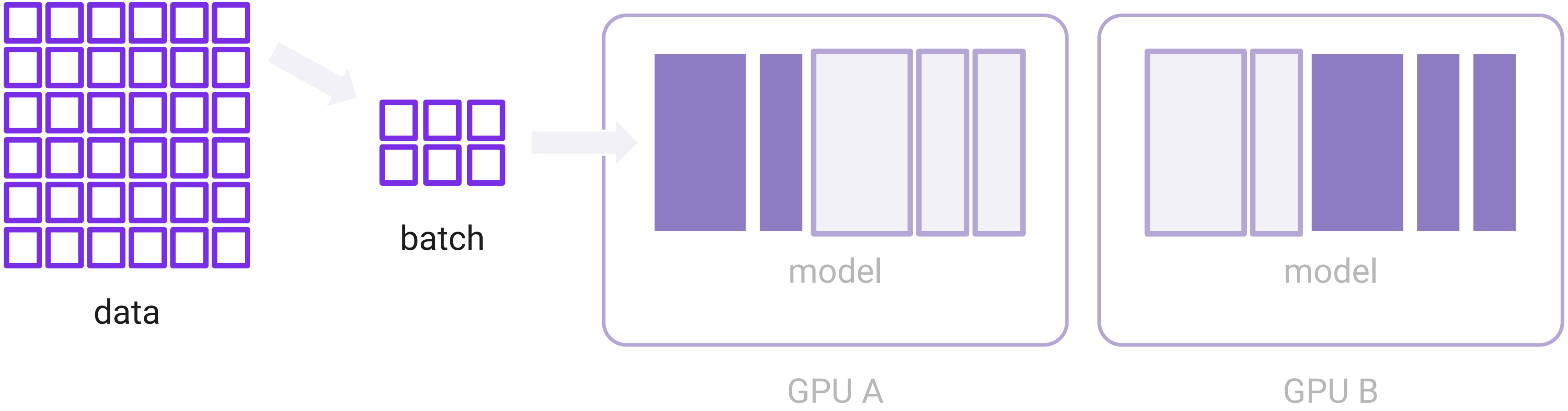


Distribute model parameters across machines in contiguous blocks

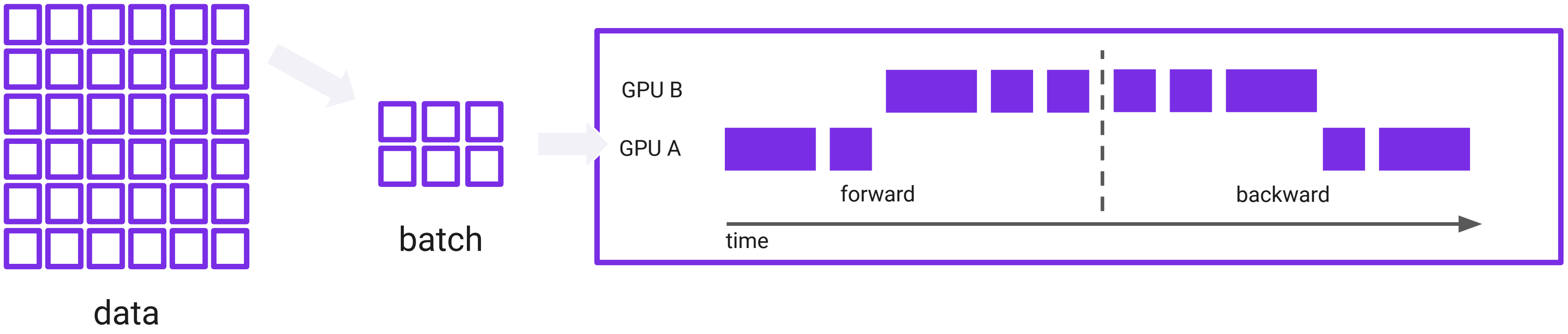
Lightning AI

Model parallelism

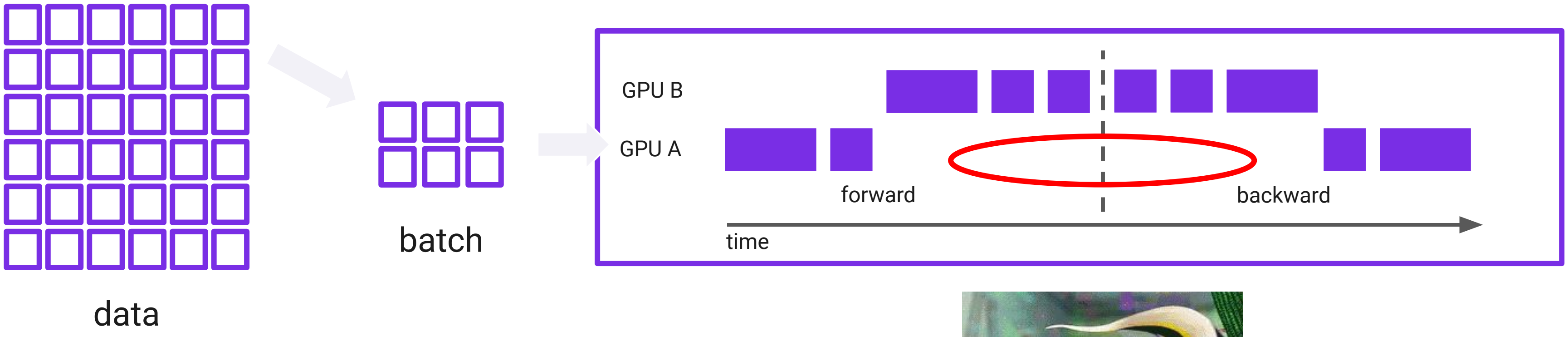


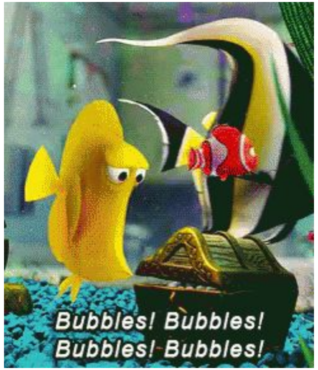


Model parallelism

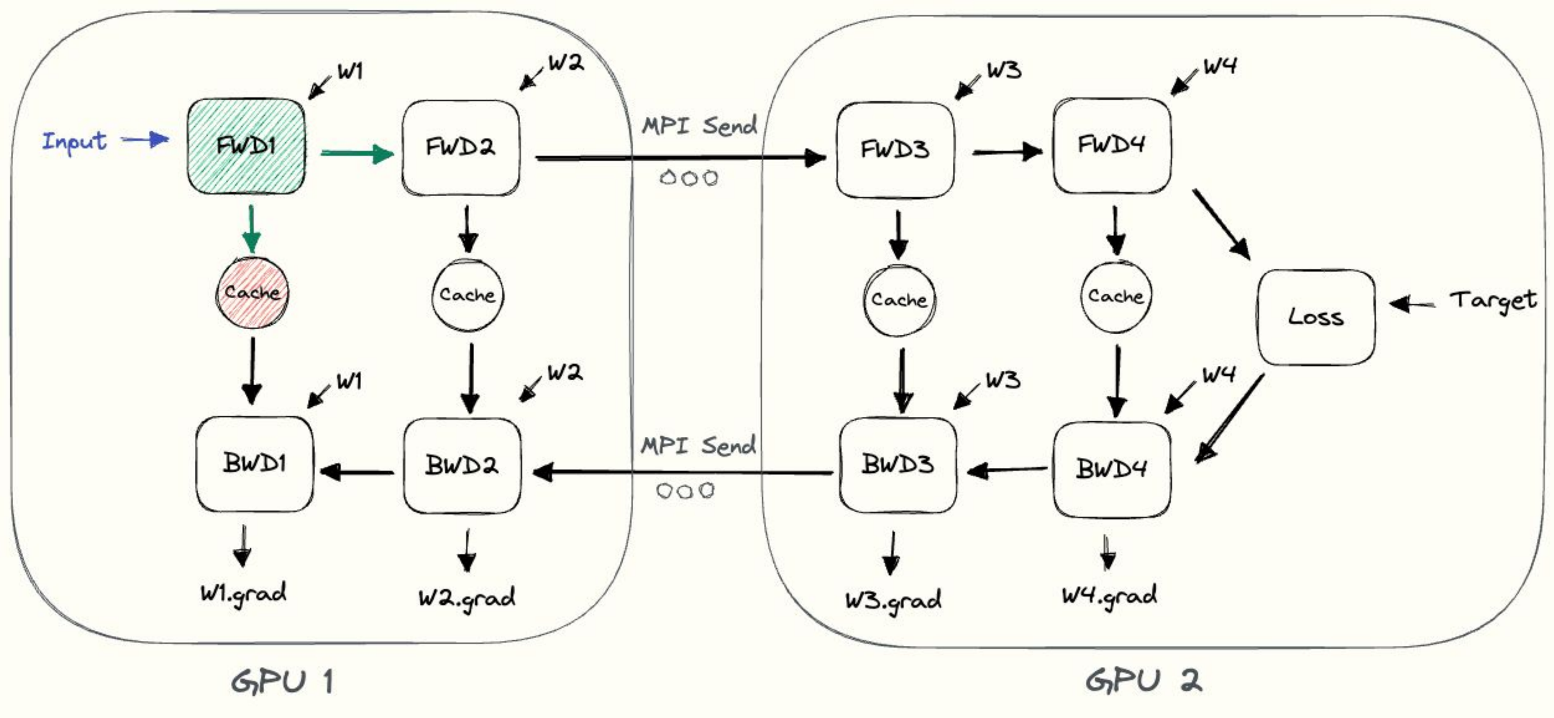


Model parallelism



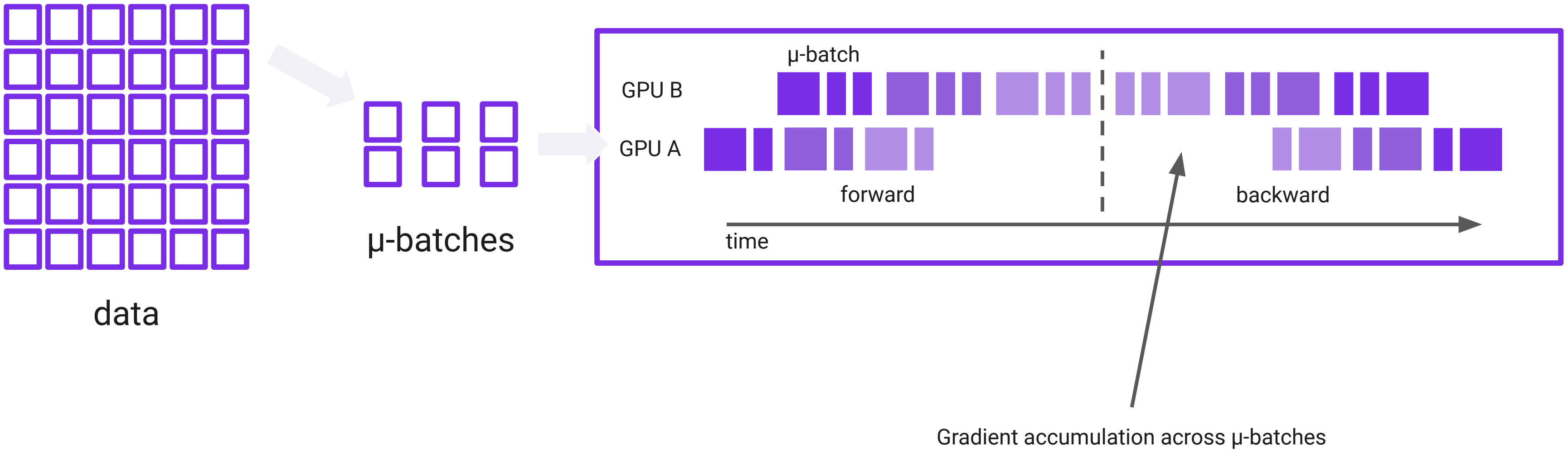


Naive model parallelism

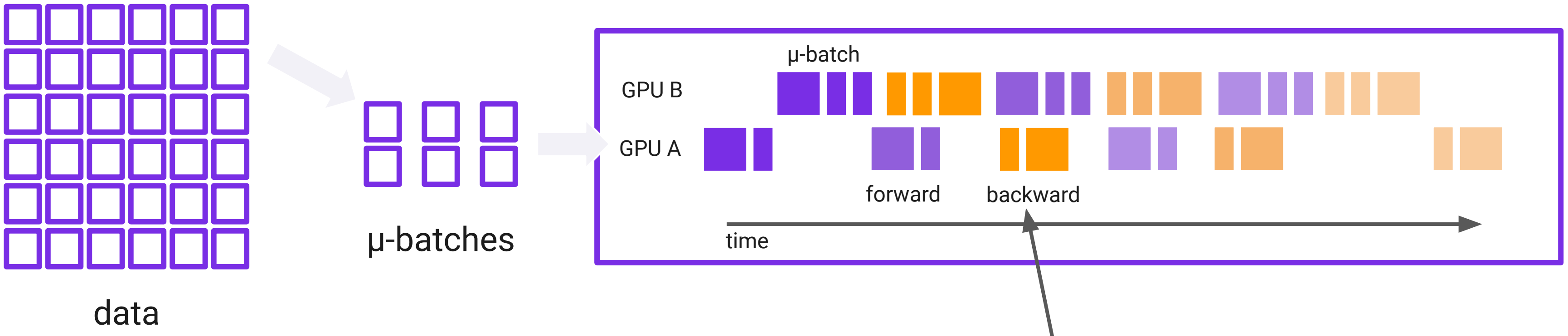


https://siboehm.com/articles/22/pipeline-parallel-training

Pipeline parallelism

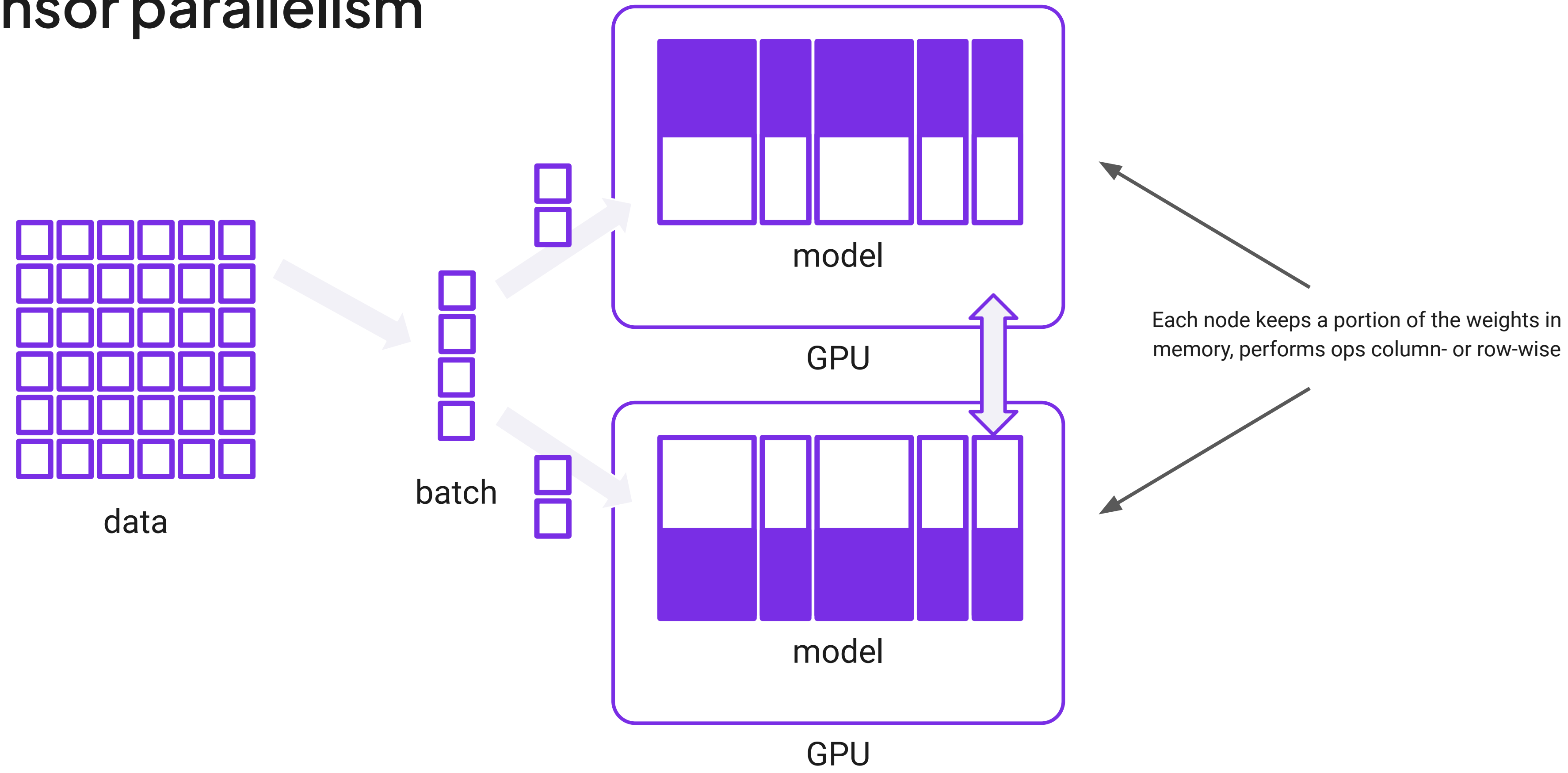


Interleaved pipeline parallelism



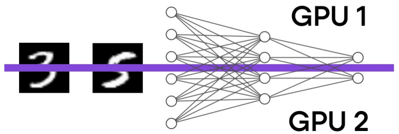
Compute backward as soon as possible
Bubble gets better as more nodes participate

Tensor parallelism

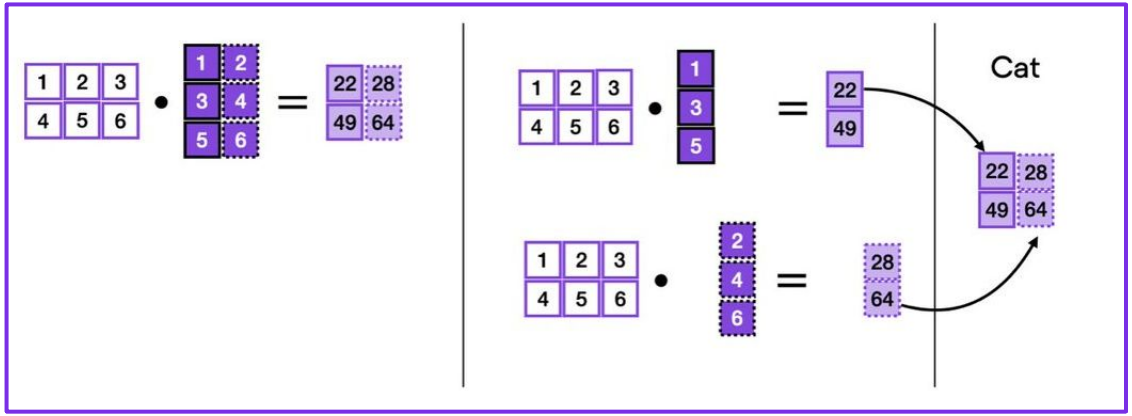


Tensor parallelism

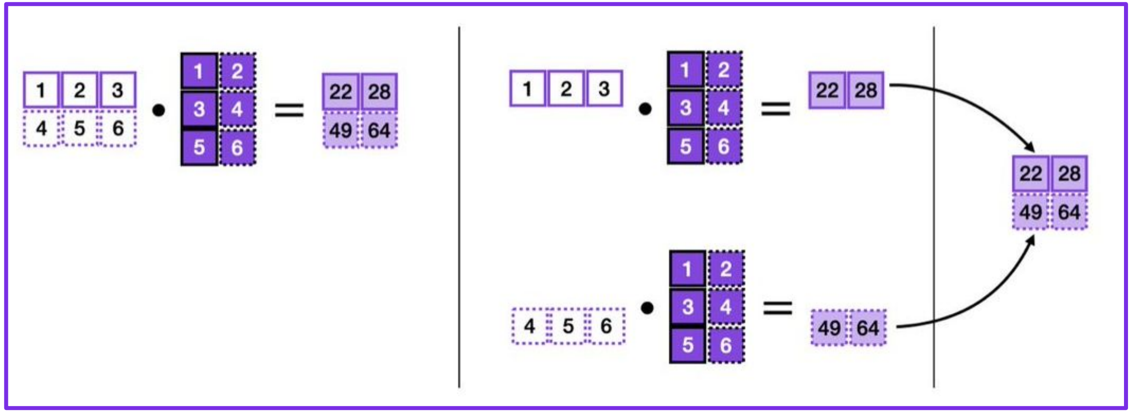
https://sebastianraschka.com/blog/2023/pytorch-faster.html



Column-wise



Row-wise

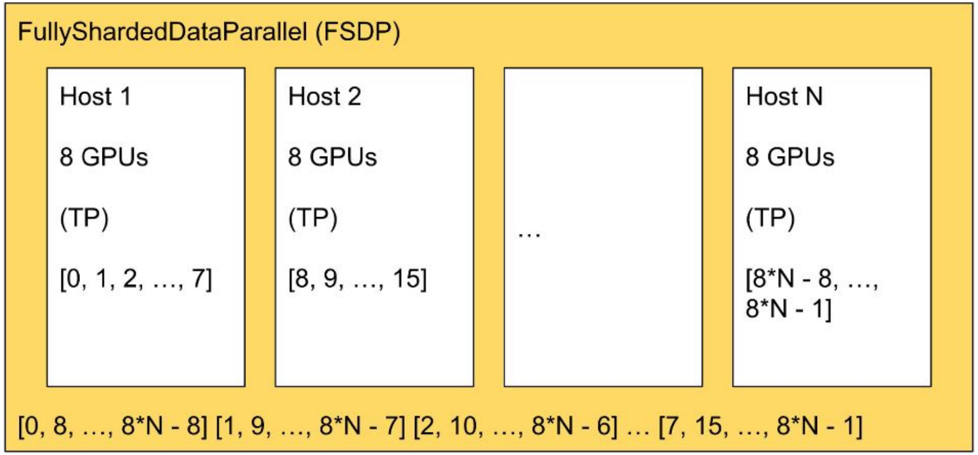


Combining parallelism: 2D, 3D, 4D

E.g. Sharded Data Parallel + Tensor Parallel in separate parallel dimensions:

- Data Parallel across hosts
- Tensor Parallel within each host

Other example: Megatron-LM



Wanchao Liang, Two Dimensional Parallelism Using Distributed Tensors

DDP

DDP - sharded DDP that shards **model parameters**, optimizer state and gradients across DDP ranks.
It can optionally offload to CPU.

```
trainer = Trainer(accelerator="gpu", strategy="ddp", devices=8, num_nodes=4)
```

DeepSpeed

- DeepSpeed ZeRO Stage 1 - Shard **optimizer states**, remains at speed parity with DDP whilst providing memory improvement
- DeepSpeed ZeRO Stage 2 - Shard **optimizer states** and **gradients**, remains at speed parity with DDP whilst providing even more memory improvement
- DeepSpeed ZeRO Stage 2 Offload - **Offload optimizer states** and **gradients** to CPU. Increases distributed communication volume and GPU-CPU device transfer, but provides significant memory improvement
- DeepSpeed ZeRO Stage 3 - **Shard optimizer states**, **gradients**, **parameters** and optionally **activations**. Increases distributed communication volume, but provides even more memory improvement
- DeepSpeed ZeRO Stage 3 Offload - **Offload optimizer states**, **gradients**, **parameters** and optionally **activations** to CPU. Increases distributed communication volume and GPU-CPU device transfer, but even more significant memory improvement.
- DeepSpeed Activation Checkpointing - **Free activations after forward pass**. Increases computation, but provides memory improvement for all stages.

DeepSpeed

```
trainer = Trainer(accelerator="gpu", devices=8, num_nodes=4, strategy="deepspeed_stage_1")
trainer = Trainer(accelerator="gpu", devices=8, num_nodes=4, strategy="deepspeed_stage_2")
trainer = Trainer(accelerator="gpu", devices=8, num_nodes=4, strategy="deepspeed_stage_2_offload")
trainer = Trainer(accelerator="gpu", devices=8, num_nodes=4, strategy="deepspeed_stage_3")
trainer = Trainer(accelerator="gpu", devices=8, num_nodes=4, strategy="deepspeed_stage_3_offload")
```

DeepSpeed

```python
import lightning as L
from lightning.pytorch.strategies import DeepSpeedStrategy
import deepspeed


class MyModel(L.LightningModule):
    ...

    def configure_sharded_model(self):
        self.block_1 = nn.Sequential(nn.Linear(32, 32),
nn.ReLU())
        self.block_2 = torch.nn.Linear(32, 2)

    def forward(self, x):
        # Use the DeepSpeed checkpointing function instead of
calling the module directly
        x = deepspeed.checkpointing.checkpoint(self.block_1, x)
        return self.block_2(x)
```

```python
model = MyModel()

trainer = L.Trainer(accelerator="gpu", devices=4,
strategy="deepspeed_stage_3_offload", precision=16)

# Enable CPU Activation Checkpointing
trainer = Trainer(
    accelerator="gpu",
    devices=4,
    strategy=DeepSpeedStrategy(
        stage=3,
        offload_optimizer=True,  # Enable CPU Offloading
        cpu_checkpointing=True,  # Offload activations to CPU
    ),
    precision=16,
)

trainer.fit(model)
```

FSDP (Fully Sharded Data Parallel)

FSDP - sharded DDP that **shards model parameters**, **optimizer state** and **gradients** across DDP ranks.

It optionally **offloads activations** and **optimizer state** to CPU.

```
trainer = Trainer(accelerator="gpu", strategy="fsdp", devices=8, num_nodes=4)
```

https://engineering.fb.com/2021/07/15/open-source/fsdp/

ColossalAI

ColossalAI - Zero-DP with dynamic chunk-based memory management and other configurable parallelization strategies.

```
class MyModel(LightningModule):
    def __init__(self):
        super().__init__()
        # don't instantiate layers here
        # move the creation of layers to
`configure_sharded_model`

    def configure_sharded_model(self):
        # create all your layers here
        self.layers = nn.Sequential(...)
```

```
from lightning_colossalai import ColossalAIStrategy

model = MyModel()
my_strategy = ColossalAIStrategy(placement_policy="auto")

trainer = Trainer(accelerator="gpu", devices=4, precision=16,
strategy=my_strategy)

trainer.fit(model)
```

Monitors the consumption of CUDA memory during the warmup phase and collects CUDA memory usage of all auto-grad operations

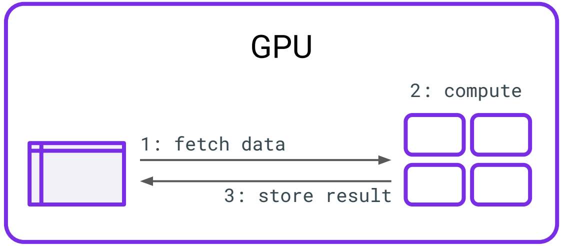
Automatically manages the data transmission between GPU and CPU according to collected CUDA memory usage information

Going fast

Eager mode

```
def forward(self, x):
    B, T, C = x.size()
    q, k ,v  = self.c_attn(x).split(self.n_embd, dim=2)
    k = k.view(B, T, self.n_head, C // self.n_head).transpose(1, 2)
    q = q.view(B, T, self.n_head, C // self.n_head).transpose(1, 2)
    v = v.view(B, T, self.n_head, C // self.n_head).transpose(1, 2)

    att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
    att = att.masked_fill(self.bias[:,:,:T,:T] == 0, float('-inf'))
    att = F.softmax(att, dim=-1)
    att = self.attn_dropout(att)

    y = att @ v
    y = y.transpose(1, 2).contiguous().view(B, T, C)
    y = self.resid_dropout(self.c_proj(y))
```

```
execute
execute
execute
execute
...
```

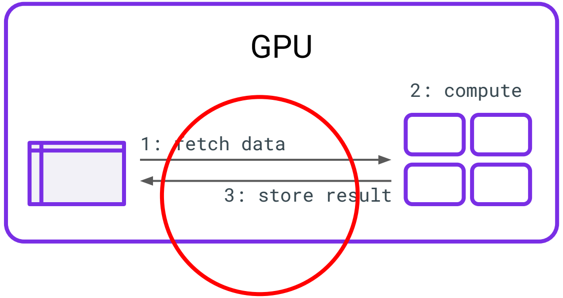


Going fast: bottleneck

Eager mode

```python
def forward(self, x):
    B, T, C = x.size()
    q, k ,v  = self.c_attn(x).split(self.n_embd, dim=2)
    k = k.view(B, T, self.n_head, C // self.n_head).transpose(1, 2)
    q = q.view(B, T, self.n_head, C // self.n_head).transpose(1, 2)
    v = v.view(B, T, self.n_head, C // self.n_head).transpose(1, 2)

    att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
    att = att.masked_fill(self.bias[:,:,:T,:T] == 0, float('-inf'))
    att = F.softmax(att, dim=-1)
    att = self.attn_dropout(att)

    y = att @ v
    y = y.transpose(1, 2).contiguous().view(B, T, C)
    y = self.resid_dropout(self.c_proj(y))
```

```
execute
execute
execute
execute
...
```

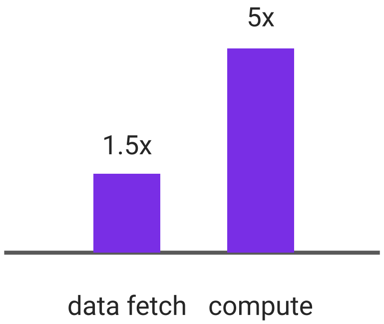


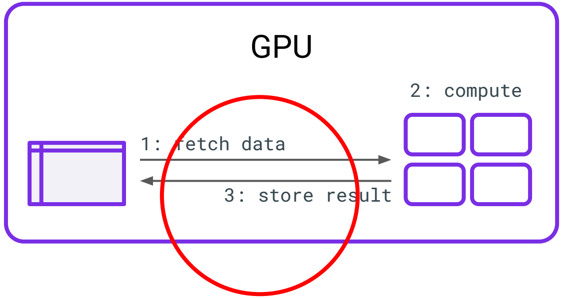
Bottleneck on modern hardware

Going fast: bottleneck

Eager mode

```
def forward(self, x):
    B, T, C = x.size()                                          ──────►  execute
    q, k ,v  = self.c_attn(x).split(self.n_embd, dim=2)         ──────►  execute
    k = k.view(B, T, self.n_head, C // self.n_head).transpose(1, 2)  ──►  execute
    q = q.view(B, T, self.n_head, C // self.n_head).transpose(1, 2)  ──►  execute
    v = v.view(B, T, self.n_head, C // self.n_head).transpose(1, 2)  ──►  ...

    att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
    att = att.masked_fill(self.bias[:,:,:T,:T] == 0, float('-inf'))
    att = F.softmax(att, dim=-1)
    att = self.attn_dropout(att)

    y = att @ v
    y = y.transpose(1, 2).contiguous().view(B, T, C)
    y = self.resid_dropout(self.c_proj(y))
```



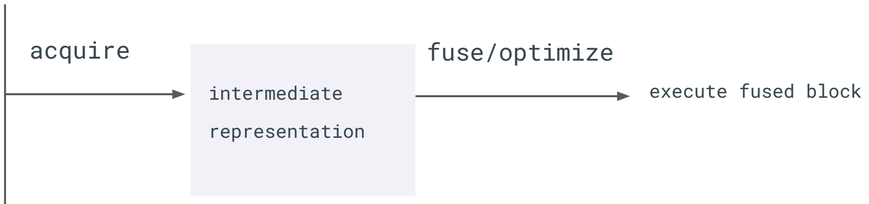


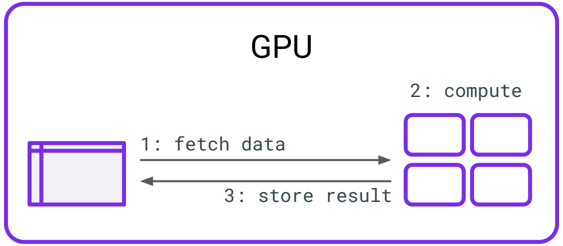
Bottleneck on modern hardware

Going fast: compiled mode

```python
def forward(self, x):
    B, T, C = x.size()
    q, k ,v  = self.c_attn(x).split(self.n_embd, dim=2)
    k = k.view(B, T, self.n_head, C // self.n_head).transpose(1, 2)
    q = q.view(B, T, self.n_head, C // self.n_head).transpose(1, 2)
    v = v.view(B, T, self.n_head, C // self.n_head).transpose(1, 2)

    att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
    att = att.masked_fill(self.bias[:,:,:T,:T] == 0, float('-inf'))
    att = F.softmax(att, dim=-1)
    att = self.attn_dropout(att)

    y = att @ v
    y = y.transpose(1, 2).contiguous().view(B, T, C)
    y = self.resid_dropout(self.c_proj(y))
```

Compiled mode



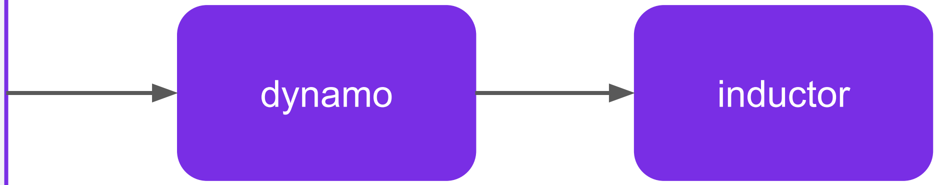


Infrequent memory access

Uninterrupted compute

Going fast: PyTorch 2.0

```python
model = NanoGPT(config)
model = torch.compile(model)
model(x)

# Works with PyTorch Lightning (+ Fabric)
model = MyLitModule()
model = torch.compile(model)
trainer.fit(model)
```

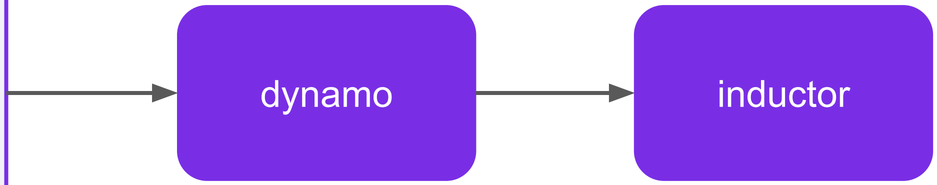


Program acquisition through tracing bytecode, optimization

Optimized execution, based on OpenAI Triton

Going fast: PyTorch 2.0

```
model = NanoGPT(config)
model = torch.compile(model)
model(x)

# Works with PyTorch Lightning (+ Fabric)
model = MyLitModule()
model = torch.compile(model)
trainer.fit(model)
```



Program acquisition through tracing bytecode, optimization

Optimized execution, based on OpenAI Triton

Status: **DDP supported, FSDP support in the works**

Demo: Model Parallel

github.com/Lightning-AI/open-bio-ml-workshop

Performance Optimization

Minimize framework overhead

Compile: Use *torch.compile* whenever possible

Logging: Log often for development, log less for long training runs

Checkpointing: Minimize frequency, checkpoint often only if training is unstable

Validation: Tune frequency based on dataset sizes

Maximize throughput

Compile: Use *torch.compile* whenever possible

Mixed precision: Speed + memory, prefer bfloat16 if available

Batch Size: Increase until OOM, avoid Malloc retries

Num Workers: Increase if GPU is waiting on data, consumes more CPU memory

Best practices

Avoid unnecessary GPU **synchronization**

```
# Avoid these:
output.item()
output.numpy()
output.cpu()

torch.cuda.empty_cache()
```

Create tensors directly on the **device**

```
# bad
t = torch.rand(4, 4).cuda()

# LightningModule:
torch.rand(4, 4, device=self.device)


# Fabric:
torch.rand(4, 4, device=fabric.device)
```

Performance flags

Speed (default)

```
# PyTorch / Fabric
torch.use_deterministic_algorithms(False)
torch.backends.cudnn.benchmark = True

# Trainer
trainer = Trainer(benchmark=True, deterministic=False)
```

Determinism

```
# PyTorch / Fabric
torch.use_deterministic_algorithms(True)
torch.backends.cudnn.benchmark = False

# Trainer
trainer = Trainer(benchmark=False, deterministic=True)
```

Comparing Trainer vs. PyTorch

Run "barebones"

```python
# Disable logging, checkpointing, etc.
trainer = Trainer(..., barebones=True)
```

Recommended for comparing implementations and **unit testing**!

Tensor cores

Perform **costly matrix multiplications** in lower precision (internal)

```
# Default
torch.set_float32_matmul_precision("high")

# Lower precision matrix multiplication
torch.set_float32_matmul_precision("highest")
torch.set_float32_matmul_precision("medium")
```

Lightning informs you if you have tensor cores

```
WARNING: You are using a CUDA device ('NVIDIA GeForce RTX 3090') that
has Tensor Cores. To properly utilize them, you should set
`torch.set_float32_matmul_precision('medium' | 'high')` which will
trade-off precision for performance.
```

Find bottlenecks (framework)

Configure profiler

```python
# Trainer
trainer = Trainer(profiler="simple", ...)
trainer.fit(...)
```

Output after fit

```
FIT Profiler Report

------------------------------------------------------------------------------------------------
|  Action                                          |  Mean duration (s)      |  Total time (s)  |
------------------------------------------------------------------------------------------------
|  [LightningModule]BoringModel.prepare_data       |  10.0001                |  20.00           |
|  run_training_epoch                              |  6.1558                 |  6.1558          |
|  run_training_batch                              |  0.0022506              |  0.015754        |
|  [LightningModule]BoringModel.optimizer_step     |  0.0017477              |  0.012234        |
|  [LightningModule]BoringModel.val_dataloader     |  0.00024388             |  0.00024388      |
|  on_train_batch_start                            |  0.00014637             |  0.0010246       |
|  [LightningModule]BoringModel.teardown           |  2.15e-06               |  2.15e-06        |
|  [LightningModule]BoringModel.on_train_start     |  1.644e-06              |  1.644e-06       |
|  [LightningModule]BoringModel.on_train_end       |  1.516e-06              |  1.516e-06       |
|  [LightningModule]BoringModel.on_fit_end         |  1.426e-06              |  1.426e-06       |
|  [LightningModule]BoringModel.setup              |  1.403e-06              |  1.403e-06       |
|  [LightningModule]BoringModel.on_fit_start       |  1.226e-06              |  1.226e-06       |
------------------------------------------------------------------------------------------------
```

Find bottlenecks (PyTorch)

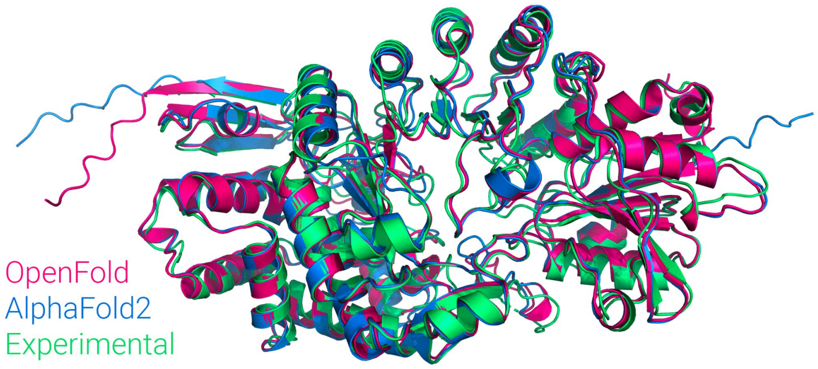
Configure profiler

```python
# Trainer
trainer = Trainer(profiler="pytorch", ...)
trainer.fit(...)
```

Output after fit

```
Profiler Report
Profile stats for: records
-------------------------------------------------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------  --
                                                   Name    Self CPU %      Self CPU   CPU total %     CPU total  CPU time avg     Self CUDA   Self CUDA %    CUDA total  CUD
-------------------------------------------------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------  --
                                          ProfilerStep*         3.48%      47.673ms        39.44%     540.549ms     180.183ms       0.000us         0.00%     113.315ms
[pl][profile][Strategy]SingleDeviceStrategy.backward...        32.14%     440.445ms        32.18%     441.004ms     147.001ms       0.000us         0.00%       3.000us
autograd::engine::evaluate_function: EmbeddingBackwa...         0.00%      57.000us        27.61%     378.372ms      63.062ms       0.000us         0.00%       4.470ms
                                     EmbeddingBackward0         0.00%      21.000us        27.60%     378.277ms      63.046ms       0.000us         0.00%       2.834ms
                               aten::embedding_backward         0.00%      15.000us        27.60%     378.256ms      63.043ms       0.000us         0.00%       2.834ms
                         aten::embedding_dense_backward         0.03%     360.000us        27.60%     378.241ms      63.040ms     659.000us         0.11%       2.834ms
                                  cudaStreamSynchronize        27.51%     377.063ms        27.51%     377.063ms      41.896ms       0.000us         0.00%       0.000us
                          [pl][profile]run_training_batch         0.05%     710.000us        26.02%     356.661ms     178.331ms       0.000us         0.00%     117.272ms
    [pl][profile][LightningModule]LitGPT.optimizer_step         0.00%      51.000us        25.97%     355.951ms     177.976ms       0.000us         0.00%     117.272ms
                                Optimizer.step#AdamW.step        23.54%     322.614ms        25.97%     355.900ms     177.950ms       0.000us         0.00%     117.272ms
[pl][profile][Strategy]SingleDeviceStrategy.training...         0.02%     224.000us         3.64%      49.858ms      16.619ms       0.000us         0.00%     175.918ms
                                   [pl][module]gpt.GPT: gpt         0.06%     773.000us         3.62%      49.634ms      16.545ms       0.000us         0.00%     175.918ms
                                  cudaDeviceSynchronize         3.26%      44.698ms         3.26%      44.698ms      44.698ms       0.000us         0.00%       0.000us
                                       cudaLaunchKernel         2.01%      27.556ms         2.01%      27.556ms       5.968us      14.669ms         2.55%      14.669ms
                                           aten::linear         0.11%       1.443ms         1.65%      22.615ms      76.922us       0.000us         0.00%     150.840ms
                                               aten::to         0.12%       1.584ms         1.33%      18.255ms      14.318us       0.000us         0.00%      56.977ms
                                         aten::_to_copy         0.27%       3.708ms         1.26%      17.297ms      17.850us       0.000us         0.00%      58.926ms
                                               aten::mm         0.58%       7.996ms         0.85%      11.625ms      26.361us     209.986ms        36.56%     211.492ms
        autograd::engine::evaluate_function: MmBackward0         0.10%       1.334ms         0.82%      11.205ms      76.224us       0.000us         0.00%     138.901ms
                                            MmBackward0         0.10%       1.370ms         0.72%       9.871ms      67.150us       0.000us         0.00%     138.901ms
-------------------------------------------------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------  --
Self CPU time total: 1.371s
Self CUDA time total: 574.402ms
```


Lightning AI



OpenFold Update: Lightning 2.0!

Reach out!

discord	discord.gg/MWAEvnC5fU
forums	lightning.ai/forums
twitter	@LightningAI

https://linktr.ee/lightningai

Thanks

Adrian Wälchli
Akihiro Nitta
Carlos Mocholí
Eden Afek
Ethan Harris
Jirka Borovec
Justus Schock
Thomas Chaton
William Falcon

\+

💜 community

https://linktr.ee/lightningai

Lightning LEAGUE

Thanks