M3239.005400 데이터사이언스를 위한 컴퓨팅 2 (001)

Homework #7

Due : 2024/12/15 (Sun)

2024-28413 이정현

1. DeepSpeed

-DeepSpeed의 ZeRO Stage 1, 2, 3, Offload에 대해 설명하라. 각각의 Stage에서 optimizer state, gradient, parameter 값들이 forward, backward, optimizer step 단계에서 어떻게 관리되는지, 그리고 N 개의 GPU가 동일하게 갖고 있는지 또는 나누어 갖고 있는지를 기술해야 한다. 또한, forward, backward, optimizer step 단계에서 어떤 데이터(optimizer state, gradient, parameter)가 통신되고, 이 통신이 어떤 방식(예: collective communication 종류)으로 이루어지는지를 설명하라.

ZeRO는 Large Model Training을 하는 과정에서 GPU 메모리 사용량과 통신량을 최소화하기 위해 만들어진 기법이다. 기본 아이디어는 중복 데이터를 최소화하기 위해 optimizer state, gradient, parameter를 여러 GPU에 걸쳐 분산(sharding)하는 것이다. 즉, 이전에는 모든 GPU가 모든 optimizer state, gradient, parameter를 복사해서 가지고 있었다면(data parallelism or model parallelism), ZeRO는 이러한 중복을 제거하여 memory efficiency를 극대화한다.

ZeRO는 크게 세 가지 Stage로 나누어지며, Stage가 높아질수록 더 많은 유형의 데이터가 sharding된다. 또한 ZeRO-Offload는 CPU나 NVMe 스토리지로 일부 메모리를 오프로딩(offload)하여 GPU 메모리 사용을 추가로 줄인다.

[ZeRO Stage 1 : Optimizer State Sharding]

이 Stage에서의 핵심은 optimizer state만 분산된다는 것이다. Optimizer step에서 사용하는 momentum과 같은 optimizer related state들이 N개의 GPU에 나눠 저장되는 방식을 사용한다. Gradient와 parameter는 여전히 모든 GPU가 전부 가지고 있게 된다.

Forward Pass : 모든 GPU가 동일한 parameter를 가지고 forward pass를 수행하게 된다. 이 과정에서 optimizer state는 직접적인 관여가 없기 때문에 optimizer state 관련된 통신은 없다고 봐도 되고, 또한 parameter들도 모든 GPU에 동일하게 존재하고 있기 때문에 forward pass에서 parameter 통신도 없다고 할 수 있다.

Backward Pass : 모든 GPU는 local로 계산한 gradient를 얻는다. 이때, 모든 GPU는 동일한 parameter를 가지고 있으므로 local로 backward를 통해 gradient를 얻을 수 있다. Gradient들은 모든 GPU에 복제되어 있으므로, backward 후 gradient를 사용하는 단계에서 각 GPU는 자기 shard에 해당하는 optimizer state를 가져와야 한다. 또한, backward 완료 후 optimizer step을 위해서 gradient를 gather하거나 optimizer state shard를 broadcast 받는 통신이 필요할 수 있다. 이때 주로 **All-Reduce**로 모든 GPU의 gradient를 평균화한다(All-Reduce는 collective 통신).

Optimizer Step : 여기서는 optimizer state (ex. Adam's m, v parameters) 들이 GPU사이에 sharing 되어있는 상태이다. 각 GPU는 자신이 가진 optimizer state shard에 해당하는 parameter, gradient 값을 모아서 업데이트를 진행한다. 이후 파라미터를 모든 GPU에 다시 동기화해야 하므로 parameter 업데이트 후 **All-Gather 또는 All-Reduce** 기반의 통신이 발생한다.

[ZeRO Stage 2 : Optimizer State & Gradients Sharding]

이 Stage에서의 핵심은 optimizer state와 더불어 gradients까지 분산된다는 것이다.
Parameter는 여전히 모든 GPU가 전부 가지고 있게 된다.

Forward Pass : 여전히 모든 GPU가 파라미터 전체를 가지고 forward pass를 수행하게 된다. Optimizer state, gradient는 이 단계에서 직접적인 관여가 없기 때문에 특별한 통신은 없다고 할 수 있다.

Backward Pass : Stage 2에서는 gradient sharding도 되어있기 때문에 local backward를 통해 구한 gradient는 부분적으로만 가지게 되거나, 전체 gradient를 구하기 위해 통신을 통해 나누어진 shard를 각 GPU가 자신의 shard 부분만 유지하게 된다. 정리하자면, backward 이후 All-Reduce 대신에 sharded reduction을 수행한다. 이를 통해서 최종적으로 각 GPU가 자신의 parameter shard에 대응하는 gradient shard만 보유하도록 한다. 이 과정에서 **All-To-All** 또는 **Scatter-Reduce** 등의 collective 통신이 이용될 수 있다.

Optimizer Step : Optimizer state와 gradient 모두 sharding되어 있으므로, 각 GPU는 자신이 담당하는 parameter shard, optimizer state shard, 그리고 gradient shard를 가지고 local 업데이트를 수행한다. 업데이트 후에도 parameter는 모든 GPU에 동일하게 유지되어야 하므로, 업데이트된 parameter shard를 **All-Gather**를 통해 다시 모든 GPU에 복제한다.

[ZeRO Stage 3 : Optimizer State & Gradients & Parameter Sharding]

이 Stage에서의 핵심은 optimizer state, gradients, 그리고 parameters까지 분산되어 모든 중복을 제거한다는 것이다. 즉, optimizer state, gradients, parameters 모두를 N개의 GPU에 분산 저장하고, 필요한 시점에만 해당 shard를 gather하여 사용하게 되는데, 이는 GPU memory use를 많이 줄일 수 있지만 통신 복잡도가 증가한다는 단점이 있다.

Forward Pass : Parameter들이 각 GPU에 나뉘어 있으므로, forward pass시 필요한 parameter shard만 가져와 local 계산을 수행해야 한다. 특정 레이어를 처리할 때 해당 레이어의 parameter shard가 없는 GPU는 다른 GPU로부터 All-Gather를 통해 필요한 parameter를 가져와서 forward 계산을 수행한다. 이때 사용되는 통신은 레이어별로 필요한 parameter를 gather하는 **All-Gather** collective 통신이다.

Backward Pass : Backward pass시 gradient 역시 sharding된 상태이며, 해당 레이어에 대응하는 parameter shard를 가진 GPU가 그 gradient shard를 업데이트한다. 또한, 필요하다면 gradient를 다른 GPU로 redistribute하는 과정도 발생한다. 이때도 **Reduce-Scatter**나 **All-To-All** 패턴의 collective 통신을 통해 gradient shard를 재분배한다.

Optimizer Step : Optimizer state 역시 sharding되어 있으므로, 각 GPU는 자신이 담당하는 parameter shard와 그에 대응하는 optimizer state shard, 그리고 gradient shard를 가지고 local 업데이트를 수행한다. 업데이트 후에도 parameter를 다시 All-Gather하지 않고, parameter shard는 각 GPU에 그대로 남는다. 즉, parameter 자체를 모두 복제하지 않고 sharding 상태로 유지한다. 필요할 때마다 forward pass에서 All-Gather를 통해 가져와 사용한다.

[ZeRO Stage Offload]

ZeRO Stage 3까지 구현한 이후에도 GPU memory가 아직도 부족한 상황이 생길 수 있다. 이를 해결하기 위해 일부 parameter, optimizer state 등을 CPU memory나 NVMe SSD로 offloading하여 GPU memory 사용량을 추가 감소시킨다. (Ex. Parameter나 optimizer state shard를 GPU에서 CPU memory로 옮겨두고, 필요할 때만 GPU로 가져오기)

Forward Pass / Backward Pass : 필요한 시점에 CPU나 NVMe에서 GPU로 parameter shard를 로딩(**All-Gather 혹은 point-to-point** 전송)하고, 계산이 끝나면 다시 CPU/NVMe로 옮긴다.

Optimizer Step : 이때에도 optimizer state를 CPU/NVMe에서 불러와 GPU에서 업데이트 후 다시 offloading할 수 있다.

이 과정에서 GPU-CPU 간 통신은 주로 **point-to-point**(CUDA-aware MPI나 NCCL + CPU 메모리 복사), NVMe는 파일 I/O 형태로 관리된다. (GPU 와 CPU/NVMe 사이의 전송이 추가되어, 이 때는 Collective 통신보다는 Host-Device copy나 I/O 연산 사용.)

-ZeRO Stage 1, 2, 3 각각에 맞는 Deepspeed Configuration 값을 확인하고, 각 설정 값이 의미하는 바를 설명하라. 또한, FP16 및 CPU Offloading을 활성화하기 위해 필요한 추가 argument들을 명시하고, 각 설정이 어떤 역할을 하는지 설명하라.

<공통 설정>

-train_batch_size : 전체 GPU와 gradient accumulation step을 포함한 global batch size

-train_micro_batch_size_per_gpu : 각 GPU당 1 step에서 처리하는 micro batch size

-optimizer : 사용할 optimizer와 해당 parameter를 지정

-zero_optimization : ZeRO 최적화를 활성화하고 단계를 지정

[ZeRO Stage 1 : Optimizer State Sharding]

```
# ZeRO Stage 1: Optimizer State Partitioning
stage_1_config = {
    "train_batch_size": args.global_batch,
    "train_micro_batch_size_per_gpu":args.micro_batch,
    "zero_optimization": {
        "stage": 1,
        "reduce_scatter": True,
        "allgather_partitions": True
    },
    "optimizer": adam_optimizer,
}
```

-**“stage”: 1** -> optimizer state (ex. Adam’s m, v parameters)만 Sharding

-**“reduce_scatter”: True** -> 그라디언트 합산 시 reduce-scatter를 통해 메모리 효율화

-**“allgather_partitions”: True** -> 필요할때만 partition을 All-Gather해서 optimizer 업데이트 수행

[ZeRO Stage 2 : Optimizer State & Gradients Sharding]

```
# ZeRO Stage 2: Optimizer State, Gradient Partitioning
stage_2_config = {
    "train_batch_size": args.global_batch,
    "train_micro_batch_size_per_gpu":args.micro_batch,
    "zero_optimization": {
        "stage": 2,
        "allgather_partitions": True,
        "allgather_bucket_size": 5e8,
        "reduce_scatter": True,
        "overlap_comm": True,
    },
    "optimizer": adam_optimizer,
}
```

-**“stage”: 2** -> optimizer state + gradient sharding

-**“allgather_partitions”: True** -> parameter, optimizer state, gradient를 필요할 때만 all-gather

-**“allgather_bucket_size”: 5e8** -> All-Gather할 때 한 번에 모으는 parameter/gradient의 양을 제한하여 communication overhead 조절

-**“reduce_scatter”: True** -> gradient 합산에 reduce-scatter 사용

-**“overlap_comm”: True** -> communication과 computation을 겹쳐서 성능 향상 시도

[ZeRO Stage 3 : Optimizer State & Gradients & Parameter Sharding]

```
# ZeRO Stage 3: Optimizer State, Gradient, Parameter Partitioning + Offloading to CPU
stage_3_config = {
    "train_batch_size": 64,
    "zero_optimization": {
        "stage": 3,
        "contiguous_gradients": True,
        "reduce_bucket_size": 5e8,
        "sub_group_size": 1e9,
    },
    "optimizer": adam_optimizer,
}
```

-**“stage”: 3** -> optimizer state, gradient, parameter sharding

-**“contiguous_gradients”: True** -> memory에 gradient를 연속적으로 배치, 성능 최적화

-**“reduce_bucket_size”: 5e8** -> reduce-scatter시 한 번에 처리하는 데이터 양 제한

-**“sub_group_size”: 1e9** -> parameter/gradient를 sub-group으로 나눠 통신 최적화

[FP16]

FP16(Mixed Precision Training)을 사용하면 메모리 사용량과 연산량을 줄일 수 있다.

활성화하기 위해 필요한 추가 Argument (코드 실행시) : **—dtype fp16**

코드 내부에서 추가적으로 사용되는 Argument

-**“enabled”: true** -> Mixed Precision Training 활성화한다

-**“loss_scale”** -> 손실 스케일링 방식을 지정하는 값. 0이면 자동 손실 스케일링(automatic loss scaling) 기법을 사용하고, 0이 아닌 양의 값으로 설정하면 해당 값이 고정 손실 스케일(Static Loss Scale)로 사용

-**“initial_scale_power”** -> 초기 손실 스케일(loss scale)을 2^(initial_scale_power) 형태로 설정

-**“loss_scale_window”** -> 손실 스케일을 조정하는 빈도를 결정

-**“hysteresis”** -> 손실 스케일 감소 시 적용되는 히스테리시스(hysteresis) parameter. 동적으로 손실 스케일을 줄여야 할 때, 즉 언더플로우가 발생했을 때 바로 줄이지 않고, 일정 횟수 확인 후 실제로 감소시키는 지연 매개변수

-**“min_loss_scale”** -> 손실 스케일의 최소값을 설정

[CPU Offloading]

Parameter나 optimizer state를 GPU memory 대신 CPU memory에 저장해서 GPU memory 사용량을 크게 낮출 수 있다.

활성화하기 위해 필요한 추가 Argument (코드 실행시): **—offload**

코드 내부에서 추가적으로 사용되는 Argument

-**“offload_optimizer”: { “device”: “cpu” }** -> optimizer state를 CPU로 offloading

-**“offload_param”: { “device”: “cpu” }** -> parameter파라미터를 CPU로 offloading

-**“pin_memory”: true** -> CPU memory pin으로 Host-Device 전송 효율 상승

-실행 파일(run.sh)의 1번부터 6번까지의 command에 따라 ZeRO Stage에 따른 GPU 메모리 사용량을 확인하라. 각 실행 단계에서 forward, backward, optimizer step 동안의 메모리 사용량을 체크하고, 만약 중간에 문제가 발생한다면 그 이유를 설명하라. torch.cuda.memory allocated()를 사용하면 GPU 메모리 사용량을 확인할 수 있다.

[./run.sh 1 : zero1]

Step 150 부근에서 time limit 걸리고 종료. (slurm세팅으로 인한 timeout)

Step	GPU Memory Used (bytes)		
	Forward	Backward	Optimizer Step
0	15150913536	10179905024	10179905024
10	20256870912	10179905024	10179905024
20	20256870912	10179905024	10179905024
30	20256870912	10179905024	10179905024
40	21032012288	10954631680	10954631680
50	21031750144	10954631680	10954631680
...	...	...	...
150	21031316480	10954198016	10954198016

[./run.sh 2 : zero2]

Step 40 부근에서 time limit 걸리고 종료. (slurm세팅으로 인한 timeout)

Stage 1과 비교했을 때 더 많은 메모리 감소를 목격할 수 있었다. 특히, Backward 이후 메모리 사용량 감소가 두드러졌다. (Gradient shard만 남기고, 나머지 부분은 다른 GPU들로 분산)

Step	GPU Memory Used (bytes)		
	Forward	Backward	Optimizer Step
0	15150913536	5477366272	5477366272
10	15553698304	5477366272	5477366272
20	15553698304	5477366272	5477366272
30	15553698304	5477366272	5477366272
40	16328505856	6251397120	6251397120

[./run.sh 3 : zero2 - offload]

Step 30 부근에서 time limit 걸리고 종료. (slurm세팅으로 인한 timeout)

GPU memory에 올라와있는 데이터가 줄어들어 메모리 사용량이 더 줄어드는 것을 목격했다. (그냥 Stage 2에 비해)

Step	GPU Memory Used (bytes)		
	Forward	Backward	Optimizer Step
0	15020889600	4953077760	4953077760
10	15029409792	4953077760	4953077760
20	15029409792	4953077760	4953077760
30	15029409792	4953077760	4953077760

[./run.sh 4 : zero3]

Step 50 부근에서 time limit 걸리고 종료. (slurm세팅으로 인한 timeout)

Stage 2와 비교했을 때 더 큰 memory 감소를 기대했으나, 결과는 예상과 달랐다.

Stage 3에서는 forward 시 필요한 shard를 All-Gather 해야 하므로 communication overhead가 증가한다. 이때 All-Gather로 인해 임시적으로 메모리가 더 많이 사용될 수 있다. (All-Gather한 parameter를 GPU에 올려두는 과정에서 이전 단계보다 큰 임시 버퍼나 통신 버퍼가 필요) 또한, Bucket size 설정(reduce_bucket_size, sub_group_size 등)이 비효율적으로 되어 있다면, 일시적으로 메모리 사용량이 증가할 수 있다고 생각하였다.

Step	GPU Memory Used (bytes)		
	Forward	Backward	Optimizer Step
0	15089373184	5026804224	5540048384
10	18901569024	5803241472	5540048384
20	18896326144	5803241472	5540048384
30	18896326144	5803241472	5540048384
40	18901569024	5803241472	5540048384
50	18896326144	5803241472	5540048384

[./run.sh 5 : zero3 - offload]

Step 30 부근에서 time limit 걸리고 종료. (slurm세팅으로 인한 timeout)

GPU memory에 올라와있는 데이터가 줄어들어 메모리 사용량이 더 줄어드는 것을 목격했다. (그냥 Stage 3에 비해)

그리고 지금까지 실험한 1,2,3,4 중에서 가장 적은 메모리를 이용하는 것을 확인하였다. (효율적인 메모리 사용)

Step	GPU Memory Used (bytes)		
	Forward	Backward	Optimizer Step
0	13925452800	3863107072	3602097152
10	16969647104	3865513472	3602097152
20	16969647104	3865513472	3602097152
30	16969647104	3865513472	3602097152

[./run.sh 6 : zero3 - fp16]

Step 90 부근에서 time limit 걸리고 종료. (slurm세팅으로 인한 timeout)

FP16을 사용하면 parameter 및 gradient size가 절반 정도로 줄어들어 메모리 사용량이 크게 감소하는 것을 확인할 수 있었다.

Step	GPU Memory Used (bytes)		
	Forward	Backward	Optimizer Step
0	7872385024	2745069568	2613789696
10	9389042688	2750467072	2617984000
20	9389042688	2750467072	3395133952
30	10164989440	3526413824	3395133952
40	10164989440	3526413824	3395133952
...	...	...	...
90	10164989440	3526413824	3395133952

-ZeRO Stage에 따른 20 iteration 동안의 학습 시간을 측정하라. 실행 파일(run.sh)의 1번부터 6 번까지의 command에 따라 학습시간을 측정하고, 중간에 문제가 발생하면 그 원인을 기술하라.

Command	Model	TIme (sec)
1	Stage 1	28.5311346054077
2	Stage 2	104.270375490189
3	Stage 2 + offloading	109.116225481033
4	Stage 3	102.268945932388
5	Stage 3 + offloading	139.791724443436
6	Stage 3 + FP16	59.0305497646332

6개의 ZeRO 모델에 대해서 측정한 시간은 다음과 같았다. (20 step 돌면 break 하는 코드를 추가해서 실험하였음)

일반적으로 ZeRO Stage가 올라갈수록 메모리 사용 효율은 개선되나, 그에 따라 복잡한 통신(collective communication), parameter sharding, All-Gather/Reduce-Scatter 연산 등이 추가되어 computation 및 communication 오버헤드가 증가한다.

-Stage 1 : Optimizer state만 분산됨으로 통신량이 비교적 적으며 메모리 절감 효과는 있으나 오버헤드는 최소화되어 속도가 빠르다

-Stage 2, 3 : 여기서는 Gradient와 Parameter까지 sharding하면서 통신 패턴이 복잡해지고, 각 스텝마다 필요한 parameter를 All-Gather하는 등의 추가 통신과 동기화 비용이 발생한다. Stage 2(약 104초), Stage 3(약 102초)는 Stage 1 대비 3배 이상의 시간이 소요되었다.

-Offloading 적용 : CPU Offloading은 GPU memory를 절감하지만, CPU-GPU 간 데이터 전송 오버헤드가 증가한다. Stage 2에서 Offload 추가 시 104.27초 → 109.12초, Stage 3에서 Offload 추가 시 102.27초 → 139.79초로 더욱 느려지는 결과를 보였다. 이는 CPU memory로 parameter / optimizer state를 이동시키는 추가 I/O 시간이 발생하기 때문이다.

-Stage 3 + FP16 : parameter 및 gradient의 메모리 사용량을 줄이고, 데이터 전송량도 절반 수준으로 줄어든다. 이로 인해 Stage 3 단독(102초) 대비 약 절반 정도(59초)로 시간이 크게 단축되었다. FP16이 communication 및 memory 사용량 감소에 긍정적인 영향을 미친다는 사실을 직접 확인할 수 있었다.