Pytorch多GPU训练

合肥工業大學

柯水洲

2022.6.23

提　纲

01 简介

02 代码操作

简介

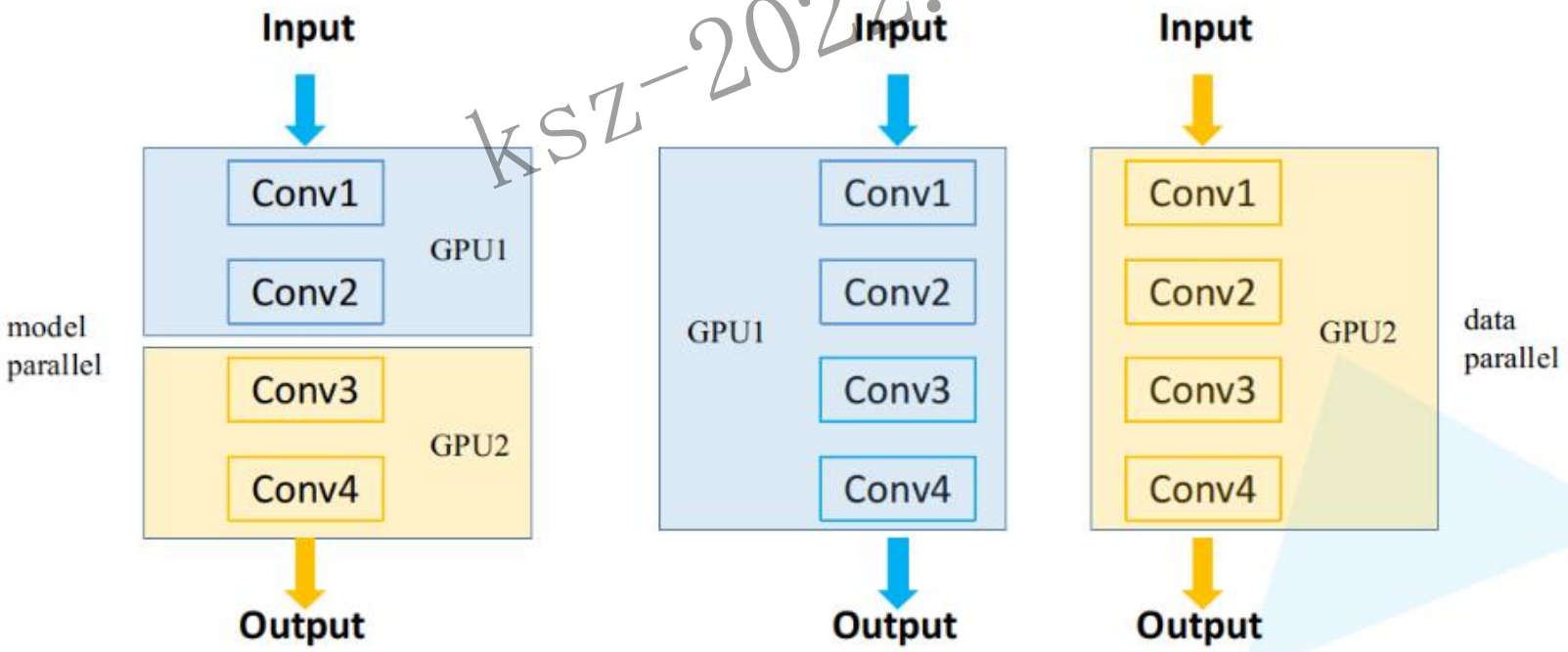
在训练模型中，为了加速训练过程，会使用多块GPU设备进行并行训练，比如单机单卡，单机多卡，多机多卡等方式:

（1）模型并行：一个Device负责处理模型的一个切片 (例如模型的一层)；

（2）数据并行：一个Device负责处理数据的一个切片 (即Batch的一部分)

同步更新：每个batch所有GPU计算完成后，再统一计算新权值

异步更新：每个GPU计算完梯度后，立即更新整体权值并同步

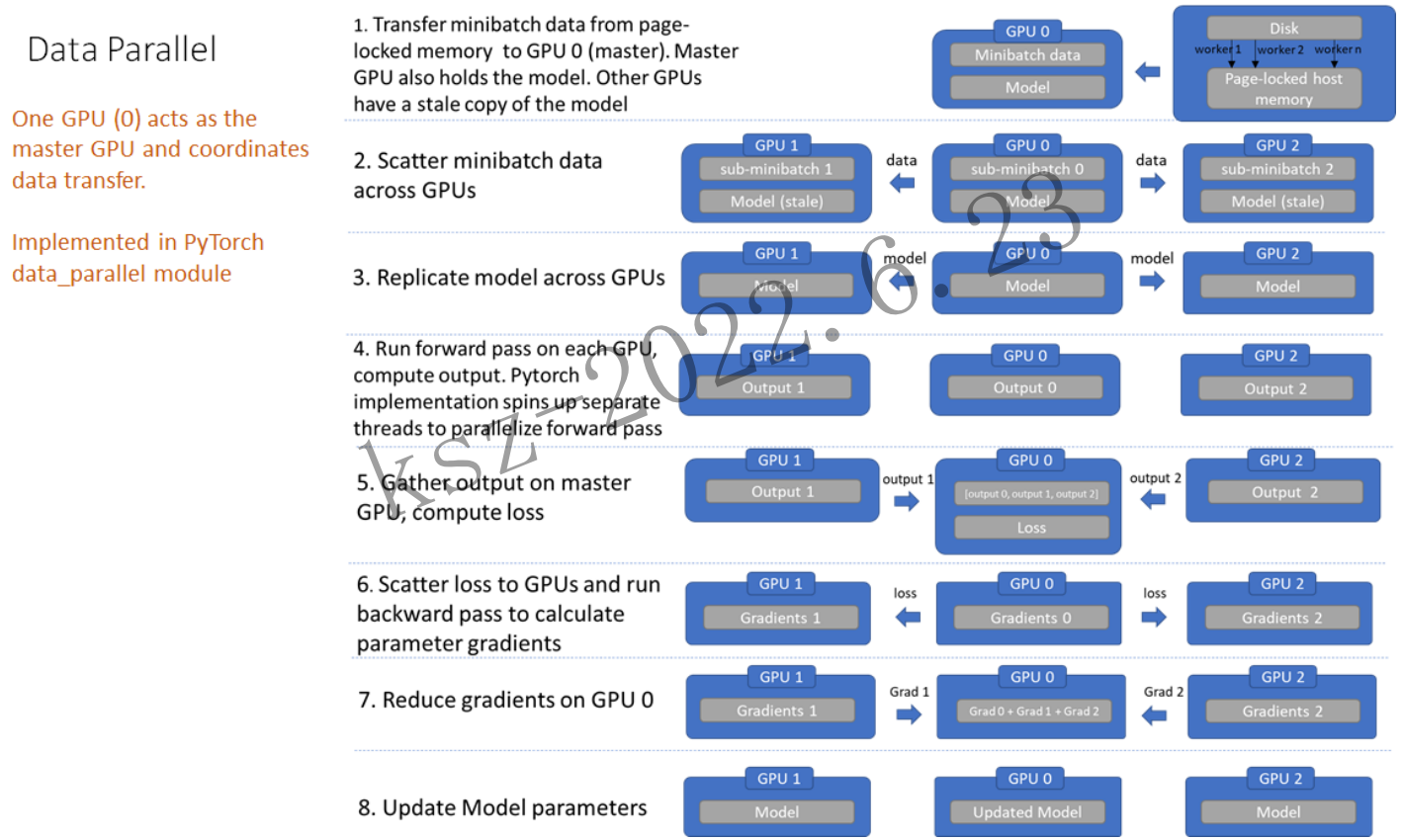


简介

（1）DataParallel是梯度汇总到gpu0，反向传播更新参数，再广播参数给其他的GPU

→ →传输参数量过大，效率低，GPU0负载过大，导致不均

（2） DataParallel基于单进程多线程→ →并行性能受到GIL争用开销的阻碍。

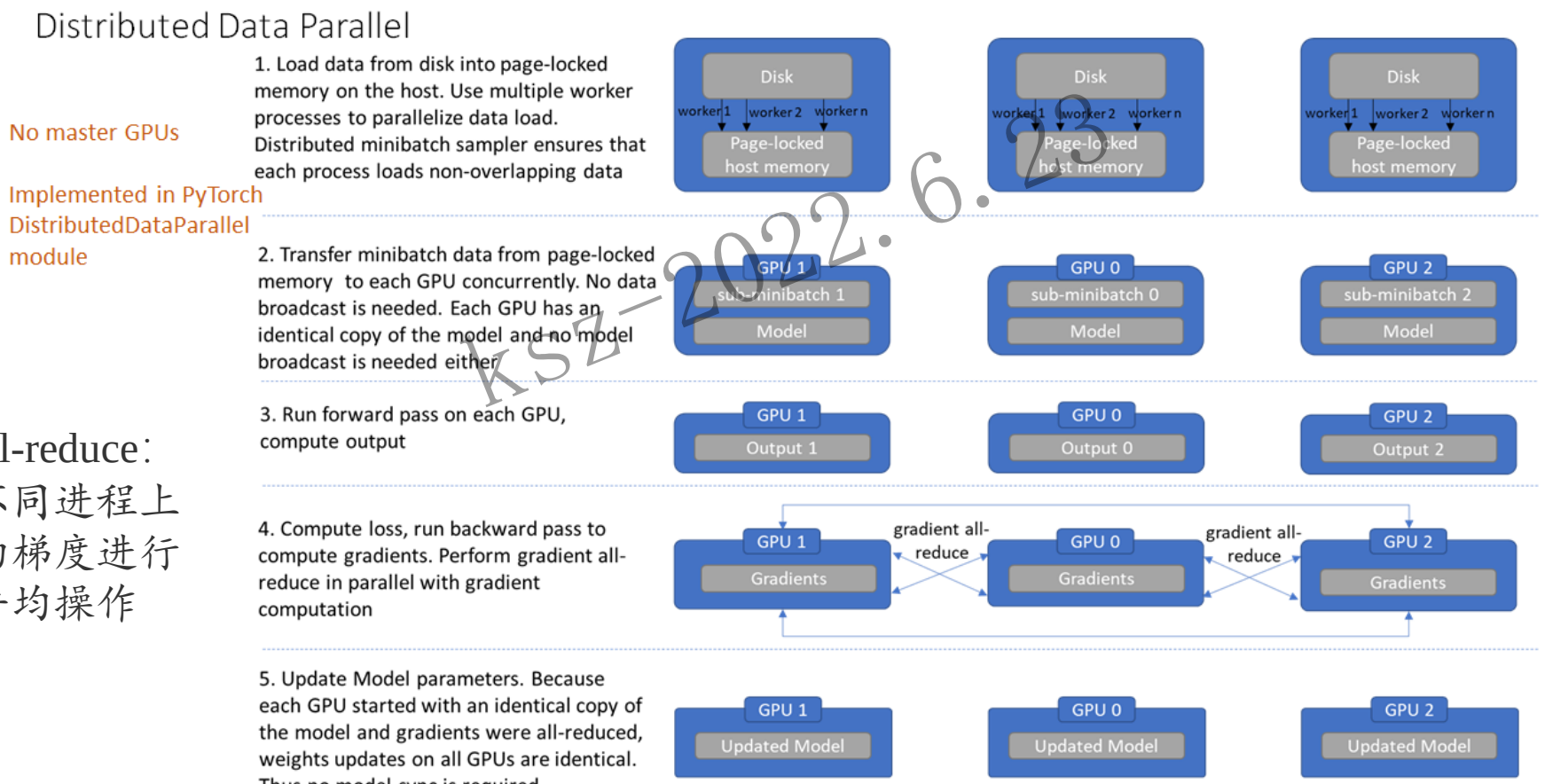


Global Interpreter Lock（GIT）（车厢）线程<进程（火车）

GIL规定，在一个进程中每次只能有一个线程在运行。这个GIL锁相当于是线程运行的资格证，某个线程想要运行，首先要获得GIL锁，然后遇到IO或者超时的时候释放GIL锁，给其余的线程去竞争，竞争成功的线程获得GIL锁得到下一次运行的机会。

简介

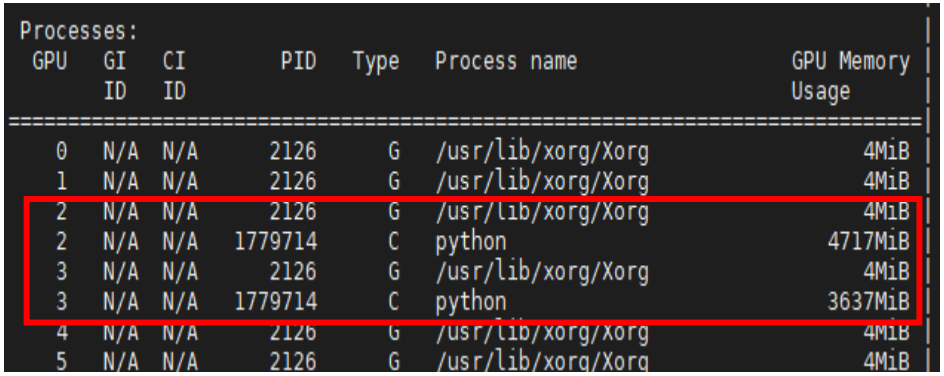
（1）DistributedDataParallel各进程梯度计算完成之后，各进程需要将梯度进行汇总平均，然后再由 rank=0 的进程，将其 broadcast 到所有进程。之后，各进程用该梯度来独立的更新参数。
→ → 传输参数量想对较少，效率提高，各GPU负载均等。
（2） DistributedDataParallel基于多进程，每个进程匹配一个CPython解释器和GIL → → 性能提升



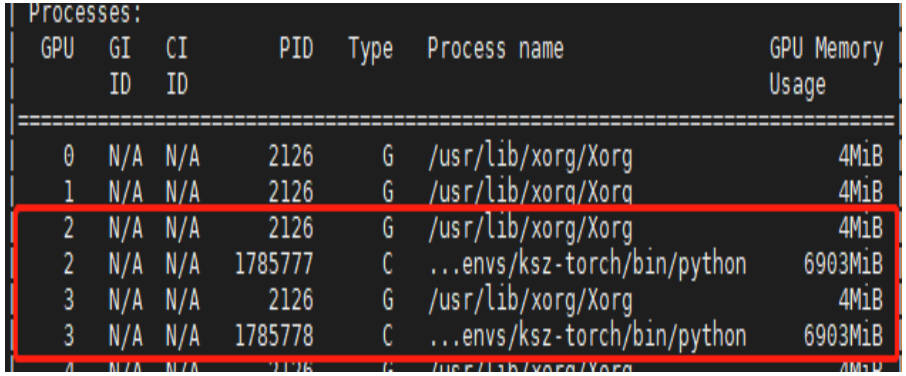
all-reduce：不同进程上的梯度进行平均操作

简介

DataParallel	DistributedDataParallel
数据并行	数据并行、模型并行
方式单机多卡	单机多卡、多机多卡
单进程、多线程	多进程方式
内存和GPU使用率负载不均，数据传输量大效率不高	完美解决负载不均衡的问题，数据传输量更少,因此速度更快,效率更高。
实现简单	实现略微复杂
batch_size设置必须为单卡的n倍	batch_size设置于单卡一样即可



```
Processes:
 GPU   GI   CI        PID   Type   Process name                  GPU Memory
       ID   ID                                                   Usage
=============================================================================
    0   N/A  N/A      2126      G   /usr/lib/xorg/Xorg                  4MiB
    1   N/A  N/A      2126      G   /usr/lib/xorg/Xorg                  4MiB
    2   N/A  N/A      2126      G   /usr/lib/xorg/Xorg                  4MiB
    2   N/A  N/A   1779714      C   python                           4717MiB
    3   N/A  N/A      2126      G   /usr/lib/xorg/Xorg                  4MiB
    3   N/A  N/A   1779714      C   python                           3637MiB
    4   N/A  N/A      2126      G   /usr/lib/xorg/Xorg                  4MiB
    5   N/A  N/A      2126      G   /usr/lib/xorg/Xorg                  4MiB
```



```
Processes:
 GPU   GI   CI        PID   Type   Process name                  GPU Memory
       ID   ID                                                   Usage
=============================================================================
    0   N/A  N/A      2126      G   /usr/lib/xorg/Xorg                  4MiB
    1   N/A  N/A      2126      G   /usr/lib/xorg/Xorg                  4MiB
    2   N/A  N/A      2126      G   /usr/lib/xorg/Xorg                  4MiB
    2   N/A  N/A   1785777      C   ...envs/ksz-torch/bin/python        6903MiB
    3   N/A  N/A      2126      G   /usr/lib/xorg/Xorg                  4MiB
    3   N/A  N/A   1785778      C   ...envs/ksz-torch/bin/python        6903MiB
```

代码操作

代码编写
- （1）Model=DataParallel(model.cuda(),deveice_ids=[0,1,2])
- （2）data = data.cuda()

注意事项
- （1）torch.save：注意模型需要调用model.modules.state_dict()
- （2）torch.load：需要注意map_location的使用

代码操作

代码编写

（1） init_process_group(backend,init_method=None,timeout=default_pg_timeout,world_size =-1, rank =- 1, store = None, group_name = '' , pg_options = None)……初始化进程组

（2）Torch.cuda.set_device(args.local_rank)

……设置可用GPU

（3）model=DistributedDataParallel(model.cuda(args.local_rank),device_ids=[args.local_rank])

……创建分布式并行模型

（4）train_sampler = DistributedSampler(train_dataset)

……为数据集创建 Sampler

（5）train_dataloader = Dataloader(…,sampler = train_sampler)

（6）data=data.cuda(args.local_rank)

（7）Python –m torch.distributed.launch --nproc_per_node=n_gpus train.py

注意事项

（1）torch.save：注意模型需要调用model.modules.state_dict()

（2）torch.load：需要注意map_location的使用

（3）在每个周期开始处，调用train_sampler.set_epoch(epoch)，用于打乱数据

（4）有了sampler，不需要在dataloader设置Shuffle=True

代码操作

（1）init_process_group(backend,init_method,world_size,rank=args.local_rank)

……初始化进程组

（1）backend：通信后端，如果使用的是Nvidia的GPU建议使用NCCL，CPU训练时使用Gloo

（2）init_method：直接使用默认的env://

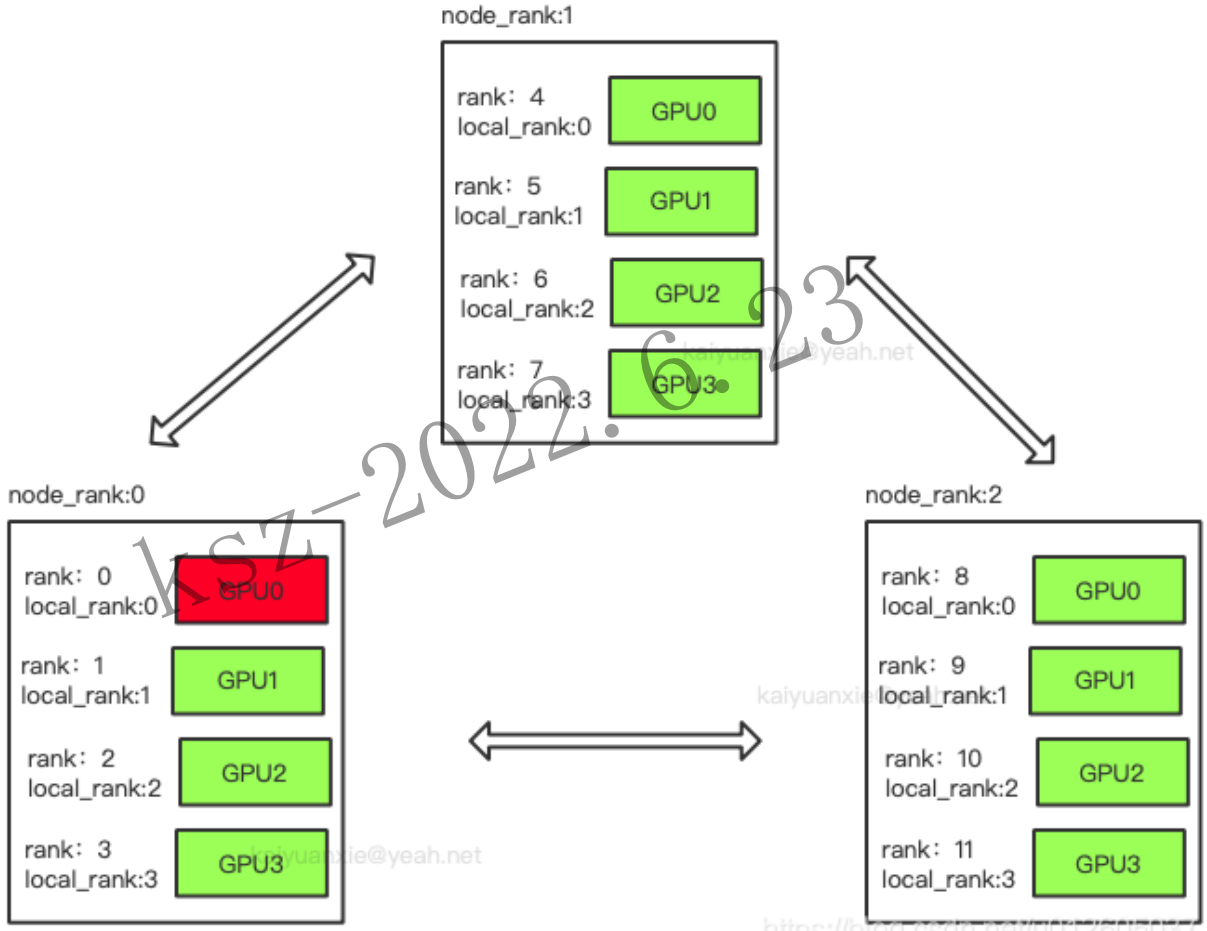
（3）world size：表示全局进程个数（几个GPU）

（4）rank：表示进程序号，用于进程间通讯，表征进程优先级。rank = 0 的主机为 master 节点

（5）local_rank：local_rank代表着一个进程在一个机器中的序号，但local_rank这个参数比较特殊，负责提供其参数的不是我们，而是一个叫torch.distributed.launch的启动工具，这个后面会再提到。

代码操作

为了方便理解举个例子，比如分布式中有三台机器，每台机器起4个进程，每个进程占用1个GPU，如下图所示：



图中：一共有12个rank，nproc_per_node=4，nnodes=3，每个节点都一个对应的node_rank。

代码操作

（2）Torch.cuda.set_device(args.local_rank)

……设置可用GPU

（3）model=DistributedDataParallel(model.cuda(args.local_rank),device_ids=[args.local_rank])

……创建分布式并行模型

我们将args.local_rank的取值作为torch.cuda.set_device()的参数。那么后面每当我们调用XX.cuda()或是XX.to('cuda')时，就都将XX放到了序号为local_rank的GPU上。因此，虽然每个进程运行同一份代码，但由于每个进程local_rank不同，每个进程的操作也不同了。

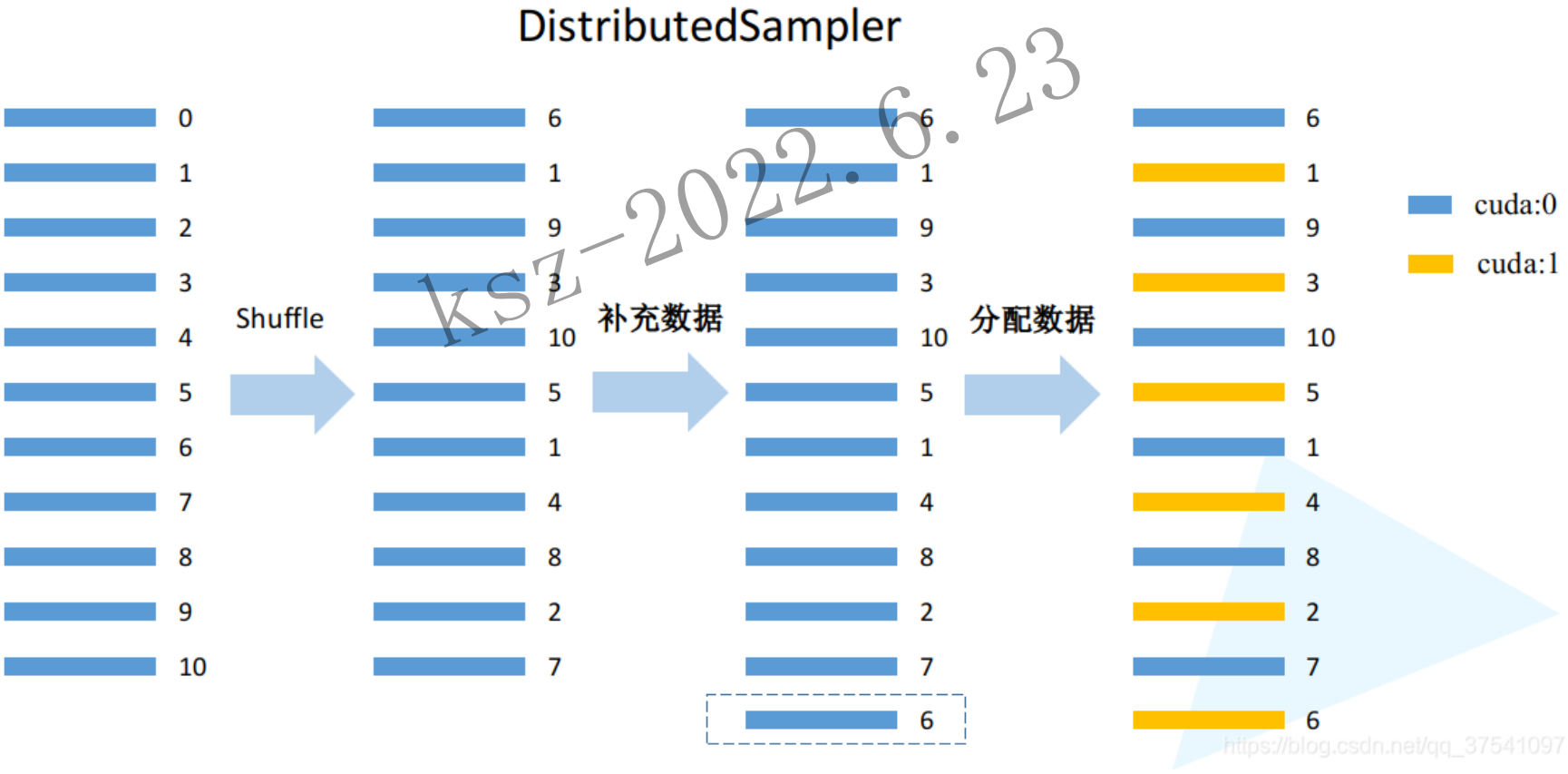
代码操作

（4）train_sampler = DistributedSampler(train_dataset)

……为数据集创建 Sampler

（5）train_dataloader = Dataloader(…,sampler = train_sampler)

（6）data=data.cuda(args.local_rank)



代码操作

（7）Python –m torch.distributed.launch --nproc_per_node=n_gpus train.py

……运行

torch.distributed.launch

代码量少点，启动速度快点

python -m torch.distributed.launch --help

-m: run library module as a script

torch.multiprocessing

拥有更好的控制和灵活性

注意事项

（1）torch.save：注意模型需要调用model.modules.state_dict()

（2）torch.load：需要注意map_location的使用

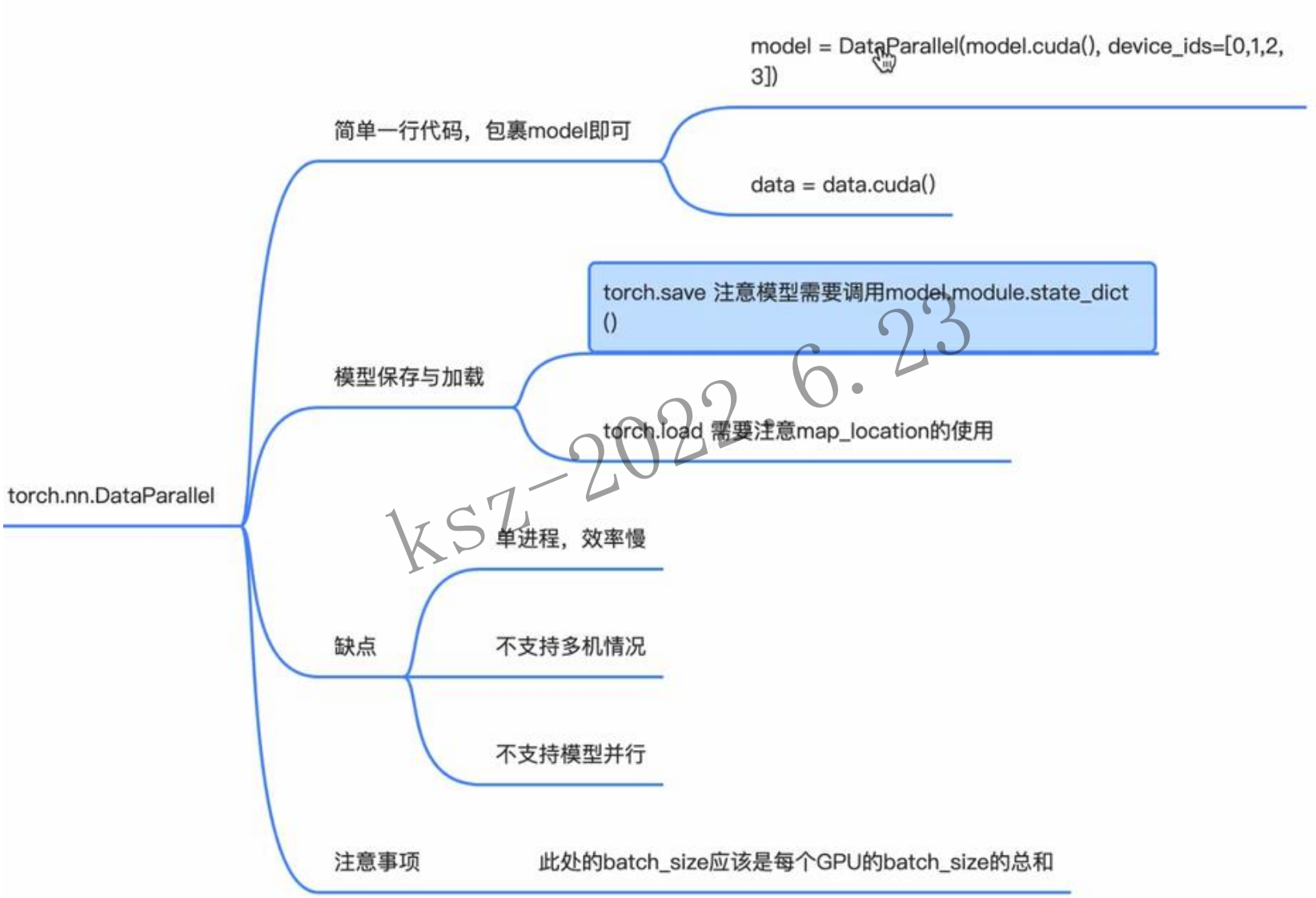
（3）在每个周期开始处，调用train_sampler.set_epoch(epoch)，用于打乱数据

（4）有了sampler，不需要在dataloader设置Shuffle=True

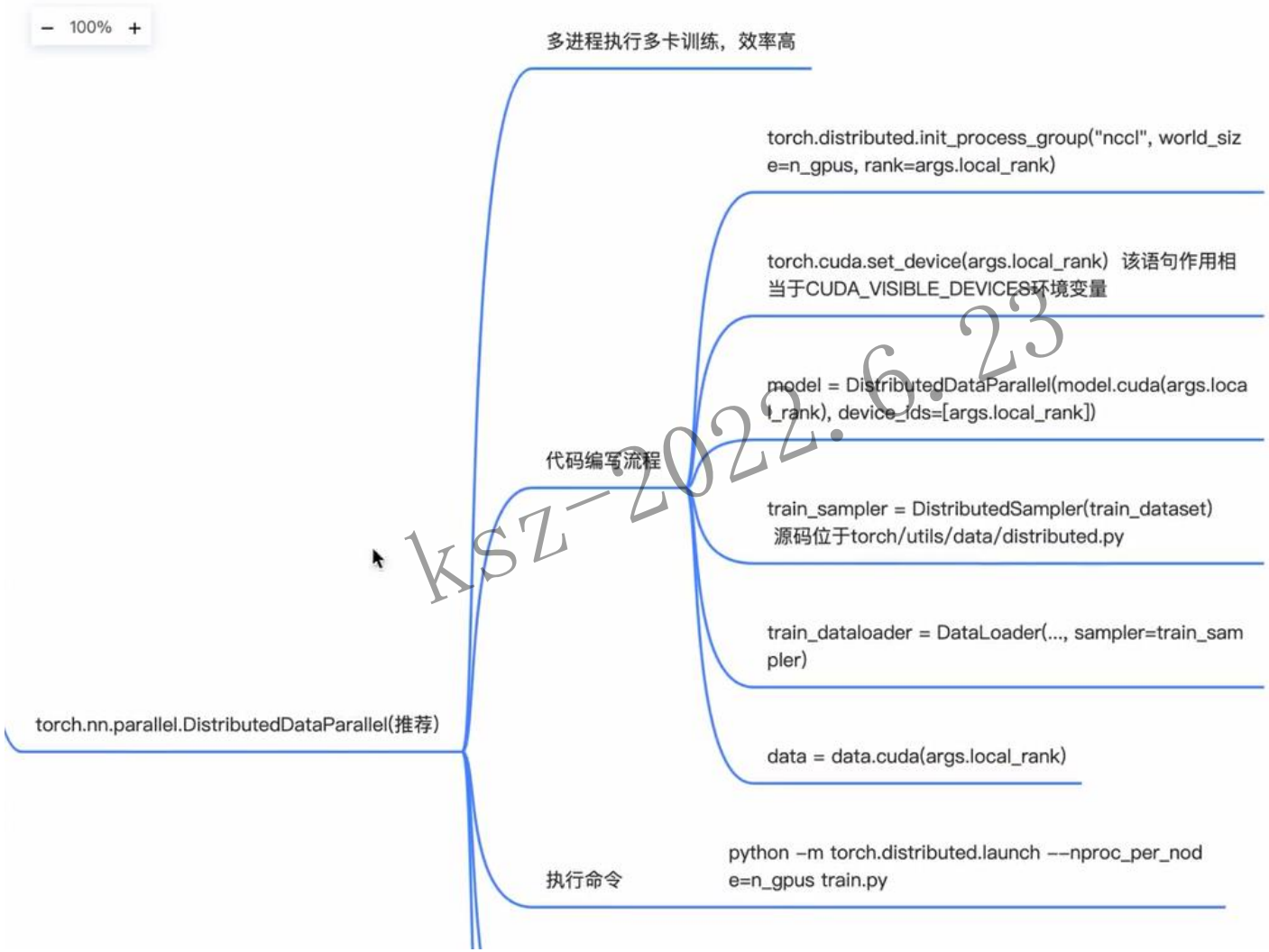
（5）代码中有local_rank = 0这种判断，作用是将一些读写操作全部放到第一个进程中进行处理，防止在不同进程中进行反复操作引发一些问题。

Q&A

简介



简介



简介