Week 4: DDP / DALI Implementation & Profiling

20230071 정다은
20230309 최건

1. Background

본 실습은 Distributed Data Parallel (DDP), 그리고 DDP에 DALI를 결합한 2가지 모드를 구현하고 이를 profiling하여 Data Parallel과 비교 분석하는 과정으로 구성된다.

DDP는 multi-GPU 환경에서 model training을 distribute하게 적용시키기 위한 PyTorch의 주요 기법이다. 기존 data parallel(DP)을 사용한 deep learning workload에서는 master GPU에게 logits가 gather되어야 하고 이러한 과정에서 task간 불균형이 일어나기 쉽다. 이 경우 task가 과부하 된 GPU가 tail latency로 작용하는 경우가 발생하는데, 이를 해소하는 것이 DDP의 핵심 목표다. DDP의 핵심 특징은 one process per one GPU, 즉 GPU당 하나의 독립적인 process를 할당하여 model replica 및 optimizer를 process별로 개별적으로 운용하는 것이다.

DALI는 deep learning 과정에서 흔히 발생하는 data preprocessing에서의 bottleneck을 해결하기 위해 NVIDIA에서 개발한, data loading 및 data preprocessing을 GPU를 통해 가속시킬 수 있는 GPU-accelerated library다. 전통적으로 data preprocessing는 CPU에서 이루어져 training pipeline의 속도 저하를 유발했다. 이를 해결하고자 DALI는 data preprocessing을 CPU에서 GPU로 asynchronous하게 offload함으로써 CPU bottleneck을 효율적으로 줄인다.

2개의 library는 모두 GPU를 최대한 활용하고자 하여 성능을 향상시키고 전체 학습 속도를 높인다는 점에 공통점이 있다. DDP는 multi GPU를 master 없이 사용하여 memory imbalance 문제, communication bottleneck 문제 등을 해결하며, DALI는 GPU를 사용한 data preprocessing acceleration이라는 별도의 영역에서 DDP를 지원한다.

2. Implementation

2-1. Producing Nsight logs (Problem 0)

```
# Scaffold
CUDA_VISIBLE_DEVICES="$LOCAL_GPU_IDS" \
nsys profile \
  --force-overwrite=true \
  --output "$NSIGHT_LOG_DIR/$NSIGHT_FILE_NAME" \
  --trace=cuda,cublas,cudnn,nvtx,osrt \
  python train_cifar.py \
    --num_gpu=$NUM_GPUS \
    --data="$DATA_DIR" \
    --ckpt="$CKPT_DIR" \
    --mode="$MODE" \
    --save_ckpt
```

먼저 scripts/launch.sh 파일을 수정해 nsys-rep 파일이 생성되게 했다. 여기서 nsys-rep 파일을 생성하는 것도 중요하지만, --trace 옵션에 nvtx를 포함하여 epoch, batch에 대한 타이밍 정보를 nsys-rep 파일이 담게 하는 것도 중요하다.

2-2. DDP Implementation (Problem 1~5)

```
def run_process(func, args):
    try:
        mp.set_start_method("spawn", force=True)
    except RuntimeError:
        pass

    nprocs = args.num_gpu
    assert nprocs > 0, "num_gpu must be > 0"
    assert torch.cuda.is_available(), "CUDA is required for DDP (nccl backend)."
    mp.spawn(
        fn=func,
        args=(args,),
        nprocs=nprocs,
        join=True
    )
```

handler/DDP/utils.py/run_process(func, args), Problem 1

먼저 DDP Training Workload Execution을 위한 multi-process를 생성했다. 위와 같이 PyTorch의 mp.spawn을 wrapping하는 역할을 수행하였다. args.num_gpu 변수를 nprocs로 사용하여 GPU 수만큼의

process를 생성하여 one process per one GPU를 지키도록 지정했고, 이 process들이 main_func()를 실행하도록 구현했다. 이는 각 GPU가 독립적인 model replica를 갖는 DDP의 기본 구조를 정립하는 단계이다.

```python
def initialize_group(proc_id, host, port, num_gpu):
    torch.cuda.set_target_device(proc_id) if hasattr(torch.cuda, "set_target_device") else None
    torch.cuda.set_device(proc_id)

    dist_url = f"tcp://{host}:{port}"
    dist.init_process_group(
        backend="nccl",
        init_method=dist_url,
        world_size=num_gpu,
        rank=proc_id,
        timeout=timedelta(minutes=10),
    )
    dist.barrier()
```

handler/DDP/utils.py/initialize_group(proc_id, host, port, num_gpu), Problem 2

프로세스 생성 후, initialize_group() 함수를 통해 distributed communication 환경을 설정했다. DDP는 GPU 간에 gradient 정보를 교환하기 위해 communication이 필수적이다. GPU와 process를 1대1로 매칭하기 위해 torch.cuda.set_device(proc_id)를 호출해 현재 process에 해당하는 고유한 GPU_ID를 할당했다. torch.distributed.init_process_group()에서 GPU 환경에 최적화된 nccl을 backend로 설정했고, communication initialization으로는 TCP를 택했다.

```python
def destroy_process():
    if dist.is_initialized():
        dist.destroy_process_group()
```

handler/DDP/utils.py/destroy_process(), Problem 3

torch.distributed.destroy_process_group()를 간단히 호출하여 현재 process의 GPU distributed group을 종료하고 관련 resource들을 정리한다.

```python
def model_to_DDP(model):
    if not torch.cuda.is_available():
        raise RuntimeError("CUDA is required for DDP.")
    if not dist.is_initialized():
        raise RuntimeError("Process group is not initialized. Call initialize_group first.")

    local_rank = torch.cuda.current_device()
    device = torch.device("cuda", local_rank)

    model = model.to(device, non_blocking=True)

    ddp_model = DDP(model, device_ids=[local_rank], output_device=local_rank,
        find_unused_parameters=False, broadcast_buffers=False)
    return ddp_model
```

handler/DDP/model.py/model_to_DDP(model), Problem 4

torch.cuda.current_device()를 통해 현재 process에 할당된 GPU ID인 local_rank를 확인하고, model.to(device)를 사용하여 model을 GPU 메모리로 이동시켰다. 이후, DDP로 model을 wrapping하여 device_ids=[local_rank]와 output_device=local_rank를 명확히 설정하여 model의 연산이 현재 process가 담당하는 고유 GPU에서만 처리되도록 지정했다.

```python
train_sampler = DistributedSampler(
    train_dataset,
    num_replicas=world_size,
    rank=rank,
    shuffle=shuffle
)
```

```python
train_loader = torch.utils.data.DataLoader(
    train_dataset,
    batch_size=train_batch,
    sampler=train_sampler,
    num_workers=num_workers,
    pin_memory=True
)
```

handler/DDP/cifar10_loader.py/get_DDP_loader(...), Problem 5 중 일부

DDP 환경에서 가장 중요한 고려 사항 중 하나는 데이터 중복이다. 일반 DataLoader를 사용할 경우 각 process가 dataset 전체를 load하여 single epoch에서 동일한 data가 여러 번 사용될 수 있다. 이를 해결하기 위해 handler/DDP/cifar10_loader.py의 get_DDP_loader() 함수에서는 DistributedSam-pler를 도입했다. 이 sampler는 world_size와 rank 정보를 기반으로 dataset을 분할하여, 각 process가 전체 data 중 자신에게 할당된 부분의 data만을 load하도록 보장한다. 이를 torch.utils.data.DataLoader에 sampler 인자로 전달하여 DDP에 최적화된 학습 pipeline을 구축하였다.

2-3. DALI Implementation

```python
def define_graph(self):
    # 1. read images and labels
    images, labels = fn.readers.file(
        file_root=self.data_dir,
        random_shuffle=self.is_train,
        name="Reader",
        shard_id=self.shard_id,
        num_shards=self.num_shards
    )
    # 2. decode
    images = fn.decoders.image(images, device="mixed", output_type=types.RGB)
```

handler/DALI/cifar10_loader.py/CifarPipeline.define_graph(...), Problem 6 중 일부

Class CifarPipeline은 DDP 환경에 맞춰 distributed data loading과 GPU 기반의 preprocessing을 정의한다. Class의 __init__()은 주어진 skeleton code를 그대로 활용하였으며, define_graph() 함수를 template에 맞춰 추가로 구현하였다. Data loading은 fn.readers.file 연산자를 사용하여 이미지와 레이블을 파일 시스템에서 읽어 들였다. 이때, shard_id와 num_shards 인자를 전달하여 DDP process 간 sharding을 자동으로 처리하도록 설정하였다. GPU를 사용한 preprocessing acceleration을 위해서는 fn.decoders.image(device="mixed")를 사용하여 decoding step을 CPU와 GPU가 협력하여 처리하도록 하였다.

이후, 학습 모드 (is_train)에서는 fn.random_resized_crop, fn.flip, 그리고 fn.crop_mirror_normalize()를 사용하여 Torchvision의 표준 증강과 동일한 효과(RandomCrop, RandomHorizontalFlip, Normalize)를 GPU 상에서 구현하였다. 특히, Cutout 증강은 fn.erase를 이용하여 pipeline에 통합되었다. 학습 모드가 아닐 시(검증/테스트)에는 모든 확률적 증강을 제거하고 fn.crop_mirror_normalize(CHW, mean/std)만 적용해 데이터 분포를 보존하며 GPU에서 결정적으로 preprocessing한다.

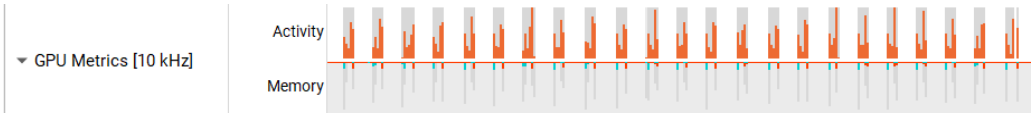
```python
if train_batch > 0:
    pipe = CifarPipeline(train_dir, train_batch, True, cutout,
                         device_id=rank, shard_id=rank, num_shards=world_size,
                         num_threads=num_workers)
    pipe.build()
    dali_iter = DALIGenericIterator([pipe], ["data", "label"], reader_name="Reader")
    train_loader = DALIWrapper(dali_iter)

if valid_size > 0:
    assert valid_batch > 0, "Validation batch size must be > 0"
    pipe = CifarPipeline(valid_dir, valid_batch, False, 0,
                         device_id=rank, shard_id=rank, num_shards=world_size,
                         num_threads=num_workers)
    pipe.build()
    dali_iter = DALIGenericIterator([pipe], ["data", "label"], reader_name="Reader")
    valid_loader = DALIWrapper(dali_iter)

if test_batch > 0:
    pipe = CifarPipeline(test_dir, test_batch, False, 0,
                         device_id=rank, shard_id=rank, num_shards=world_size,
                         num_threads=num_workers)
    pipe.build()
    dali_iter = DALIGenericIterator([pipe], ["data", "label"], reader_name="Reader")
    test_loader = DALIWrapper(dali_iter)
```

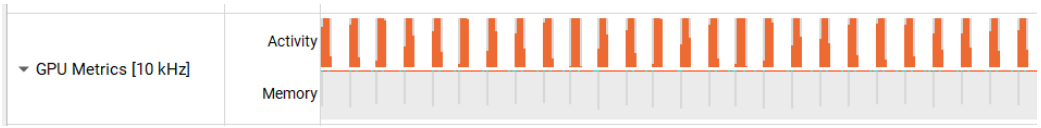
handler/DALI/cifar10_loader.py/get_DALI_loader(...), Problem 7 중 일부

함수 get_DALI_loader(...)는 CIFAR-10을 DALI 파이프라인으로 로드하기 위해 split별 CifarPipeline을 build한 뒤 [pipe]를 DALIGenericIterator, DALIWrapper로 감싸 pytorch와 동일한 (data, label) 인터페이스로 제공한다. 분산 실행행을 전제로 dist.get_world_size()와 dist.get_rank()을 사용하여 num_shards = world_size, shard_id = rank로 sample을 sharding하고 파이프라인의 device_id=rank로 gpu를 지정한다. train_batch가 0보다 클 때는 인자로 받은 cutout을 활성화하여 cutout 포함 학습 증강을 사용한 파이프라인을, validate_batch나 test_batch가 0보다 클 때는 확률적 증강 없이 정규화만 적용된 결정적 파이프라인을 사용하여 loader를 생성한다.

3. Profiling

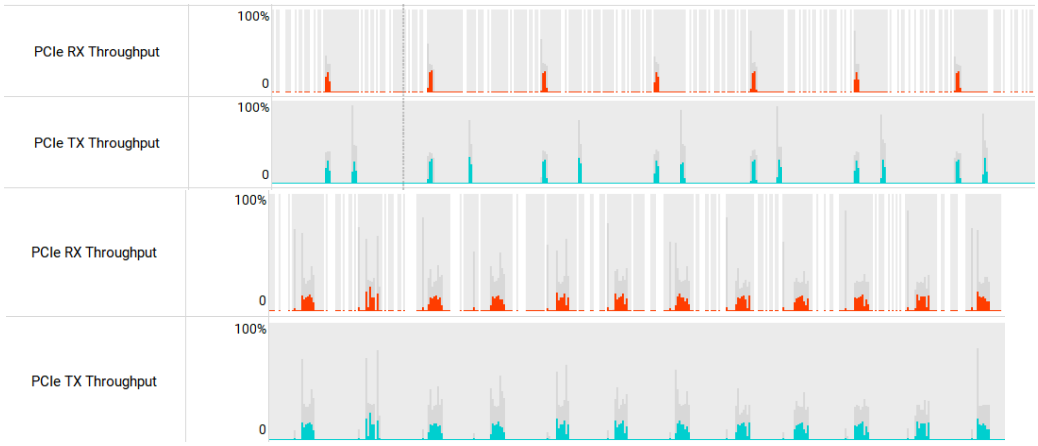
3-1. DP Profiling





(상) gpu_4.nsys-rep의 GPU Metrics 중 일부. (하) gpu_1.nsys-rep의 GPU Metrics 중 일부.

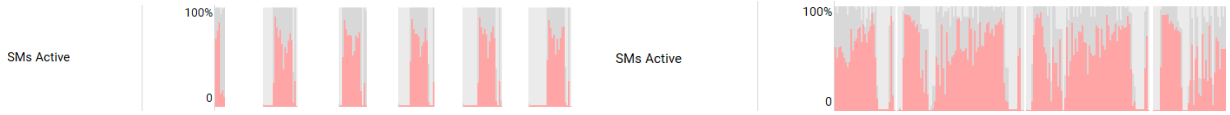
DP 환경에서는 GPU 개수가 많아질수록 memory utilization이 좋지 않게 나타났다. 또한, Batch Time이 1 GPU 환경에서 2.469s, 2 GPU 환경에서 3.206s로 나타났기에, GPU간 communication overhead의 영향이 꽤나 크고 이로 인해 GPU의 memory utilization이 저하되었을 것이라고 해석된다. Epoch compilation time은 직접 측정한 결과 1 GPU에서 42.623s, 2 GPU에서 36.866s로 2 GPU 환경이 근소하게 앞섰으나 결국 DP 방식은 parallelization의 효율이 좋지 못하다는 것을 알 수 있다.



(상) DP에서의 PCIe TX/ RX Pattern. (하) DDP에서의 PCIe TX/RX Pattern

DDP에서는 GPU의 수가 늘어날수록 PCIe/ NVLink 이용률이 증가한다. 1 GPU 환경에서는 GPU 간 communication이 없지만, GPU가 4개인 경우, batch마다 GPU간 communication이 발생해 PCIe/NVLink 이용률이 증가하고 따라서 communication overhead 도 증가하였다. 다만 DDP는 DP와 달리, master GPU에 bottleneck이 생기는 master-gather 방식보다는 각자 GPU들이 개별적으로 필요할 때에만 통신하여 communication을 균등하게 수행할 수 있는 all-reduce/all-gather 방식을 채택했다.

실제로, DDP의 경우, PCIe TX와 RX가 같은 타이밍에 동시에 켜져 양방향으로 send와 recv가 동시에 나는 집단 통신 패턴을 확인할 수 있었지만, DP의 경우, TX만 길게 꼬리가 남는 비대칭을 확인할 수 있다. 또, DDP가 DP에 비해, streaming multiprocessor(SM) active의 값이 높은 것 역시, DP가 DDP에 비해, overhead가 크고, GPU의 idle time이 더 길다는 것을 의미한다. Epoch compilation time를 직접 측정한 결과가 1 GPU에서는 40.738s, 2 GPU에서는 21.319s로, 거의 linear speed-up의 효과를 보았다. 즉, DDP는 master GPU가 없어 bottleneck을 많이 해소시킴에 따라 parallelization 효율이 좋다고 볼 수 있다.



(좌) DDP에서의 SMs Active Pattern. (우) DDP+DALI에서의 SMs Active Pattern.

DDP가 parallelization 효율을 향상시켰다면, DALI를 도입했을 때에는 epoch compilation time이 1 GPU 환경 기준 epoch compilation time이 40.738s에서 22.557s로 크게 단축되었다. 이 성능 향상은 batch마다 반복되는 data loading 및 preprocessing이 DDP 환경에서는 CPU-Bound bottleneck을 일으키며, DALI가 이를 효과적으로 줄이기 때문이다. DALI는 data loading 및 preprocessing을 GPU의 idle SM에게 asynchronous하게 넘긴다. 따라서, SM이 거의 쉬지 않아 전체적인 GPU 활용도가 크게 상승한다.

따라서, GPU computation이 진행되는 동시에 next batch의 data loading 및 preprocessing이 GPU에서 동시에 이루어지는 효과, 즉 data loading이 GPU computation time과 overlap되는 효과가 발생한다. 이는 CPU가 GPU보다 computation 속도가 저열해 생기는 CPU-Bound bottleneck을 개선함과 동시에, GPU 활용률이 극적으로 상승하고 결론적으로 epoch 전체 시간이 크게 단축되는 결과를 가져온다.