Pytorch 框架中的并行计算

张亦弛

XJTU

December 8, 2022

Table of Contents

1. 并行化与分布式的深度学习
2. Pytorch 计算的并行化
3. 数据并行
4. 流水并行
5. 与贫穷共存

Parallelism, Concurrent and Distributed

并行计算：

- 多处理器并行（Multicore & SMT）
- GPU 处理器并行
- CUDA 与并行计算

Parallelism, Concurrent and Distributed

并行计算：

- 多处理器并行（Multicore & SMT）
- GPU 处理器并行
- CUDA 与并行计算

并发计算：

- 多任务的并行处理
- 计算与非计算负载混合
- 任务间顺序与依赖

Parallelism, Concurrent and Distributed

并行计算：

- 多处理器并行（Multicore & SMT）
- GPU 处理器并行
- CUDA 与并行计算

并发计算：

- 多任务的并行处理
- 计算与非计算负载混合
- 任务间顺序与依赖

分布式计算：

- 共享与独享主存空间
- 多节点并行计算（物理的与逻辑的）

Deep Learning with Parallelism and Distribution

数据与模型的高可并行性

Deep Learning with Parallelism and Distribution

数据与模型的高可并行性

更多的显存资源：

- 更大的 batch size
- 超大模型的训练

更多的计算资源：

- 加速模型训练
- 推理时更高的吞吐量

Deep Learning with Parallelism and Distribution

数据与模型的高可并行性

更多的显存资源：

- 更大的 batch size
- 超大模型的训练

更多的计算资源：

- 加速模型训练
- 推理时更高的吞吐量

多物理节点的分布式计算

- 单节点 PCIe 通道的限制：8 $\sim$ 10 GPUs/node
- 超大型训练实例：3072 $\times$ A100 80G、吞吐量 502 PFLOPS、1 兆参数的 GPT 模型训练 [1]

Table of Contents

1. 并行化与分布式的深度学习
2. Pytorch 计算的并行化
3. 数据并行
4. 流水并行
5. 与贫穷共存

CUDA Semantics in Pytorch

- nn.Module 与 torch.Tensor 的 device 属性——所处设备（cpu 或 cuda:x）
- 通常情况下，不同设备间无法相互运算（设备独享主存空间、无法互相访问）
- Extras：CUDA 异步执行（CUDA_LAUNCH_BLOCKING=1）

Table of Contents

1. 并行化与分布式的深度学习
2. Pytorch 计算的并行化
3. 数据并行
4. 流水并行
5. 与贫穷共存

torch.nn.DataParallel

- 基于多线程的实现
- 自动进行数据分割
- 每个线程分别执行对应数据的 forward 与 backward

```
model = nn.Sequential(...)

model = nn.DataParallel(model)
model = model.cuda()

f_loss = nn.MSELoss()
optim = optim.Adam(model.parameters())

for x in get_dataloader():
    optim.zero_grad()
    y = model(x.cuda())
    loss = f_loss(y, get_target_tensor())
    loss.backward()
    optim.step()
```

torch.nn.DataParallel

- 基于多线程的实现
- 自动进行数据分割
- 每个线程分别执行对应数据的 forward 与 backward

主要问题：

- 所有数据优先加载到 torch.device('cuda:0')，造成显存浪费
- Python 自身多线程实现的性能问题
- 无法与模型并行共同使用

```
model = nn.Sequential(...)

model = nn.DataParallel(model)
model = model.cuda()

f_loss = nn.MSELoss()
optim = optim.Adam(model.parameters())

for x in get_dataloader():
    optim.zero_grad()
    y = model(x.cuda())
    loss = f_loss(y, get_target_tensor())
    loss.backward()
    optim.step()
```

Python GIL

- 线程与进程

Python GIL

- 线程与进程
- CPython 解释器

Python GIL

- 线程与进程
- CPython 解释器
- CPython 中的全局解释器锁（Global Intepreter Lock）
- 源于单核环境、多线程与多核的冲突

Python GIL

- 线程与进程
- CPython 解释器
- CPython 中的全局解释器锁（Global Intepreter Lock）
- 源于单核环境、多线程与多核的冲突
- CPU 与 I/O
- 解释器内与解释器外

torch.nn.parallel.DistributedDataParallel

- 分布式架构的并行训练
- 单节点多 GPU 场景下的高性能（由于 GIL 的存在）
- 多个进程同时运行程序 python 代码

torch.nn.parallel.DistributedDataParallel

- 分布式架构的并行训练
- 单节点多 GPU 场景下的高性能（由于 GIL 的存在）
- 多个进程同时运行程序 python 代码

运行：

- `torchrun --nproc_per_node N training.py`
- `python -m torch.distributed.launch --nproc_per_node N training.py`

torch.nn.parallel.DistributedDataParallel

- 分布式架构的并行训练
- 单节点多 GPU 场景下的高性能（由于 GIL 的存在）
- 多个进程同时运行程序 python 代码

运行：

- `torchrun --nproc_per_node N training.py`
- `python -m torch.distributed.launch --nproc_per_node N training.py`

```python
# Example 1: simple forward/backward computation
torch.distributed.init_process_group(backend='nccl')

rank = torch.distributed.get_rank()
device = torch.device('cuda', rank)

model = nn.Sequential(...)
model = nn.parallel.DistributedDataParallel(model)
model = model.to(device)

f_loss = nn.MSELoss()
optim = optim.Adam(model.parameters())
optim.zero_grad()

x = get_input_tensor().to(device)
y = model(x)

loss = f_loss(y, get_target_tensor())
loss.backward() # Process Synchronization
optim.step()

if rank == 0:
    print("Completed calculation")
```

torch.nn.parallel.DistributedDataParallel

```python
# Example 2: DDP with data loading
torch.distributed.init_process_group(backend='nccl')
rank = torch.distributed.get_rank()
device = torch.device('cuda', rank)

dataset = MyDataset(...)
sampler = torch.utils.data.distributed.DistributedSampler(dataset)
loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
     sampler=sampler)

model = nn.Sequential(...)
model = nn.parallel.DistributedDataParallel(model)
model = model.to(device)
f_loss = nn.MSELoss()
optim = optim.Adam(model.parameters())

for epoch in range(epochs):
    loader.sampler.set_epoch(epoch) # Refresh random seed on epoch start

    for x in loader:
        optim.zero_grad()
        x = x.to(device)
        y = model(x)
        loss = f_loss(y, get_target_tensor())
        loss.backward()
        optim.step()
```

torch.nn.parallel.DistributedDataParallel

```python
# Example 2: DDP with data loading
torch.distributed.init_process_group(backend='nccl')
rank = torch.distributed.get_rank()
device = torch.device('cuda', rank)

dataset = MyDataset(...)
sampler = torch.utils.data.distributed.DistributedSampler(dataset)
loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
      sampler=sampler)

model = nn.Sequential(...)
model = nn.parallel.DistributedDataParallel(model)
model = model.to(device)
f_loss = nn.MSELoss()
optim = optim.Adam(model.parameters())

for epoch in range(epochs):
    loader.sampler.set_epoch(epoch) # Refresh random seed on epoch start

    for x in loader:
        optim.zero_grad()
        x = x.to(device)
        y = model(x)
        loss = f_loss(y, get_target_tensor())
        loss.backward()
        optim.step()
```

存储模型参数：

```python
if rank == 0:
    torch.save(model.state_dict(),
      CHECKPOINT_PATH)

dist.barrier() # Force sync
ddp_model.load_state_dict(torch.load(
      CHECKPOINT_PATH))
```

输出 loss 值：

```python
def print_averaged_value(value):
    torch.distributed.all_reduce(value)
    value /= torch.distributed.
      get_world_size()
    if rank == 0:
        print(value)
```

Simple IPC Methods in Distributed Pytorch

Key-Value 存储：TCPStore、FileStore、HashStore（单进程）、类似 Dict 的数据结构
点对点通信：send(tensor, dst_rank)、recv(tensor, src_rank)、点对点发送数据
Collective Functions:

- 伪代码说明：函数名 (输入输出参数名: 参数类型)，in 代表函数输入、out 代表函数输出（实际使用需要初始化大小合适的 Tensor 以供函数返回数据）、inout 代表输入与输出视情况而定

Simple IPC Methods in Distributed Pytorch

Key-Value 存储：TCPStore、FileStore、HashStore（单进程）、类似 Dict 的数据结构
点对点通信：send(tensor, dst_rank)、recv(tensor, src_rank)、点对点发送数据
Collective Functions:

- 伪代码说明：函数名 (输入输出参数名: 参数类型)，in 代表函数输入、out 代表函数输出（实际使用需要初始化大小合适的 Tensor 以供函数返回数据）、inout 代表输入与输出视情况而定
- broadcast(inout tensor: Tensor, in sr_rank: int)，从 src_rank 广播到所有进程（input if is src_rank）
- all_reduce(in tensor: Tensor, in op: ReduceOp)，对所有进程上的 tensor 执行 op 操作（SUM、PRODUCT、MIN、MAX）
- all_gather(out tensor_list: list, in tensor: Tensor)，收集所有进程上的 tensor
- scatter(out tensor: Tensor, inout scatter_list: list, in src_rank: int)，将 scatter_list 中的 tensor 分布到各个进程（input if is src_rank）
- barrier()，同步所有进程到该位置
- broadcast、reduce、gather、scatter 都有相应的 _object 版本，但在 nccl 后端下有着更高的要求（通常要求是对象可以 pickle），因此不推荐使用。

Table of Contents

1. 并行化与分布式的深度学习
2. Pytorch 计算的并行化
3. 数据并行
4. 流水并行
5. 与贫穷共存

Model Sharding

模型分片：解决过大模型无法在单一 GPU 上训练的问题。

梯度更新：

对双层神经网络，设 L 为损失函数、
f_2 为第二层函数，参数为 $\mathbf{w}_2$、
f_1 为第一层函数，参数为 $\mathbf{w}_1$。
则有

$$\frac{\partial L}{\partial \mathbf{w}_1} = \frac{\partial L}{\partial f_2} \cdot \frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_1}{\partial \mathbf{w}_1}$$

$$\frac{\partial L}{\partial \mathbf{w}_2} = \frac{\partial L}{\partial f_2} \cdot \frac{\partial f_2}{\partial \mathbf{w}_2}$$

Model Sharding

模型分片：解决过大模型无法在单一 GPU 上训练的问题。

梯度更新：

对双层神经网络，设 L 为损失函数、f_2 为第二层函数，参数为 $\mathbf{w}_2$、f_1 为第一层函数，参数为 $\mathbf{w}_1$。则有

$$\frac{\partial L}{\partial \mathbf{w}_1} = \frac{\partial L}{\partial f_2} \cdot \frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_1}{\partial \mathbf{w}_1}$$

$$\frac{\partial L}{\partial \mathbf{w}_2} = \frac{\partial L}{\partial f_2} \cdot \frac{\partial f_2}{\partial \mathbf{w}_2}$$

Pytorch autograd 支持下的简单实现

```
gpu0 = torch.cuda.device(0)
gpu1 = torch.cuda.device(1)
model1 = nn.Sequential(...).to(gpu0) # Very large model part1
model2 = nn.Sequential(...).to(gpu1) # Very large model part2
f_loss = nn.MSELoss()
optim = optim.Adam(list(model1.parameters()) + list(model2.
      parameters())
optim.zero_grad()
x = get_input_tensor()

y = model1(x)
y = model2(y)

loss = f_loss(y, get_target_tensor())
loss.backward()
optim.step()
```

Pipeline Model Parallelism

- CPU 指令流水
- 深度学习流水并行

Pipeline Model Parallelism

- CPU 指令流水
- 深度学习流水并行

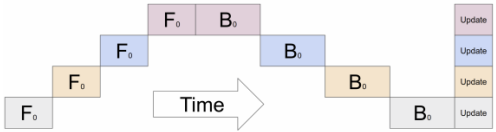

Figure: 无流水的训练流程

Pipeline Model Parallelism

- CPU 指令流水
- 深度学习流水并行
- $\frac{\partial L}{\partial \mathbf{w}_1} = \frac{\partial L}{\partial f_2} \cdot \frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_1}{\partial \mathbf{w}_1}$
- $\frac{\partial L}{\partial \mathbf{w}_1}$ 梯度的计算需要 $\frac{\partial L}{\partial f_2}$

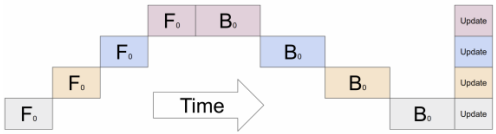

Figure: 无流水的训练流程

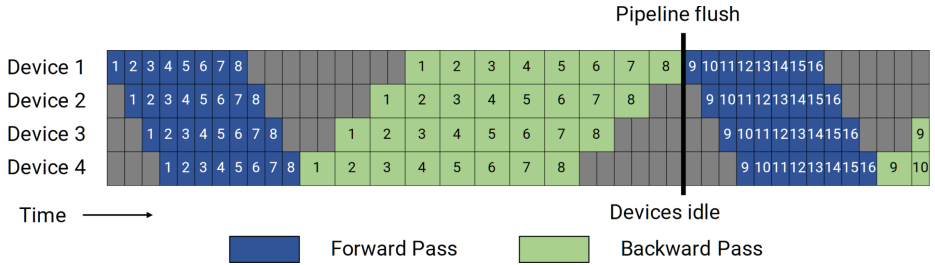

Figure: GPipe 流水并行示意

Pipeline in Pytorch (Experimental)

- 采用 GPipe 算法
- 实验性特征
- 限制 1：在与 DistributedDataParallel 共同使用时无法自动保存 checkpoint
- 限制 2：无法支持跨节点的流水并行训练

```
# Initialize RPC
torch.distributed.rpc.init_rpc('worker', rank=0, world_size=1)

gpu0 = torch.cuda.device(0)
gpu1 = torch.cuda.device(1)
model1 = nn.Sequential(...).to(gpu0) # Very large model part1
model2 = nn.Sequential(...).to(gpu1) # Very large model part2

model = nn.Sequential(model1, model2)
model = Pipe(model, chunks=8)

f_loss = nn.MSELoss()
optim = optim.Adam(model.parameters())
optim.zero_grad()
x = get_input_tensor()
y = model(x).local_value()
loss = f_loss(y, get_target_tensor())
loss.backward()
optim.step()
```

Table of Contents

1. 并行化与分布式的深度学习
2. Pytorch 计算的并行化
3. 数据并行
4. 流水并行
5. 与贫穷共存

Live with Poverty

- 低精度训练——FP16、BF16（AMP）、TF32（automatic, not default since 1.12）
- 低显存训练——在不同设备间交换参数

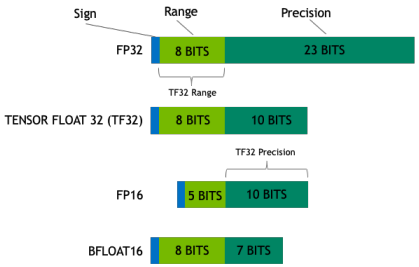

Figure: 浮点数格式示意

```
model1 = nn.Sequential(...) # Very large model part1
model2 = nn.Sequential(...) # Very large model part2
f_loss = nn.MSELoss()
optim1 = optim.Adam(model1.parameters())
optim1.zero_grad()
optim2 = optim.Adam(model2.parameters())
optim2.zero_grad()
x = get_input_tensor()
model1 = model1.cuda()

y = model1(x)
model1 = model1.cpu()
model2 = model2.cuda()
y = model2(y)

loss = f_loss(y, get_target_tensor())
loss.backward()
optim2.step()
model2 = model2.cpu()
model1 = model1.cuda()
optim1.step()
```

References, Q&A

[1] Deepak Narayanan et al. “Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM”. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC ’21. New York, NY, USA: Association for Computing Machinery, Nov. 13, 2021, pp. 1–15. ISBN: 978-1-4503-8442-1. DOI: 10.1145/3458817.3476209. URL: https://doi.org/10.1145/3458817.3476209 (visited on 12/07/2022).