W4_DDP_DALI_Training

박한비, 최승희

<Code Implementation>

Problem0. Run DP example

scripts/launch.sh 파일을 수정해서 Nsight 로그(.nsys-rep)가 자동으로 저장되도록 주어진 코드를 수정해야 하는 문제이다. 기존 launch.sh 코드를 보면, 그냥 Python 스크립트를 실행할 뿐, Nsight profiler를 호출하지 않는다. 즉, 코드에 프로파일 로그를 생성할 명령이 포함되어 있지 않다. Nsight 로그를 생성하도록 nsys profle 코드를 추가해서 구현했다.

Problem1. Implement DDP - utils.py, def run_process()

여러 개의 프로세스를 생성해서 DDP 학습을 수행하기 위한 함수이다.
mp.spawn()을 이용해서 각 GPU마다 하나씩 프로세스를 띄운다. 각 프로세스는 proc_id와 args를 받아 독립적으로 실행되며, join=True로 모든 프로세스가 종료될 때까지 메인 프로세스가 기다린다.

Problem2. Implement DDP - utils.py, def initialize_group()

DDP 학습을 위해 모든 프로세스 간의 통신 채널을 설정하고, 각 프로세스가 사용할 GPU를 지정하는 초기화 함수이다.
init_process_group()으로 TCP 기반의 통신 세션을 생성하고, torch.cuda.set_device()를 이용해 proc_id를 기준으로 각 프로세스에 GPU를 하나씩 할당한다.

Problem3. Implement DDP - utils.py, def destroy_process()

DDP 학습이 모두 끝난 뒤, GPU 간 통신 그룹(process group)을 해제하는 함수이다.
dist.destroy_process_group() 함수를 호출해서 init_process_group()으로 생성된 통신 세션을 종료해서, 모든 GPU 간의 연결을 끊고, 메모리를 정리하도록 구현했다.

Problem4. Implement DDP - model.py, def model_to_DDP()

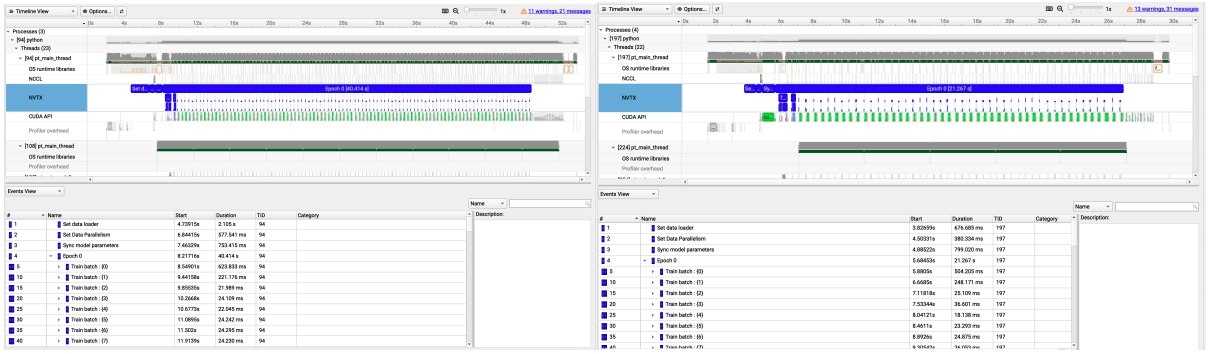
현재 프로세스에 할당된 GPU로 모델을 이동시키고, 해당 모델을 DDP로 감싸서 여러 프로세스/GPU 간의 gradient sync가 자동으로 이루어지게 한다.
CUDA가 없는 환경에서는 DDP를 쓸 수 없기 때문에, 모델을 그대로 반환한다.
한 프로세스가 여러 GPU를 자동으로 쪼개서 사용하는 DP와 달리, DDP는 여러 프로세스가 각자 자신의 GPU를 하나만 사용하며 통신으로 동기화한다. 따라서, DDP에서는 각 프로세스가 사용할 정확한 GPU 번호(device id, local rank)를 지정해준다.

Problem5. Implement DDP - cifar10_loader.py, def get_DDP_loader()

기본 data loader를 사용하면, 프로세스들이 같은 sample data를 중복으로 읽게 되어 비

효율적이다. 따라서, 공평한 training을 위하여 data가 적절히 분배되도록 data loader를 구현한다. 기존의 DistributedSampler를 활용할 수 있다.
DistributedSampler를 사용해서 전체 데이터셋을 프로세스 수로 나누어 각 프로세스가 서로 다른 샘플을 받게 한다. Dataset들을 구성 후 DistributedSampler를 이용하여 sampler를 구성한다. 해당 sampler를 활용하여 각 data loader(train_loader, valid_loader, test_loader)를 생성, 반환한다.

<Profiling of DDP Implementation>
구현을 마친후, run_cluster_ddp.sh를 실행하여 얻어낸 nsight log를 profile한 결과는 아래와 같다. 첫번째 사진은 GPU 수가 1일 때, 두번째 사진은 GPU 수가 2일 때의 결과이다.



GPU 수	Set data loader	Set data parallelism	Sync model parameters	Epoch 시간(s)
1	2.105s	577.541ms	753.415ms	40.414s
2	676.685ms	380.334ms	799.020ms	21.267s

Sync model parameters를 제외한 대부분의 function의 호출 시간이 GPU 개수가 증가했을 때 대폭 감소한 것을 확인할 수 있다. 이는 GPU끼리 일을 분배하여 각 프로세스가 할 일의 양이 줄어들기 때문이다. Sync model parameters는 GPU 개수가 아닌, 모델 크기(파라미터 수)에 비례하여 호출 시간이 결정된다.

Problem6. Implement DALI - cifar10_loader.py, class CifarPipeline
해당 클래스는 DALI를 위해 필요한 data 처리 파이프라인을 구현한다. 해당 파이프 라인은 DP/DDP의 파이프라인과 동일하게 동작할 수 있도록 한다.
DDP의 CifarPipeline을 참고하며 구현하였다. Init 함수에서는 각 필드가 전달받은 파라미터로 초기화되도록 한다.
define_graph 함수에서는 images, labels을 데이터 소스가 디렉터리 이미지면 fn.readers.file, 데이터 소스가 CIFAR-10 포맷이면 fn.decoders.image으로 images를 전달받아 사용한다. 전달받은 images를 적절하게 Padding, Flip, Crop, Mirror, Nomalize, Cutout 하여 반환해준다.

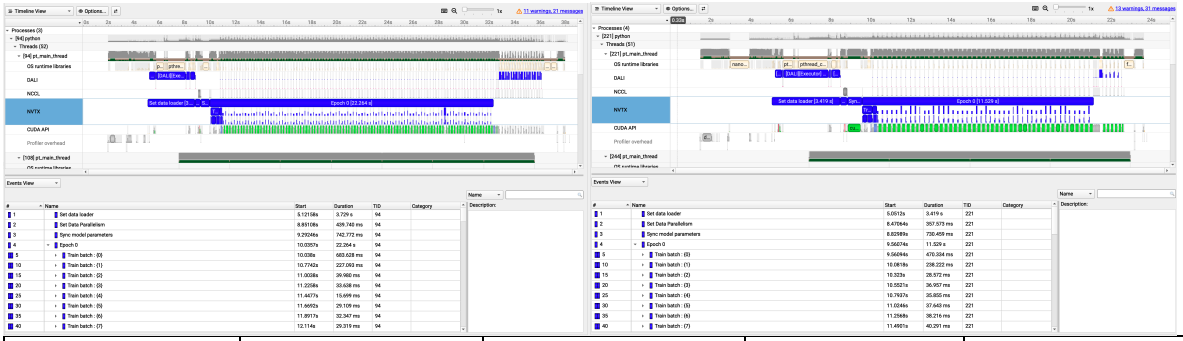
Problem7. Implement DALI - cifar10_loader.py, def get_DALI_loader()

해당 함수에서는 파이프라인을 Iterator로 감싸고, 과제에서 제공되는 DALIWrapper로 다른 loader와 동일한 인터페이스처럼 동작하도록 만들어 반환해야 한다.

CifarPipeline 함수로 파이프라인 인스턴스는 process 별로 생성하도록 하고 해당 파이프라인과 DALIGenericIterator를 이용해 Iterator를, DALIWrapper를 이용해 data loader를 제작한다.

<Profiling of DALI Implementation>

구현을 마친후, run_cluster_ddp_dali.sh를 실행하여 얻어낸 nsight log를 profile한 결과는 아래와 같다. 첫번째 사진은 GPU 수가 1일 때, 두번째 사진은 GPU 수가 2일 때의 결과이다.



GPU 수	Set data loader	Set data parallelism	Sync model parameters	Epoch 시간(s)
1	3.729s	439.740ms	742.772ms	22.264s
2	3.419s	357.573ms	730.459ms	11.529s

데이터와 연산이 랭크 별로 분할되기 때문에 epoch 시간은 크게 줄으나, set data loader와 set data parallelism는 각 랭크에서 일어나야 하기 때문에 GPU 개수에 따라 크게 변하지 않는 것을 확인할 수 있다. 또 Sync model parameters는 GPU 개수가 아닌, 모델 크기(파라미터 수)에 비례하여 호출 시간이 결정된다.

<Profile GPU/Mem Utils with given nsight logs>

1. **DP**

GPU 수	GPU Util (%)	Mem Util (%)	Epoch 시간(s)
1	99.0	1.0	30.398
2	98.1	1.9	32.169
4	97.1	2.9	34.362

GPU utilization은 GPU 수가 늘어날수록 조금씩 감소하는 경향을 보였지만,

Memory Utilization은 조금씩 상승했다. 이는 한 프로세스가 여러 장치를 순차적으로 커

널 런치하는 구간이 끼면 미세한 비활성 구간이 생기고, GPU를 늘릴수록 총 메모리 사용량이 증가하기 때문이다.
또한 DP에서는 각 업무들이 GPU의 각 프로세스들에 병렬적으로 들어가도록 구현이 되어있지 않아 GPU 개수가 늘어나도 epoch 시간이 오히려 증가하는 모습을 보인다.

2. DDP

GPU 수	GPU Util (%)	Mem Util (%)	Epoch 시간(s)
1	98.5	1.5	32.4
2	96.9	3.1	17.5
4	95.8	4.2	13.5

GPU utilization은 GPU 수가 늘어날수록 조금씩 감소하는 경향을 보였지만, DDP 단독에 비해 더 높은 수준을 유지했다. 이는 DALI가 CPU 기반 데이터 로딩 과정을 GPU 내부에서 처리함으로써, 데이터 전송 지연 구간을 최소화했기 때문이다. 따라서 GPU 간 통신으로 인한 delay는 존재하지만,데이터 입출력 bottleneck이 줄어서 GPU가 더 안정적으로 연산을 지속할 수 있게 되어, gpu util이 전반적으로 증가한 것으로 보인다.
GPU 수가 늘어날수록 Memory Utilization은 0.7% → 2.4% → 4.9%로 상승했다. 이는 각 GPU가 독립적인 DALI 파이프라인을 유지하기 위해 추가적인 버퍼와 데이터 캐시를 확보해야 하기 때문이다. multi GPU 환경에서는 DALI가 각 GPU별로 데이터를 병렬로 전처리하기 때문에 mem util이 증가하는 것은 자연스러운 결과라고 해석했다.

3. DALI

GPU 수	GPU Util (%)	Mem Util (%)	Epoch 시간(s)
1	99.3	0.7	10.65
2	97.6	2.4	7.19
4	95.1	4.9	6.72

GPU utilization은 GPU 수가 늘어날수록 조금씩 감소하는 경향을 보였지만, DDP 단독에 비해 더 높은 수준을 유지했다. 이는 DALI가 CPU 기반 데이터 로딩 과정을 GPU 내부에서 처리함으로써, 데이터 전송 지연 구간을 최소화했기 때문이다. 따라서 GPU 간 통신으로 인한 delay는 존재하지만,데이터 입출력 bottleneck이 줄어서 GPU가 더 안정적으로 연산을 지속할 수 있게 되어, gpu util이 전반적으로 증가한 것으로 보인다.
GPU 수가 늘어날수록 Memory Utilization은 0.7% → 2.4% → 4.9%로 상승했다. 이는 각 GPU가 독립적인 DALI 파이프라인을 유지하기 위해 추가적인 버퍼와 데이터 캐시를 확보해야 하기 때문이다. multi GPU 환경에서는 DALI가 각 GPU별로 데이터를 병렬로 전처리하기 때문에 mem util이 증가하는 것은 자연스러운 결과라고 해석했다.