Distributed Training 101

https://huggingface.co/spaces/nanotron/ultrascale-playbook?section=high-level_overview

이승한 (Seunghan Lee)

1. Introduction

1. Introduction

[Motivation] LLM을 학습하려면 multi-GPU 필요 → **GPU cluster orchestration**필요

누구나 Llama, DeepSeek 같은 최신 모델을 다운로드하고 논문을 읽을 수 있음

→ 하지만 진짜 어려운 부분은 **"이 모델을 어떻게 학습했는가?"**

핵심

- GPU를 어떻게 나눠 쓰는지 (**Parallelism**)
- 메모리를 어떻게 최적화하는지 (**ZeRO, kernel fusion**)
- **분산 학습** 코드를 어떻게 작성하는지

1. Introduction

분산 학습 지식 & 코드를 정리해주는 오픈소스 가이드

- GPU 1개 → 10개 → 100개 → 1000개로 확장하는 방법 설명
- 이론 → 코드 예제 → 실험 벤치마크까지!

핵심 주제들 (Pretraining 뿐 아니라 Fine-tuning에도 필수적)

- **Data/Tensor/Pipeline/Context** Parallelism

→ 여러 GPU에 연산을 나눠서 병렬 처리하는 다양한 기법

- **ZeRO, Kernel Fusion**

→ 메모리 절약과 속도 향상을 위한 기법

1. Introduction

이 책의 3가지 토대

1. Quick intro to theory

- 이론적 직관: Transformer의 어느 부분이 메모리를 많이 먹는지, 학습 중 언제 메모리 피크가 오는지 등
  - 예: “Feedforward layer가 hidden dim 때문에 메모리를 가장 많이 차지한다.”

2. Clear code implementations

- 교육용 단순 코드: **Picotron**
- 실제 프로덕션 코드: **Nanotron**

3. Real benchmarks

- 단일 레시피는 없다! (네트워크, 인프라에 따라 다름)
- 다양한 실험 결과(최대 512 GPU) 제공 → 독자가 자기 환경과 비교 가능

1. Introduction

[Overview] 이 책에서 다루는 기술들은 **아래 세 가지 핵심 문제**를 해결하기 위한 것

1. Memory usage

- 한 번의 training step이 GPU 메모리에 안 들어가면 → 학습 자체가 불가능!
- Ex) batch size, model size가 너무 크면 → OOM 발생

2. Compute efficiency

- GPU가 가능한 한 “계산”에 집중하도록! (데이터 전송, idle 등으로) 낭비되는 시간을 줄여야 함
- Ex) GPU가 바쁘게 행렬 곱만 하고 있으면 최고 효율 (통신 대기 시간은 낭비)

3. Communication overhead

- 여러 GPU를 쓸 때 필연적으로 GPU 간 통신이 필요. 하지만 통신 중에는 GPU는 idle 상태 → 성능 낭비
- 해결: intra-node (빠른 NVLink)와 inter-node (느린 InfiniBand) 통신 효율을 최대로 활용 & Overlap 있도록

1. Introduction

Trade-off: 많은 기법들이 **메모리-계산-통신** 사이에서 하나를 희생해 다른 것을 개선

- Ex) **Recomputation**: 중간 activation을 저장하지 않고, 필요할 때 다시 계산

  → (장) 메모리 절약, (단) 계산량 증가

- Ex) **Tensor Parallelism**: 텐서를 쪼개 여러 GPU에서 동시에 계산

  → (장) 속도 향상, (단) 통신량 증가

- **정답은 절대 하나가 아니다!** → 인프라 환경에 따라 최적의 균형을 찾는 것이 핵심!

Summary: LLM 학습의 병목은 **메모리 / 계산 효율 / 통신 비용** 세 가지

- 모든 병렬화 및 최적화 기법은 결국 이 세 축 사이에서 **trade-off를 조절**하는 것

2. First Steps: Training on One GPU

2-1. Overview

2-1. Overview

1. 기본적인 학습 루프
2. Batch size (bs)
3. Batch size의 trade-off
4. Tokens 기준의 batch size
5. 첫 번째 challenge: OOM

2-1. Overview

1. 기본적인 학습 루프

- 모델을 단일 GPU에서 학습할 때는 크게 3단계가 반복
  - Step 1) **Forward** pass
  - Step 2) **Backward** pass
  - Step 3) **Optimization** step
- 분산학습도 결국 **위 세 과정을 분산**하는 것

2-1. Overview

2. Batch size (bs)

- 정의: 한 번의 forward/backward에서 처리하는 sample 개수
  - **작으면**: Gradient가 noisy

    → 학습 초반에 landscape를 빠르게 탐색, but 후반에는 수렴 불안정
  - **크면**: Gradient가 정확

    → 안정적이지만, 각 토큰을 효율적으로 사용 X convergence가 더 느릴 수 있음
- 예시: **DeepSeek-V3/R1**: bs = 3,072 → 15,360으로 **점진적으로 증가**시켜 469B tokens까지 학습

2-1. Overview

3. Batch size의 trade-off

- **작은** bs → 더 많은 optimizer step 필요 → **compute time** 증가
- **큰** bs → **OOM** 위험 증가 (GPU 메모리 부족)
- 실제: Final 성능은 **optimal batch size** 근처에서는 **크게 변하지 않음**

2-1. Overview

4. Tokens 기준의 batch size

- (LLM 학습에서는) 보통 “샘플 개수(bs)”가 아니라 **토큰 개수(bst)”**로 봄
- 계산: **bst = bs X seq** (seq = sequence length)
- 예시:
  - Recent LLM sweet spot: **4M-60M** tokens per batch
  - LLaMA-1: **4M** tokens/batch, total **1.4T** tokens
  - DeepSeek: **60M** tokens/batch, total **14T** tokens

2-1. Overview

5. 첫 번째 challenge: OOM

- bs를 크게 키우면 → GPU memory가 부족! (OOM)
- Multi-GPU 학습: 위 문제를 풀기 위해 **parallelism** 사용
  - **Data/tensor/pipeline/context** parallelism 등

- **Data** Parallelism
  - **ZeRO 1,2,3** (O-G-P)
- **Model** parallelism
  - **Tensor** Parallelism (column + row)
    - **Tensor** parallelism
    - **Sequence** parallelism
  - **Context** Parallelism
    - **Ring** attention & **Zig-Zag Ring** attention
  - **Pipeline** Parallelism
    - **Forward** only
    - All Forward All Backward (**AFAB**)
    - 1 Forward 1 Backward (**1F1B**)
    - **Interleaved** Pipeline Parallelism

2-1. Overview

Summary

- **기본 loop**: Forward → Backward → Optimizer step
- **Batch size**: 수렴 속도, 안정성, compute 효율성에 직결
- LLM 학습에서는 batch size를 **token 단위**로 보는 것이 일반적
  - Sweet spot은 최근 수백만~수천만 tokens 수준
- 큰 batch size를 쓰려면 필연적으로 **OOM** 이슈 발생

  → 이게 분산 학습의 이유!

2-2. Memory Usage in Transformers

2-2. Memory Usage in Transformers

A) Profiling the memory usage

- Transformer 학습할 때 GPU 메모리에 크게 **네 가지** 저장:
  - **(1) Parameters / Weights** (모델 가중치)
  - **(2) Gradients** (역전파에서 계산되는 gradient)
  - **(3) Optimizer states** (Adam 같은 Optimizer가 저장하는 momentum, variance 등)
  - **(4) Activations** (Forward pass 중간 결과: Backward pass의 gradient 계산에 필요)

2-2. Memory Usage in Transformers

A) Profiling the memory usage

- Note: 메모리 사용량을 정확히 계산하기는 쉽지 않음
  - 이유: **CUDA kernel overhead**: 보통 1~2GB 고정적으로 차지
    - Ex) torch.ones((1,1)).to("cuda") 실행 → nvidia-smi로 확인하면 기본 메모리 소모 보임
  - **버퍼 / 중간 결과 / 메모리 조각화(fragmentation)**: 소량이지만 상시 존재
    - 따라서 이런 부분은 대체로 작은 상수항이라 무시 가능

2-2. Memory Usage in Transformers

A) Profiling the memory usage

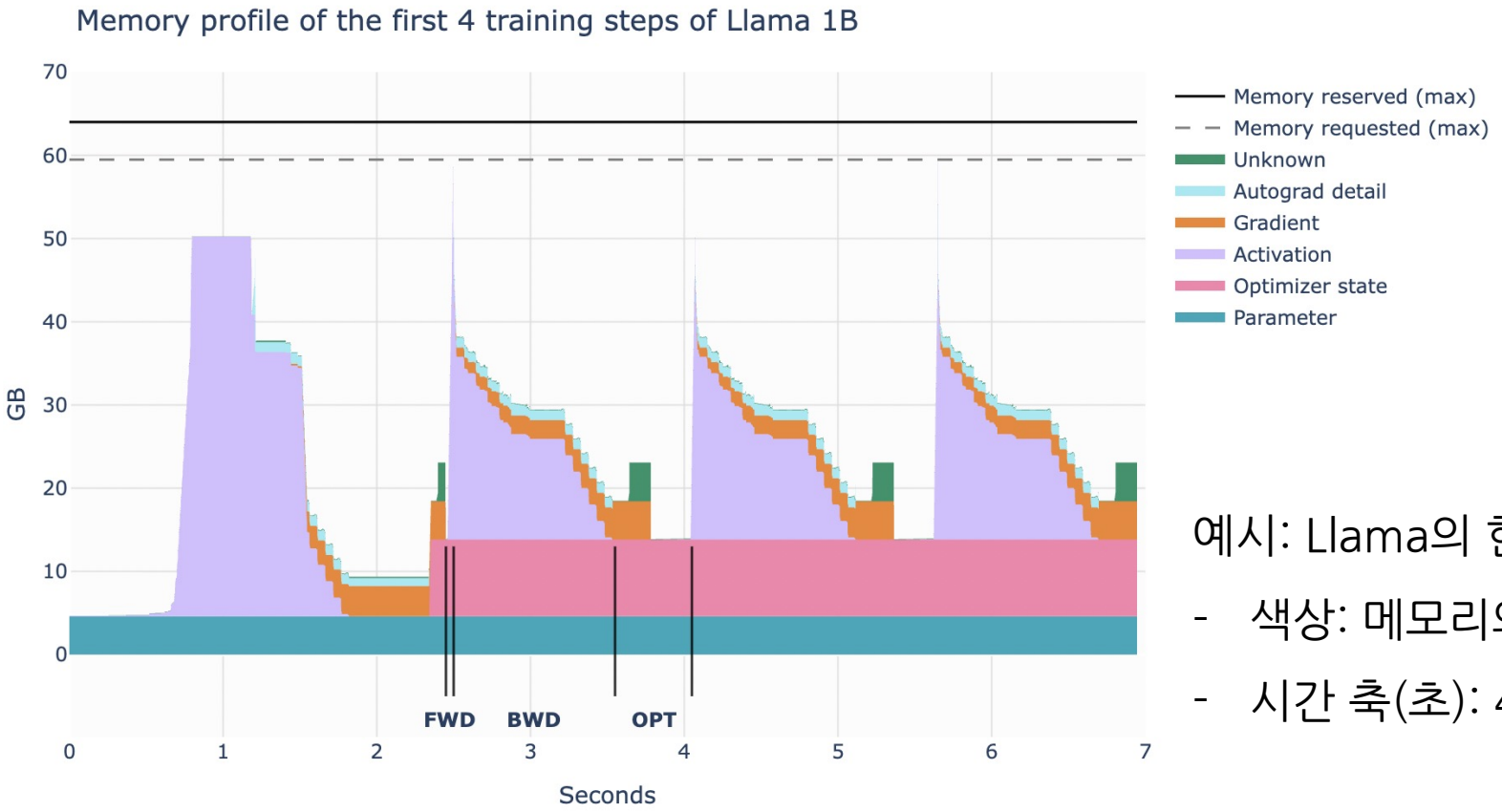
- Tensor의 **메모리 크기**를 결정하는 요소
  - **(1) Shape**: batch size, seq length, hidden dim, att heads, vocab size, sharding 여부 등
  - **(2) Precision**: FP32(4바이트), BF16/FP16(2바이트), FP8(1바이트)
    - 정밀도가 낮을수록 메모리 사용량이 줄어듦

  → Mixed precision training 섹션에서 자세히 다룸

- **Memory profiling** (메모리 사용 관찰)
  - PyTorch profiler를 이용하면 학습 중 메모리 분포를 볼 수 있음
  - 메모리 사용은 step마다 다르게 변함

2-2. Memory Usage in Transformers

A) Profiling the memory usage



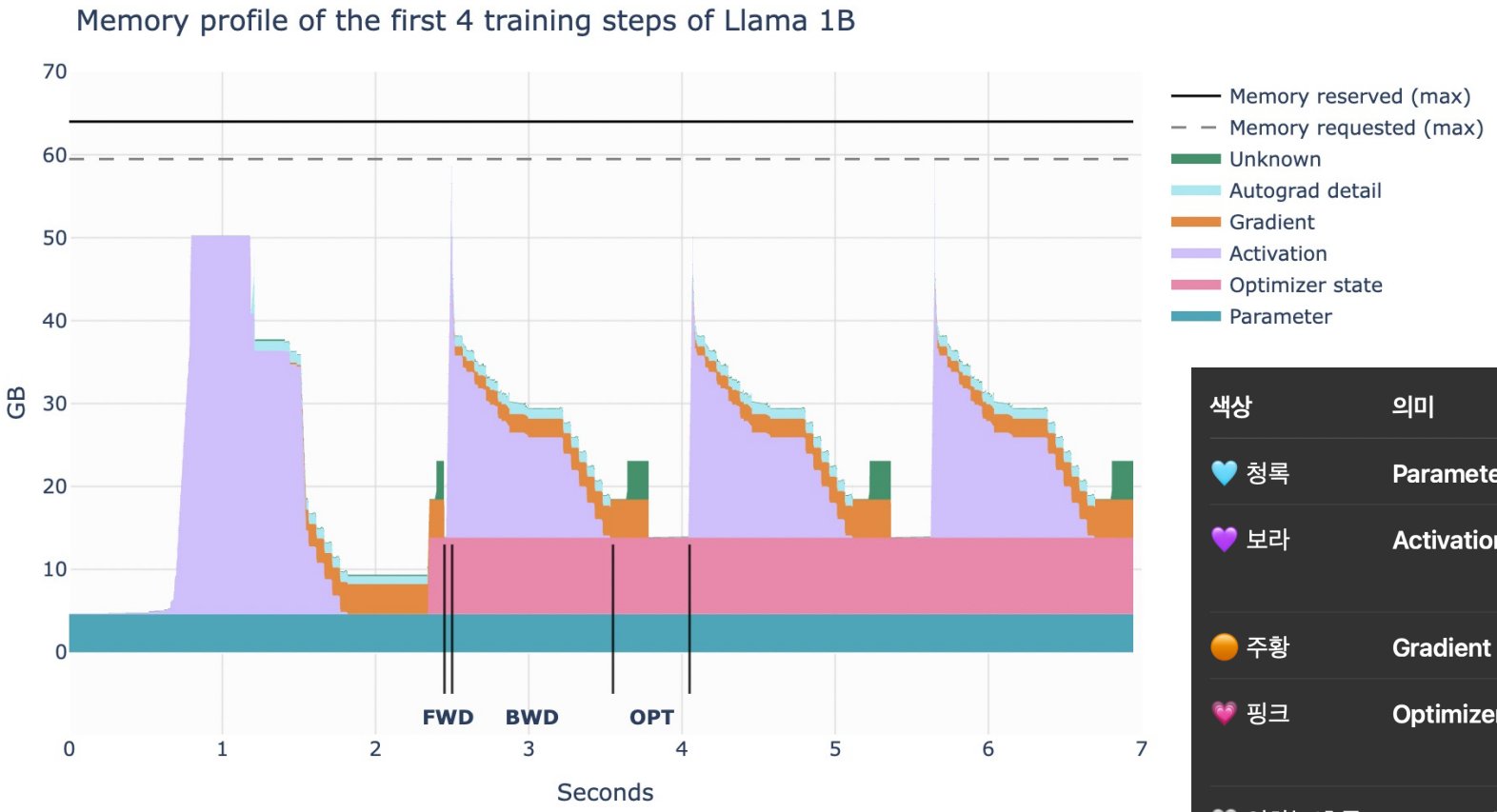
Memory profiling (메모리 사용 관찰)

예시: Llama의 한 training step 동안 **GPU 메모리 사용 패턴**

- 색상: 메모리의 용도
- 시간 축(초): 4 step

2-2. Memory Usage in Transformers

A) Profiling the memory usage



Memory profiling (메모리 사용 관찰)

색상	의미	설명
💙 청록	**Parameter**	모델 가중치 — 항상 기저에 존재 (변하지 않음)
💜 보라	**Activation**	Forward pass 중간 결과. Forward 시 급격히 증가하고 Backward 중 점차 해제됨
🟠 주황	**Gradient**	Backward 단계에서 생성. Backward가 끝나면 일시적으로 감소
💗 핑크	**Optimizer state**	Adam 등이 유지하는 momentum / variance. 첫 step 이후부터 상시 유지
🩶 연하늘/초록	**Autograd buffers 및 unknown**	PyTorch autograd 내부 버퍼 및 CUDA 캐시

2-2. Memory Usage in Transformers

A) Profiling the memory usage

- **Forward pass (FWD)**
  - 입력이 layer를 지나면서 중간 **activation**이 급격히 **늘어남** → 보라색 **급상승**
  - 이유: (향후 backward에서) Gradient 계산을 위해 이 activation들을 임시 저장
- **Backward pass (BWD)**
  - **Gradient**가 계산되며 주황색 부분이 **쌓임**
  - 동시에 사용된 **activation**들이 역전파가 진행되며 점차 해제 → 보라색 **감소**
- **Optimization step (OPT)**
  - **Optimizer state** (momentum, variance)가 **추가 저장** → 핑크색 새로 형성
  - 이 optimizer state는 다음 step 이후에도 남아있어 baseline **메모리를 영구적으로 높임**

2-2. Memory Usage in Transformers

B) Memory for Weight/Grads/Optim states

- LLM의 메모리 사용량을 구성하는 기본 세 요소:
  - **모델 가중치 (Parameters / Weights)**
  - **그래디언트 (Gradients)**
  - **옵티마이저 상태 (Optimizer states)**
- 이 세 가지는 전체 GPU 메모리의 대부분을 차지

2-2. Memory Usage in Transformers

B) Memory for Weight/Grads/Optim states

[1] 파라미터 수

- 간단한 **트랜스포머 모델**의 총 파라미터 수는 다음 식으로 근사

$$N = h \cdot v + L \cdot \left(\frac{1}{2}h^2 + \frac{1}{3}h\right) + 2h$$

- h: hidden dimension
- v: vocabulary size
- L: layer 수

h^2에 지배적!

→ 따라서 h이 커질수록 모델 크기가 제곱 비율로 증가

(즉, **모델을 2배** 키우면 파라미터는 **4배**, 메모리도 거의 **4배** 필요)

2-2. Memory Usage in Transformers

B) Memory for Weight/Grads/Optim states

[2] FP32 (Full Precision) 학습 시 메모리

- 각 항목의 byte 수:
  - Weights: **4 bytes**
  - Gradients: **4 bytes**
  - Optimizer states (Adam): momentum + variance = 4 + 4 = **8 bytes**

$$m_{params} = 4N, \quad m_{grad} = 4N, \quad m_{opt} = 8N$$

총합 = **16 bytes** × N (파라미터당).

2-2. Memory Usage in Transformers

B) Memory for Weight/Grads/Optim states

[3] BF16 (Mixed Precision) 학습 시 메모리

- 요즈음 일반적으로 **Mixed Precision** 사용 (weight & gradient에 한해서는 2 bytes)
- 각 항목의 byte 수:
  - Weights: **2 bytes**
  - Gradients: **2 bytes**
  - Optimizer states (Adam): momentum + variance = 4 + 4 = **8 bytes**
- 일부 라이브러리(Nanotron 등)는 **gradient를 추가적으로 FP32**로 저장해 안정성을 높임 → **+4 bytes**
- FP32로 저장된 weights를 “**master weights**”라고 부름

2-2. Memory Usage in Transformers

B) Memory for Weight/Grads/Optim states

[3] BF16 (Mixed Precision) 학습 시 메모리

- Q1) Mixed Precision은 메모리를 절약? No!
  - BF16를 사용할 때, 계산은 반정밀도로 하더라도
  - (정밀한 update를 위해) **FP32 버퍼를 추가로** 들고 있어야!

  → 이로 인해 **+ 4N bytes**

2-2. Memory Usage in Transformers

B) Memory for Weight/Grads/Optim states

[3] BF16 (Mixed Precision) 학습 시 메모리

- Q2) 그러면 애초에 4 bytes 짜리로 저장하면 되는거 아닌가? No!
  - 핵심은 **"계산"**과 **"저장"**이 완전히 다른 리소스라는 점!

구분	계산(Compute)	저장(Storage)
의미	연산이 실제로 일어나는 과정 (행렬 곱 등)	GPU 메모리에 값을 보관하는 공간
대표 예	forward/backward 시 multiply, add, matmul 등	weights, gradients, optimizer states
단위	Tensor Core 연산 단위	GPU VRAM 용량

GPU에서 **가장 큰 병목은 계산 속도(throughput)** 이지, 저장공간이 아님

→ 그래서 "계산은 가볍게(BF16)" 하면서도, 정확한 값은 "저장 시 FP32"로 보존하는 전략!

2-2. Memory Usage in Transformers

B) Memory for Weight/Grads/Optim states

[3] BF16 (Mixed Precision) 학습 시 메모리

- Q2) 그러면 애초에 4 bytes 짜리로 저장하면 되는거 아닌가? No!
  - a) **계산** (Forward/Backward): **BF16** 사용
    - BF16은 2byte이므로 메모리 대역폭이 절반 → **연산 속도가 2배 이상 빨라짐**
    - Tensor Core는 FP16/BF16에서 최고 효율 발휘
    - 즉, 연산 속도 향상 + activation 메모리 절감
  - b) **업데이트** (Update/Optimization): **FP32** 사용
    - BF16은 정밀도가 낮아서 아주 작은 gradient를 0으로 날려버림 (underflow)
    - 때문에, **weight 업데이트는 FP32 master copy**로 수행해 **정확도 유지**

2-2. Memory Usage in Transformers

B) Memory for Weight/Grads/Optim states

[3] BF16 (Mixed Precision) 학습 시 메모리

Precision	연산 속도	GPU 효율	Activation 메모리	안정성
FP32	느림	낮음	큼	안정적
BF16	빠름	높음	작음	불안정(정밀도 손실)
BF16 + FP32 copy	✅빠르고 ✅안정	✅높은 효율	✅안정 업데이트	✅균형형

- **FP32만 쓰면:** 느려서 대형 모델 학습 불가능.
- **BF16만 쓰면:** 빠르지만 수치 불안정, 학습 실패 위험.
- **BF16 + FP32 copy:** 둘의 장점을 모두 취함.

2-2. Memory Usage in Transformers

B) Memory for Weight/Grads/Optim states

[3] BF16 (Mixed Precision) 학습 시 메모리

- Q3) Gradient accumulation 시, 오히려 BF16 > FP32일 수도?
  - Yes. FP32 == BF16인데, **Grad accum.**까지 할 경우에는 **오히려 BF16 > FP32**이다!

Model parameters	FP32 or BF16 w/o FP32 grad acc	BF16 w/ FP32 grad acc
1B	16 GB	20 GB
7B	112 GB	140 GB
70B	1120 GB	1400 GB
405B	6480 GB	8100 GB

2-2. Memory Usage in Transformers

B) Memory for Weight/Grads/Optim states

[3] BF16 (Mixed Precision) 학습 시 메모리

- **Q4) Gradient accum은 FP32에서도 할 수 있는거 아닌가?**
  - Yes! Grad accum & Precision 자체는 **서로 무관**
  - 다만, **FP/BF16환경에서는 필수에 가까워짐**
    - FP32: Gradient를 FP32로 저장
    - BF16: Gradient를 FP16로 저장

  → 이런 상황에서 누적을 BF16으로 하면 **작은 gradient들이 계속 사라지고 학습 불안정**

2-2. Memory Usage in Transformers

C) Memory for Activations

Activations란?

- Forward pass 중 각 layer에서 계산된 **중간 출력값**
- **저장의 필요성**: Backward pass 때 **gradient 계산에 다시 써야하므로**, Forward 중에 전부 저장해 둠

Questions

- Q1) Activation이 **얼마나 많은 메모리**를 차지하는지?
- Q2) Activation **explosion**이 왜 생기는지?

2-2. Memory Usage in Transformers

C) Memory for Activations

Activations에 필요한 메모리

$$m_{\text{act}} = L \cdot \text{seq} \cdot \text{bs} \cdot h \cdot \left(\frac{3}{4} + \frac{5 \cdot n_{\text{heads}} \cdot \text{seq}}{h} \right)$$

항목	의미
L	레이어 개수
seq	sequence length (토큰 길이)
bs	batch size (샘플 수)
h	hidden dimension
n_{heads}	attention heads 수

항목	관계
batch size (bs)	activation 메모리에 **선형(linear)** 비례
sequence length (seq)	activation 메모리에 **제곱(quadratic)** 비례
hidden dim (h)	activation 메모리에 선형 비례
레이어 수 (L)	선형 비례

- batch size를 2배로 하면 메모리도 2배
- seq length를 2배로 하면 메모리는 4배

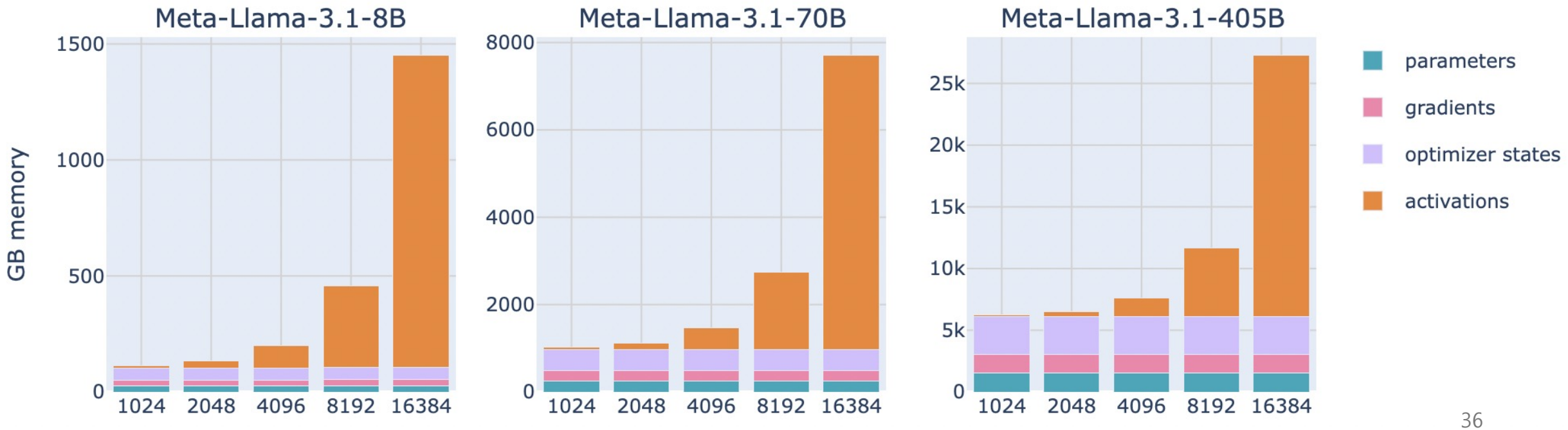
→ 따라서, seq length를 키우면 activation이 지배적인 메모리가 됨!

2-2. Memory Usage in Transformers

C) Memory for Activations

- [x축] Sequence length (토큰 길이)
- [y축] GPU 메모리 (GB)

seq length를 키우면 activation이 지배적인 메모리가 됨!



2-2. Memory Usage in Transformers

C) Memory for Activations

[그림] 요약

- (1) Param/Grad/Optim → **모델 크기**에 dependent
- (2) Activations → **seq len**와 **bs**에 dependent
  - 특히 seq len이 8k~16k 이상이면 activation이 전체 메모리의 대부분을 차지

생기는 문제점: **"Activation explosion"**

- Llama-70/405B 같은 모델: 긴 sequence 학습 시 activation만으로도 수천~수만 GB 메모리
- GPU 메모리 한계(예: H100 = 80GB)에 바로 부딪힘

→ 이 "activation explosion"이 **LLM 학습에서 가장 큰 메모리 병목**

2-3. Activation Recomputation

2-3. Activation Recomputation

A) Activation Recomputation이란

핵심: “메모리를 아끼기 위해 일부 **activations을 저장하지 않고**, 나중에 필요할 때 **다시 계산**한다”

아래 셋은 모두 같은 개념

- **Activation recomputation**
- **Gradient Checkpointing**
- **Rematerialization**

학습 중 Forward → Backward 순으로 진행할 때, 원래는

- Forward: 각 layer의 **activation을 저장**
- Backward: 이 activation을 이용해 **gradient를 계산**

2-3. Activation Recomputation

A) Activation Recomputation이란

- 문제점: 모든 activation을 저장하면 GPU 메모리가 터짐!
  - 특히 sequence 길이가 길수록 더 위험!
- 해결책: 일부 activation은 저장하지 않고, **Backward 때 필요하면 다시 계산(recompute)** 하자!

→ 즉, **메모리를 절약**하는 대신 **계산량을 늘리는** 전략

2-3. Activation Recomputation

B) Checkpoint 전략

1. Full recomputation

- 각 Transformer layer 사이를 checkpoint로 지정
  - [장] 메모리 절약 극대화 (최대)
  - [단] 하지만 계산량 30~40% 증가 → 학습 시간도 길어짐

2. Selective recomputation

- 모든 layer를 다시 계산할 필요는 없음
  - Attention은 FLOPs 대비 메모리 점유가 크지 않음 → 저장할 필요 X
  - FFN은 activation 크기가 커서 checkpointing 효율 ↑

2-3. Activation Recomputation

B) Checkpoint 전략

1. Full recomputation 예시

- GPT-3 (175B):
  - GPU 메모리 사용량: 1/3으로 줄어듬 + 학습 속도는 약 1.3-1.4배 느려짐

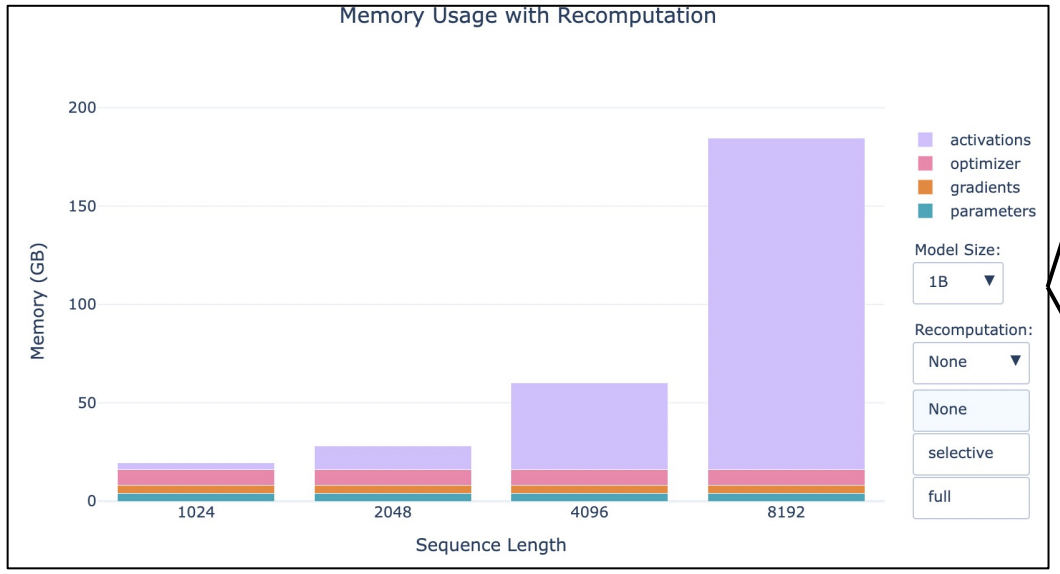
2. Selective recomputation 예시

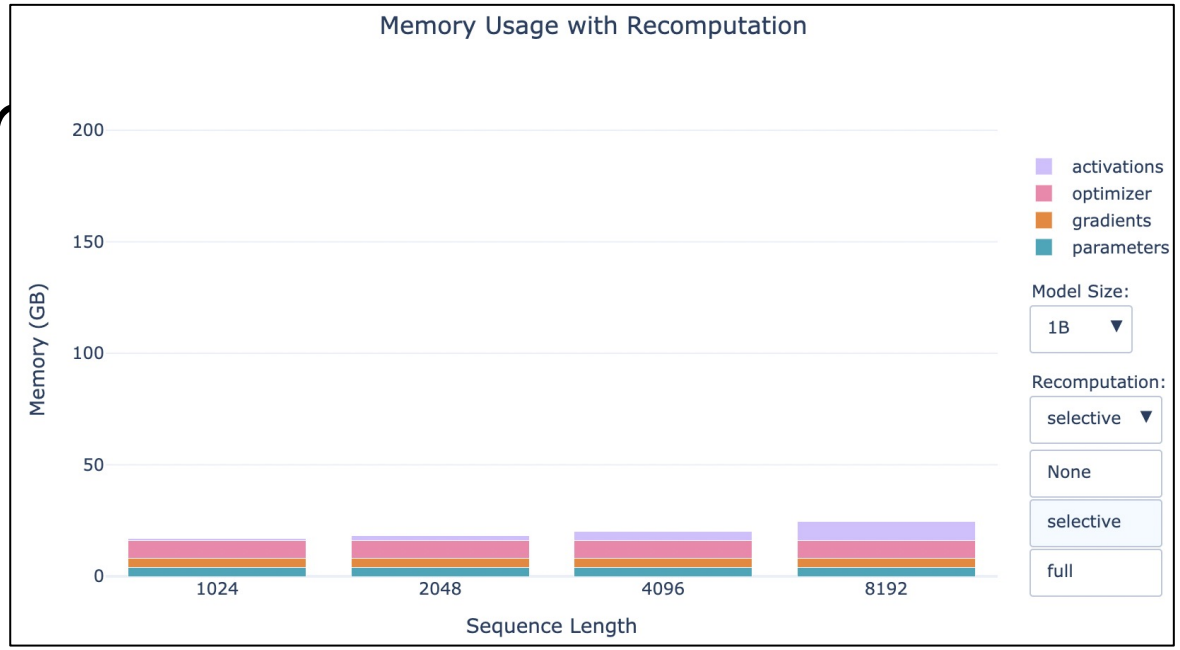
- GPT-3 (175B):
  - GPU 메모리 사용량: 70% 감소 + 계산량은 2.7% 증가
- DeepSeek-V3:

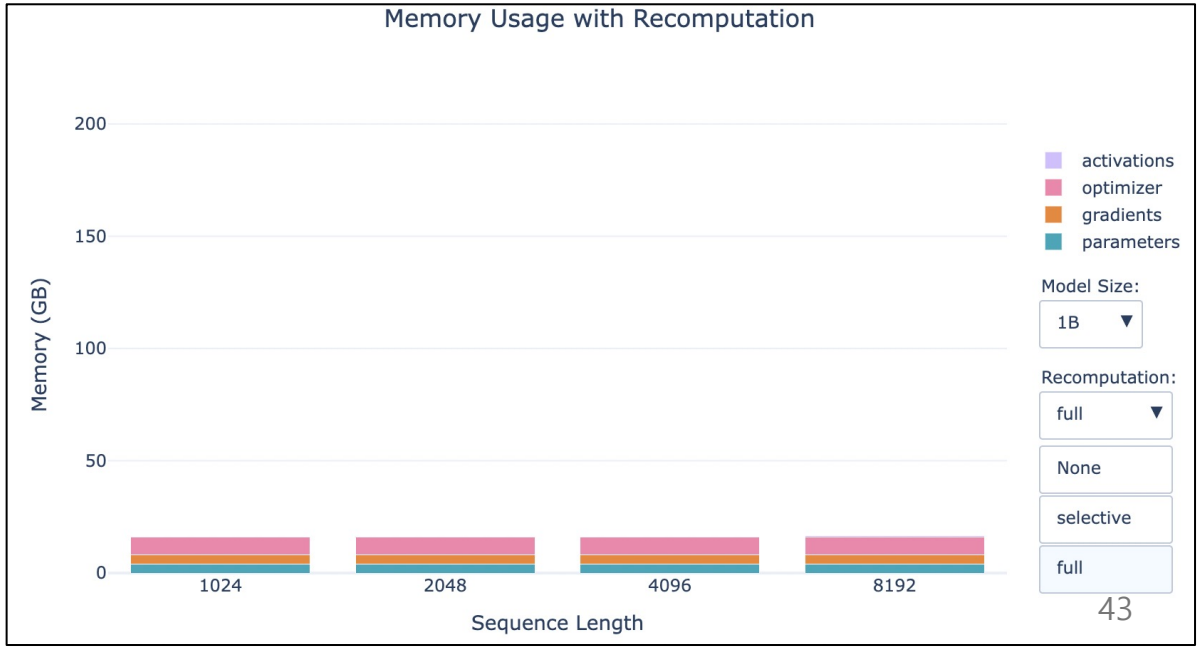
  “Multi-Head Latent Attention (MLA)” 구조로 attention activation 저장 크기를 더 줄임

2-3. Activation Recom

B) Checkpoint 전략







+ 특히 **작은 모델**에서는 seq len가 클수록, activation 비중이 커지므로, **recomputation 효과가 큼!**

2-3. Activation Recomputation

C) FLOPs & 효율성 (HFU vs. MFU)

[1] Floating Points Operations (FLOPs)

- 부동소수점 연산 횟수 (“얼마나 많은 계산을 했는가”)
  - 덧셈 1회 → 1 FLOP
- **Activation recomputation이 FLOPs를 늘린다!**
  - Recomputation X: 원래 한 번의 forward + backward면 충분
  - Recomputation O: Backward 중에 일부 forward 계산을 다시 함
- Example) forward 연산 = 100 FLOPs, backward 연산 = 200 FLOPs
  - Recomputation 시 backward에서 forward 일부를 다시 계산
    - 예: 50 FLOPs 추가 → 총 350 FLOPs 수행!

2-3. Activation Recomputation

C) FLOPs & 효율성 (HFU vs. MFU)

[2] Hardward FLOPs Utilization (HFU)

- 요약: **GPU가 얼마나 바쁘게 일했는가**를 보는 관점
- 공식: (**Total** FLOPs) / (GPU FLOPs)
- Recomputation을 쓰면 실제 수행한 FLOPs가 늘어나니까 **HFU는 상승**

[3] Model FLOPs Utilization (MFU)

- 요약: 모델 입장에서 **필요한** 연산량 대비 효율을 보는 관점
- 공식: (**Forward + Backward** FLOPs) / (GPU FLOPs)
- Recomputation은 “필요 없는 추가 계산”이므로, **MFU는 더 낮음**

2-3. Activation Recomputation

C) FLOPs & 효율성 (HFU vs. MFU)

예시)

- GPU 최대 성능: 1초에 1000 FLOPs 가능.
- 모델 1 step의 이론적 연산량: 500 FLOPs

구분	실제 수행한 FLOPs	계산시간	HFU	MFU
기본 학습	500	1초	500/1000 = 50%	500/1000 = 50%
Recomputation 적용	700 (500 + 재계산 200)	1.1초	700/1000 = 70% (GPU 더 바쁨)	500/1000 = 50% (모델은 동일 연산)

2-3. Activation Recomputation

C) FLOPs & 효율성 (HFU vs. MFU)

Summary

- Recomputation은 GPU를 더 바쁘게 만들어서
  - **HFU**는 올라가지만
  - 중복 계산이 많아져서 **MFU**는 낮아짐
- GPU 벤치마크할 때 HFU만 보고 판단하면 안됨!
  - Recomputation을 쓰면 GPU는 더 바쁘지만, 실제로는 중복 계산이 많음
- 따라서, 모델 학습 효율을 평가할 때는 **MFU (필요 연산 대비 속도)를 봐야 더 공정**

2-3. Activation Recomputation

D) Flash Attention과의 관계

- 대부분의 LLM 훈련은 **FlashAttention**을 사용
- FlashAttention은 자체적으로 **backward 시 attention score와 matrix를 recompute**
- FlashAttention = **selective recomputation**을 내장한 형태

2-3. Activation Recomputation

E) Trade-off 관계

- Recomputation은 …
  - (단) **계산량**은 조금 늘리지만
  - (장) **GPU 메모리** 접근을 줄여서

  → 전체 학습 속도는 **오히려 더 빨라질 수 있음**
- 이유:
  - GPU에서 **"계산"**은 빠르고, **"메모리 접근"**은 느리기 때문.
  - Compute > Memory access 구조에서 **recomputation이 오히려 효율적**

2-3. Activation Recomputation

E) Trade-off 관계

구분	저장 메모리	계산량	특징
No recomputation	매우 큼	적음	빠르지만 GPU 메모리 폭발
Full recomputation	매우 작음	+30~40%	메모리 절약 극대화, 속도 저하 큼
Selective recomputation	중간	+2~5%	**최적 trade-off**, 대부분 LLM이 사용

2-4. Gradient Accumulation

2-4. Gradient Accumulation

핵심 아이디어

- “큰 batch를 한 번에 GPU에 못 올리니까, **여러 개의 작은 batch (micro-batch)로 나눠서** 차례대로 처리한 뒤, 각 **batch의 gradient를 더해서** 한 번에 optimizer update를 하는 방법”
- 가장 단순하면서도 효과적인 방법

$$bs = gbs = mbs \times grad_{acc}$$

용어	의미
micro-batch size	한 번의 forward/backward에서 처리하는 샘플 수
gradient accumulation steps	몇 번의 micro-batch를 누적할지
global batch size	optimizer step 한 번에 실제로 반영되는 전체 샘플 수

2-4. Gradient Accumulation

Memory 절약 효과

- Activation 메모리는 batch 크기에 비례 (linear)
- 이때, gradient accumulation은
  - “한 번에 **하나**의 micro-batch만 메모리에 올리고 처리”
  - 한 번에 저장할 activation은 작고, **이전 batch의 것은 바로 비움**

2-4. Gradient Accumulation

단점: Slow

- 한 Optimizer step에 (Forward + Backward)를 **여러 번 수행**해야 함
  - 즉, 연산 횟수는 그대로인데 순차 실행 → 시간이 더 걸림
- GPU utilization이 낮아질 수도 있음

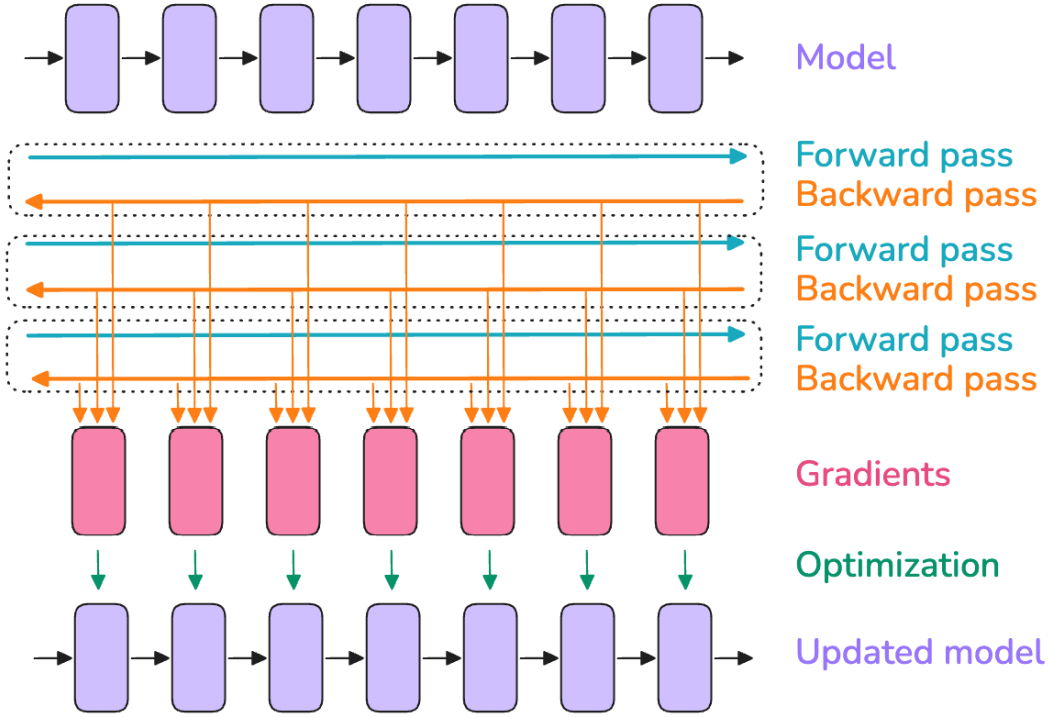
Data Parallelism + Gradient Accumulation

- 사실, micro-batch “간” (Forward + Backward)는 서로 독립적
  - 따라서 이걸 여러 GPU에서 병렬 처리할 수 있음 → **Data Parallelism**

2-4. Gradient Accumulation

Summary

항목	설명
목적	큰 batch를 나눠서 메모리 폭발(OOM) 방지
공식	$gbs = mbs \times grad_{acc}$
장점	Activation 메모리 ↓ (micro-batch만 유지)
단점	연산량 동일 → 속도 ↓
확장	여러 GPU로 병렬화 → Data Parallelism



Using gradient accumulation means **we need to keep buffers where we accumulate gradients that persist** throughout a training step, whereas without gradient accumulation, in the backward pass gradients are computed while freeing the activation memory, which means lower peak memory use.

3. Data Parallelism

3-1. Overview

3-1. Overview

A) Data Parallelism의 개념

- 개념: 한 모델을 **여러 GPU에 복제**해서, **서로 다른 micro-batch**를 동시에 처리
- 예시:
  - GPU1: Batch A
  - GPU2: Batch B
- 수행 방법:
  - Step 1) 각 GPU가 **독립적**으로 forward → backward 수행한 뒤
  - Step 2) Gradient를 평균 내어(**all-reduce**) 모델을 동기화

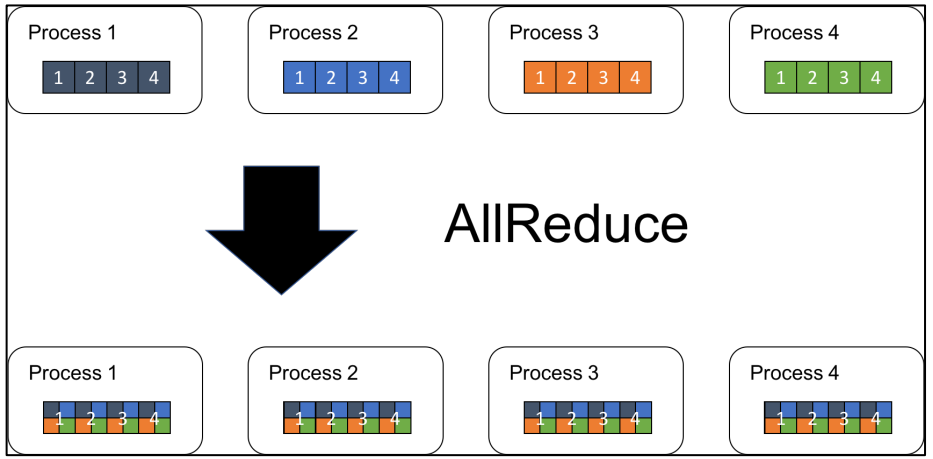
3-1. Overview

B) All-reduce

- 유의점: GPU마다 다른 mini-batch로 학습 → 당연히 Gradient가 다르게 계산됨
- 해결책: **All-Reduce** 연산을 사용해 **모든 GPU의 gradient를 평균**
  - 각 GPU가 계산한 gradient를 전부 합침 & 나누어 평균
  - 이에 따라, 모든 GPU가 같은 평균 gradient를 **공유** → 모델 파라미터 동기화

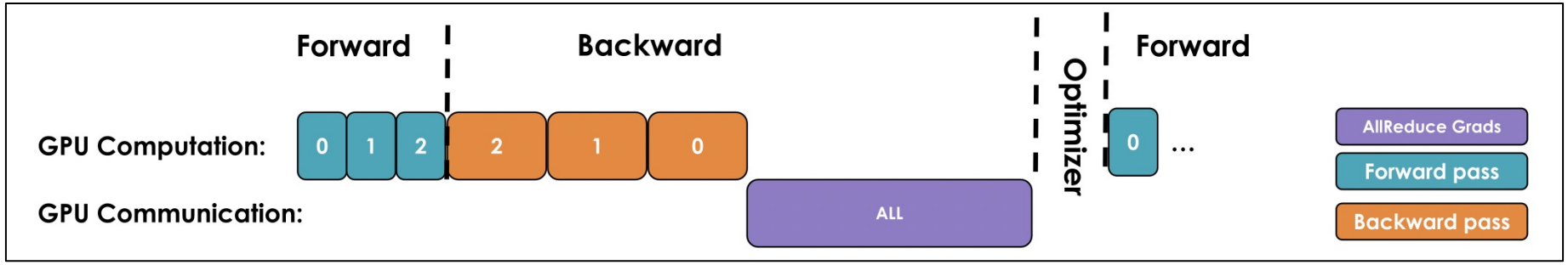
3-1. Overview

B) All-reduce



모든 노드(GPU)의 값을 더한 뒤,

그 결과를 모든 노드에 **다시 나눠주는 통신 방식**



여기서 0,1,2는 layer의 index

https://www.google.com/url?sa=i&url=https%3A%2F%2Fbrunch.co.kr%2F%40chris-song%2F96&psig=AOvVaw2vKwpdvLBlRL-RhjrOF_qh&ust=1759666950917000&source=images&cd=vfe&opi=89978449&ved=0CBUQjRxqFwoTCLjA763EipADFQAAAAAdAAAAABAE

3-1. Overview

C) Naïve DP의 한계

- 가장 단순한 구현
  - Step 1) 모든 GPU가 forward + backward를 완료
  - Step 2) Gradient를 all-reduce
  - Step 3) Optimizer step
- 위의 문제점: Backward 끝나고 **통신(all-reduce)을 기다리는 동안 GPU들이 idle**

  → 즉, 계산과 통신 이 **"직렬적(sequential)"**으로 수행되는 게 비효율적

3-1. Overview

D) Naïve DP의 해결책

통신과 계산을 겹치기 (Overlap)

- Backward 중간중간에 partial gradient가 생기면 → **그때마다 바로 all-reduce를 시작**
- 즉, gradient 계산(**Compute**) & gradient 동기화(**Communication**)를 동시에 진행

용어	의미
Model replica / instance	GPU마다 복사된 동일한 모델
Micro-batch	각 GPU가 처리하는 데이터 단위
All-Reduce	모든 GPU의 gradient를 평균 내어 동기화하는 통신 연산
Overlap	통신과 계산을 동시에 진행하는 최적화 기법

3-2. DP vs. DDP

3-2. DP vs. DDP

A) DP란? DDP란?

- 공통점: “모델의 파라미터를 여러 GPU에서 동시에 학습”하기 위한 parallelism
- 차이점: **(1) DP (Data Parallel)**
  - **한 프로세스(process)** 안에서 여러 GPU를 사용 & 내부적으로는 싱글 스레드 + 여러 장비 구조
  - 하나의 **메인 프로세스(GPU 0)** 가 있음
  - Forward/Backward 후, 각 GPU의 gradient를 **GPU 0으로 모아서 평균**
  - **GPU 0이 weight를 업데이트**한 뒤, 다시 **다른 GPU에 broadcast**
  - 모든 gradient가 GPU 0으로 모이므로, **GPU 0이 I/O 및 연산 부담이 큼**
    - 멀티 노드 불가 (한 머신 내 여러 GPU만 사용 가능)
    - 비효율적 메모리 복제 (모델이 각 GPU마다 새로 복사됨)

3-2. DP vs. DDP

A) DP란? DDP란?

- 공통점: “모델의 파라미터를 여러 GPU에서 동시에 학습”하기 위한 parallelism
- 차이점: **(2) DDP (Distributed Data Parallel)**
  - **GPU마다 독립된 프로세스** (=각자 Python 프로세스, 독립된 모델 복제).
  - 각 프로세스는 자기 GPU의 데이터를 가지고 Forward/Backward.
  - Backward 중 gradient를 NCCL 통신(collective all-reduce)으로 자동 평균화
  - 모든 GPU가 동시에 동일한 업데이트를 적용
    - GPU 0 병목 없음 — 통신을 all-reduce로 분산 처리
    - 멀티 노드 지원 (클러스터 간 통신도 가능)
    - 학습 속도와 효율이 우수, 메모리 사용도 DP보다 적음

3-2. DP vs. DDP

B) Summary

구분	DP (DataParallel)	DDP (DistributedDataParallel)
프로세스 구조	1개 프로세스, 여러 GPU 스레드	GPU당 1개 프로세스
통신 방식	Gradient를 GPU0으로 모아서 update	All-reduce로 모든 GPU가 동등하게 통신
확장성	단일 머신 한정	멀티 노드(클러스터) 지원
병목	GPU0에 집중됨	없음 (분산 통신)
권장 여부	구버전/테스트용	PyTorch 공식 권장 방식
코드 예시	nn.DataParallel(model)	nn.parallel.DistributedDataParallel(model)

3-3. Three Optimization Steps

3-3. Three Optimization Steps

Data Parallelism의 3가지 최적화

- “계산(Backward)과 통신(All-Reduce) 을 **겹치고**, 작은 통신을 **묶고**, 불필요한 통신은 **생략**”
- 구체적인 세 단계:
  - A) First Optimization: **Overlap** Gradient Synchronization with Backward Pass
  - B) Second Optimization: **Bucketing** gradients
  - C) Third Optimization: **Interplay** with gradient accumulation

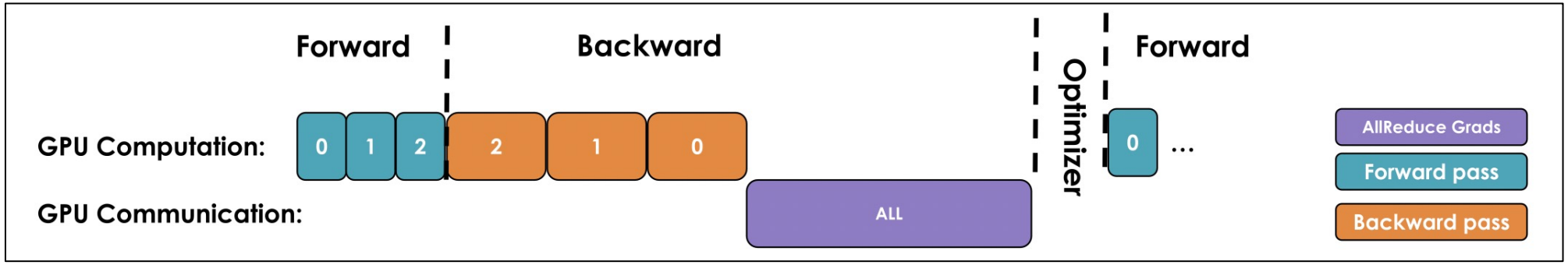
3-3. Three Optimization Steps

A) First Optimization: Overlap Gradient Synchronization with Backward Pass

- Naïve DP의 문제점:
  - **Backward가 전부 끝난 뒤에야** all-reduce를 실행 → GPU가 통신 중에는 **idle 상태**
- 해결책:
  - Layer별로 gradient가 계산되는 **즉시** 바로 all-reduce를 시작.
  - 즉, backward(계산)과 all-reduce(통신)를 **동시**에 수행.
- 효과: 전체 학습 **속도 향상**
  - 이유: (Backward와 통신이 겹치므로) GPU가 쉬는 시간 거의 없음 → **통신 대기 시간 ↓**

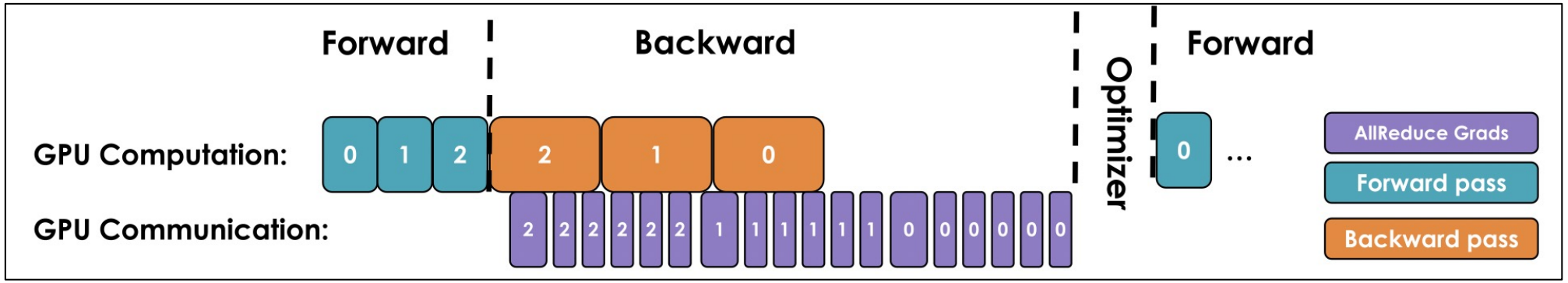
3-3. Three Optimization Steps

A) First Optimization: Overlap Gradient Synchronization with Backward Pass



Backward가 오른쪽(마지막 layer)부터 시작되자마자, 그 layer의 gradient를 즉시 all-reduce 시작

→ **GPU는 계속 계산을 하면서 동시에 통신도 수행!**



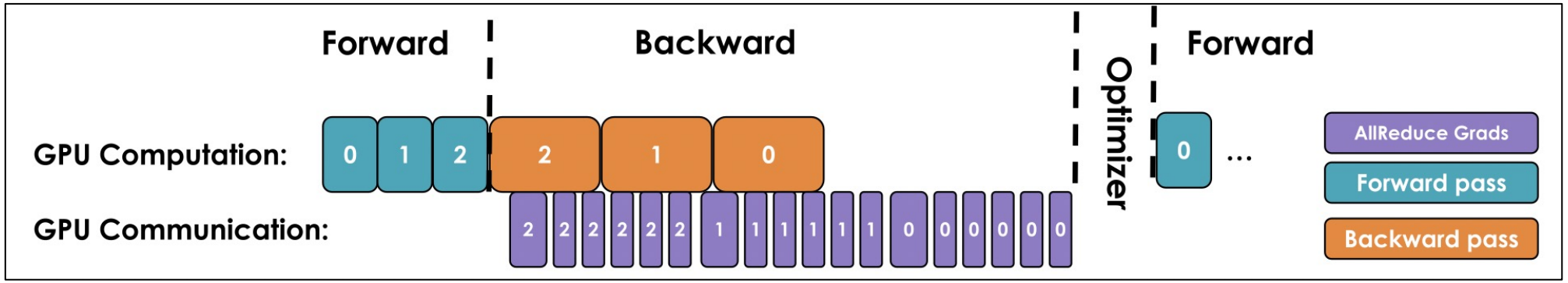
3-3. Three Optimization Steps

B) Second Optimization: Bucketing gradients

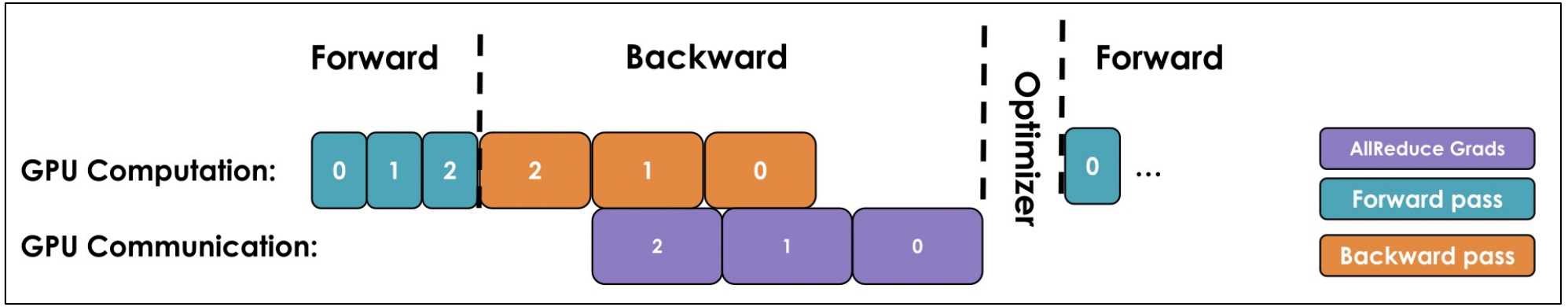
- Naïve DP의 문제점:
  - 파라미터가 수억 개라면, gradient tensor도 수십만 개
  - **작은** 텐서 단위로 all-reduce를 반복하면 → **통신 오버헤드 폭발**
- 해결책:
  - 여러 gradient를 묶어서 하나의 **큰 버킷(bucket)** 으로 만들고, **버킷 단위로 한 번의 all-reduce 수행**
  - "작은 택배 여러 번 보내지 말고 큰 상자 몇 개로 묶어서 보내자."
- 효과: **통신 효율 증가**
  - 작은 텐서 단위의 **통신 호출 횟수 ↓**
  - GPU 간 **네트워크 트래픽 감소**

3-3. Three Optimization Steps

B) Second Optimization: Bucketing gradients



Bucket으로 묶어서 처리



3-3. Three Optimization Steps

C) Third Optimization: Interplay with gradient accumulation

- Naïve DP의 문제점: (**Gradient accum** 시…)
  - 각 micro-batch마다 backward가 **여러 번** 일어남
  - Naive 구현은 매번 all-reduce를 호출 → **불필요한 중복 통신** 발생
- 해결책:
  - micro-batch 사이에서는 gradient sync를 **잠시 꺼두기**
  - **마지막 backward (optimizer step 직전)에만** 한 번 all-reduce 수행.
- 효과:
  - 불필요한 all-reduce 제거 → **통신 부하 ↓**
  - accumulation의 본래 의도(큰 batch 효과)는 유지

3-3. Three Optimization Steps

A--C) Summary

최적화 단계	핵심 아이디어	효과
1. Overlap	backward와 all-reduce를 동시에 수행	통신 대기 시간 제거
2. Bucketing	작은 gradient 묶어 한 번에 전송	통신 호출 횟수 감소
3. no_sync()	accumulation 중간엔 통신 생략	불필요한 통신 제거

Data Parallelism의 효율을 극대화하려면,

- (1) backward와 통신을 **겹치고**
- (2) 작은 gradient를 **묶어서** 전송하고
- (3) accumulation 중간엔 통신을 **생략 (no_sync)**

3-4. Revisiting Global Batch Size

3-4. Revisiting Global Batch Size

A) Global Batch Size

- **Data Parallelism** + **Gradient Accumulation**을 합친 "최종적인 batch size 확장 공식"
  - **병렬적**으로(batch-wise) 처리할 수 있는 GPU 수와,
  - **순차적**으로(accumulate) 처리할 수 있는 micro-batch 수를 곱하기

$$bs = gbs = mbs \times grad\_acc \times dp$$

기호	의미
mbs	micro-batch size (GPU 하나가 한 번에 처리하는 샘플 수)
$grad_acc$	gradient accumulation steps (순차적으로 누적하는 횟수)
dp	data parallelism (동시에 병렬로 돌리는 GPU replica 개수)
gbs	global batch size (전체 batch 크기)

3-4. Revisiting Global Batch Size

B) Data Parallelism & Gradient Accumulation의 관계

- **Gradient Accumulation**
  - **순차적** (sequential) 기법
  - GPU 메모리가 부족할 때, 작은 batch를 여러 번 반복해서 누적

  → 속도는 **느려짐** (순차적으로 돌아가기 때문)

- **Data Parallelism**
  - **병렬적** (parallel) 기법
  - 여러 GPU가 동시에 각기 다른 데이터 샘플을 학습.

  → 속도는 **빨라짐** (병렬 수행).

3-4. Revisiting Global Batch Size

B) Data Parallelism & Gradient Accumulation의 관계

- 둘 다 batch size를 늘릴 수 있는 방법이지만,
  - **Data Parallelism**은 "**속도 향상**(병렬화)"에 유리
  - **Gradient Accumulation**은 "**메모리 절약**"에 유리

항목	Data Parallelism (dp)	Gradient Accumulation (grad_acc)
처리 방식	병렬 (동시에 여러 GPU)	순차 (한 GPU에서 여러 번)
목적	속도 향상	메모리 절약
확장 한계	GPU 개수 한계	시간 한계
GPU 메모리 영향	그대로 유지	적게 사용
Compute 효율	높음	낮음

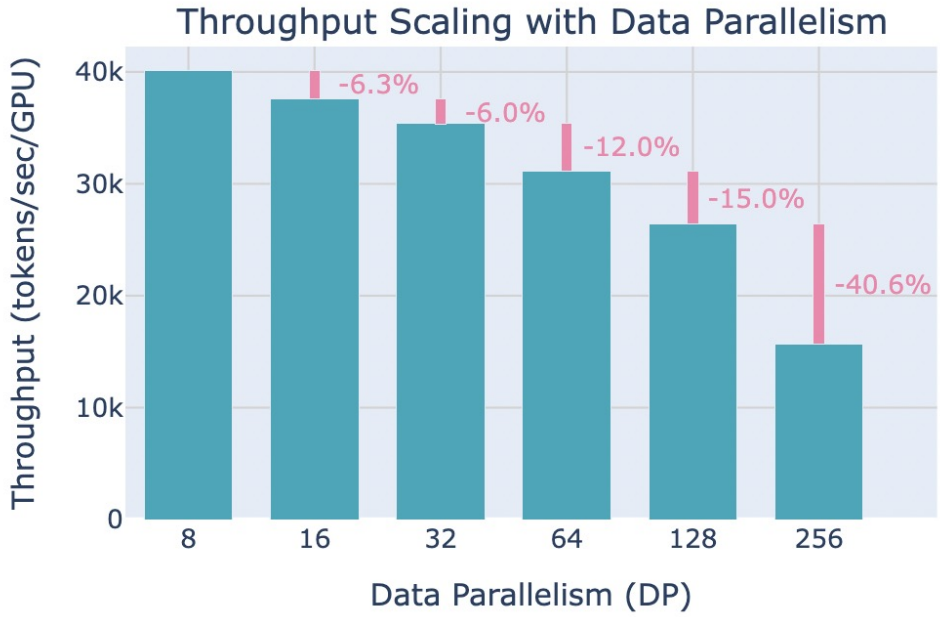
- 현실에서의 사용
  - Step 1) 먼저 **가능한 한 많은 GPU(dp)** 를 사용해 병렬 처리 확장
  - Step 2) 그래도 global batch가 부족하거나 GPU 메모리가 부족하면 **gradient accumulation**을 추가

3-5. Our Journey Up to Now

3-5. Our Journey Up to Now

A) Throughput Scaling with Data Parallelism

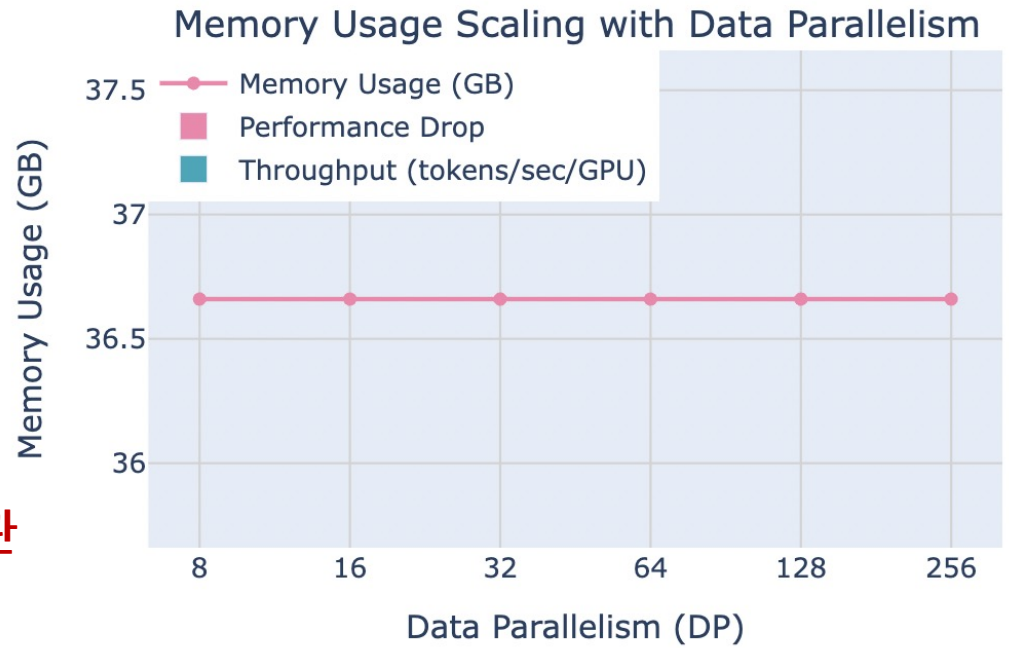
- x,y축
  - [x축] Data Parallelism (GPU 개수, DP rank)
  - [y축] Throughput (GPU당 초당 처리 토큰 수)
- 결과: **GPU로 늘릴수록 ..**
  - Total throughput은 증가하지만
  - **GPU당 효율(Throughput per GPU) 은 점점 떨어짐**
- 이유: GPU 늘어날수록 **통신 오버헤드(All-Reduce)**가 급증
  - 특히 100개 이상 넘어가면 ring latency (통신이 한 바퀴 도는 데 걸리는 시간)가 병목이 됨
  - 즉, 계산과 통신의 완벽한 overlap이 점점 어려워져서 효율이 급락



3-5. Our Journey Up to Now

B) Memory Usage Scaling with Data Parallelism

- x,y축
  - [x축] Data Parallelism (GPU 개수, DP rank)
  - [y축] GPU Memory
- 결과: **GPU당 메모리 사용량은 거의 일정** (~36.8GB)
- 이유:
  - DP는 각 GPU에 완전한 모델 복사본을 올림
    - GPU당 메모리는 모델 크기에 의해 결정, **GPU 개수와 무관**
  - 대신 데이터가 나뉘어 들어가므로 배치 전체 크기는 증가
    - 그러나 GPU당 메모리엔 변화 없음

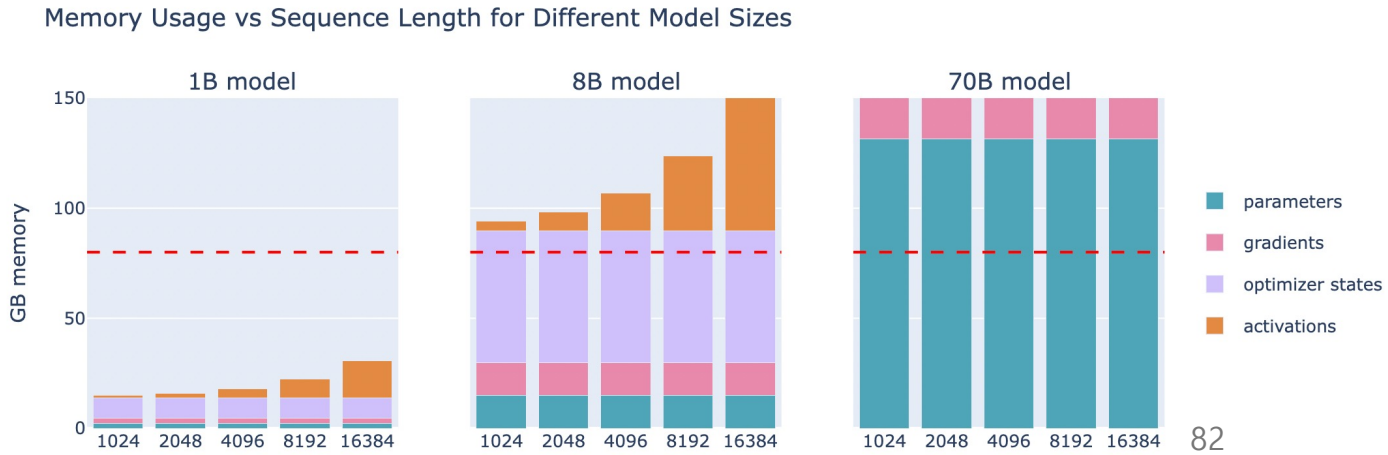


3-5. Our Journey Up to Now

C) Memory Usage vs. Sequence Length

- x,y축
  - [x축] Sequence Length
  - [y축] GPU Memory
  - 빨간 점선 = GPU 메모리 한계선(예: 80GB).
- 결론:
  - 8B 모델: Seq 8k~16k에서 OOM 위험
  - **70B 모델: Param/Optimizer 비율**이 절대적

모델	메모리 증가 원인
1B 모델	Activation이 작아서 전체 메모리 적음
8B 모델	Activation이 커지며 Sequence 길이에 따라 메모리 폭증
70B 모델	Parameter(파란색)가 압도적 → Sequence 길이에 거의 영향 없음



3-5. Our Journey Up to Now

D) Data Parallel 실전 세팅

- Step 1) **Global batch size (gbs)** 결정
  - 연구나 실험을 통해 가장 잘 수렴하는 batch 크기 선택 (ex) 4M tokens)
- Step 2) **Sequence length** 결정
  - 보통 2k~8k tokens (웹 데이터의 대부분 문서 길이가 4k 이하라서 효율적)
- Step 3) GPU 1개가 감당할 수 있는 **최대 micro-batch size (mbs)** 찾기
  - GPU 메모리에 맞춰 mbs를 조절 (OOM 안 날 정도로)
- Step 4) **GPU 수(dp)** 정하기
  - GPU 여러 대로 나눠서 병렬 처리
- Step 5) (남은 gbs를 채우기 위한) **grad_acc** 계산
  - gbs = mbs X grad_acc X dp

3-5. Our Journey Up to Now

D) Data Parallel 실전 세팅

- 예시) 목표: gbs = 4M tokens (seq = 4k → 1024 samples)
  - GPU 당 mbs = 2
  - GPU 수(dp) = 12
  - grad_acc = 1024 / (128 × 2) = 4 → grad_acc 4로 설정 시 목표 달성 가능!

3-5. Our Journey Up to Now

E) GPU가 더 많아지면?

- 예시) GPU 512대 사용
  - 같은 목표 batch(4M tokens)를 유지하려면 → **grad_acc = 1로 줄여도 충분**
  - 즉, GPU 수를 늘릴수록 순차 누적(accumulation) 대신 병렬 처리로 **속도 향상**
  - 하지만, 512대 이상부터는 **통신 병목 (ring latency)** 이 커져서 효율 급락 → **throughput 하락**

  → 이게 바로 **Data Parallelism의 스케일링 한계**

3-5. Our Journey Up to Now

F) DP의 한계 & 해결책

- 한계: DP는 **간단하고 강력**하지만
  - 모델 전체 복제 → **GPU 메모리 낭비**
  - **통신 오버헤드** → 대규모 GPU 환경에서 효율 급락
- 해결책:
  - DP 내 해결 책: **ZeRO (Sharded Optimizer)**
  - DP 외 해결책: **Tensor / Pipeline / Context Parallelism**

    → 즉, **Model 자체**를 쪼개는 방법들로 확장해야 함

3-6. Zero Redundancy Optimizer (ZeRO)

3-6. Zero Redundancy Optimizer

A) Overview

- 기본 DP의 한계점: GPU가 많아질수록 → **통신 오버헤드**!
- 기본 DP 예시: 기본 DP에서는 **각 GPU**가 모델의 **"완전한 복사본"**을 가짐
  - 세 가지 (Parameters + Gradients + Optimizer states) 모두 복사
  - 모델이 70B라면, GPU 8개일 때 → 모두 70B씩 들고 있음.
  - 즉, 메모리 사용이 **GPU 수 × 모델 크기**만큼 **중복**!

  → GPU가 늘어날수록 통신량 + 메모리 낭비가 폭증

3-6. Zero Redundancy Optimizer

A) Overview

- ZeRO (**Zero Redundancy** Optimizer) 란?
  - 목적: DP에서 생기는 **메모리 중복 (redundancy)** 을 없애기 위해 **메모리 최적화**
  - 핵심: "모든 GPU가 같은 걸 들고 있을 필요 X. **각 GPU가 일부만** 들고 **나머지는 다른 GPU에서** 가져오자"
  - 방법: Optimizer states, Gradients, Parameters를 **나누어 저장**
  - **DeepSpeed** 라이브러리에서 처음 구현됨

- ZeRO의 효과
  - 각 GPU가 메모리의 **1 / DP** 수만 차지
  - GPU 개수가 많을수록 **메모리 효율**이 **선형**으로 향상

3-6. Zero Redundancy Optimizer

A) Overview

- ZeRO의 세 단계 (종류)

단계	분할(Partitioning) 대상	메모리 절감 효과	추가 통신량
ZeRO-1	Optimizer states	약 4~8배 감소	낮음
ZeRO-2	Optimizer states + Gradients	약 8~16배 감소	중간
ZeRO-3	Optimizer states + Gradients + Parameters	**최대 절감 (모델 전체 1/DP 크기)**	**높음**

얼마나 분할할지 / 통신해야하는지도 결국 **trade-off** 문제

3-6. Zero Redundancy Optimizer

A) Overview

- ZeRO-1: **Optimizer State**
  - Optimizer state (예: Adam의 momentum, variance)을 각 GPU에 분산 저장.
  - GPU당 optimizer state 메모리 → 기존의 1/DP 수준으로 감소.
- ZeRO-2: **Optimizer State + Gradient**
  - Optimizer step 시, 필요한 gradient만 all-reduce → 통신량 약간 증가
  - Optimizer + Gradient 전체 1/DP로 축소
- ZeRO-3: **Optimizer State + Gradient + Parameter**
  - 각 GPU는 모델 전체의 일부 파라미터만 들고 있음.
  - Forward/Backward 시 필요한 파라미터만 통신으로 불러와 계산
  - 전체 메모리 footprint가 1/DP로 감소, but 통신량이 가장 많음

3-6. Zero Redundancy Optimizer

A) Overview

- Q1) 왜 **Activation**은 못 나누나?
- A1) 당연히 못 나눔. ZeRO도 DP 계열임.
  - Activations은 **입력 data가 다르기** 때문에 **GPU마다 서로 다른 값**! (즉, DP replica 간 **중복 X**)

  → Activation은 여전히 recomputation으로 해결해야 함

3-6. Zero Redundancy Optimizer

A) Overview

- Q2) 왜 Zero-1/2/3이 **Optimizer, Gradient, Parameter**순으로 추가가 된 것일까?
- A2)
  - **"효과 대비 복잡도"**면에서 가장 논리적인 순서

항목	메모리 비중(대략적)	중복 정도
Optimizer states	가장 큼 (파라미터의 2~4배)	완전 중복
Gradients	파라미터 크기와 동일	완전 중복
Parameters	파라미터 크기	완전 중복

즉, optimizer가 가장 크고, parameters는 가장 critical (항상 존재해야 함).

그래서 ZeRO는 **"크고 덜 중요한 것부터 제거"** 하는 방향으로 설계

3-6. Zero Redundancy Optimizer

A) Overview

- Q2) 왜 Zero-1/2/3이 **Optimizer, Gradient, Parameter**순으로 추가가 된 것일까?
- A2)

	분할 대상	선택 이유
Stage 1: **Optimizer**	momentum, variance	- 전체 메모리 중 **가장 큰 부분**을 차지 (weights의 2~4배) - Forward/Backward 시 필요 없음, 오직 **optimizer step에서만** 사용. → 즉, 사용 빈도가 **낮고**, 분할·통신이 가장 **쉽다**.
Stage 2: **+ Gradients**	Backward 결과물	- 메모리에서 **두 번째로 큰** 항목 - Gradient는 **backward** 중 생성되고, **optimizer step** 시점에서만 필요 - 따라서 전 구간에서 항상 상주할 필요가 없음 → 분할하기 용이
Stage 3: **+ Parameters**	모델 가중치	- 메모리에서 차지 비중은 크지만, **Forward/Backward 전 과정에서 항상 필요**. - 즉, 활성 사용 중(active use) → shard하면 **통신/재조립 비용이 큼** → 마지막 단계에서야 시도 가능

3-6. Zero Redundancy Optimizer

A) Overview

- Q2) 왜 Zero-1/2/3이 **Optimizer, Gradient, Parameter**순으로 추가가 된 것일까?
- A2) Summary
  - (1) **메모리 비중**이 큰 순서대로 (i.e., 가장 절약 효과가 큰 것부터)
  - (2) 계산 중 **사용 빈도**가 낮은 순서대로 (i.e., 통신·재조립 부담이 적은 것부터)
  - (3) **통신 오버헤드** 증가를 단계적으로 감당할 수 있게 설계

3-6. Zero Redundancy Optimizer

A) Overview

- Summary

항목	Data Parallelism (기본)	ZeRO-1	ZeRO-2	ZeRO-3
Optimizer states	전체 복제	✅ 분할	✅ 분할	✅ 분할
Gradients	전체 복제	복제	✅ 분할	✅ 분할
Parameters	전체 복제	복제	복제	✅ 분할
Activation	고유	고유	고유	고유
메모리 절감	❌ 없음	🔷 4~8x	🔶 8~16x	🟢 최대 (1/DP)
통신 비용	낮음	낮음	중간	높음

3-6. Zero Redundancy Optimizer

B) Memory Usage Revisited

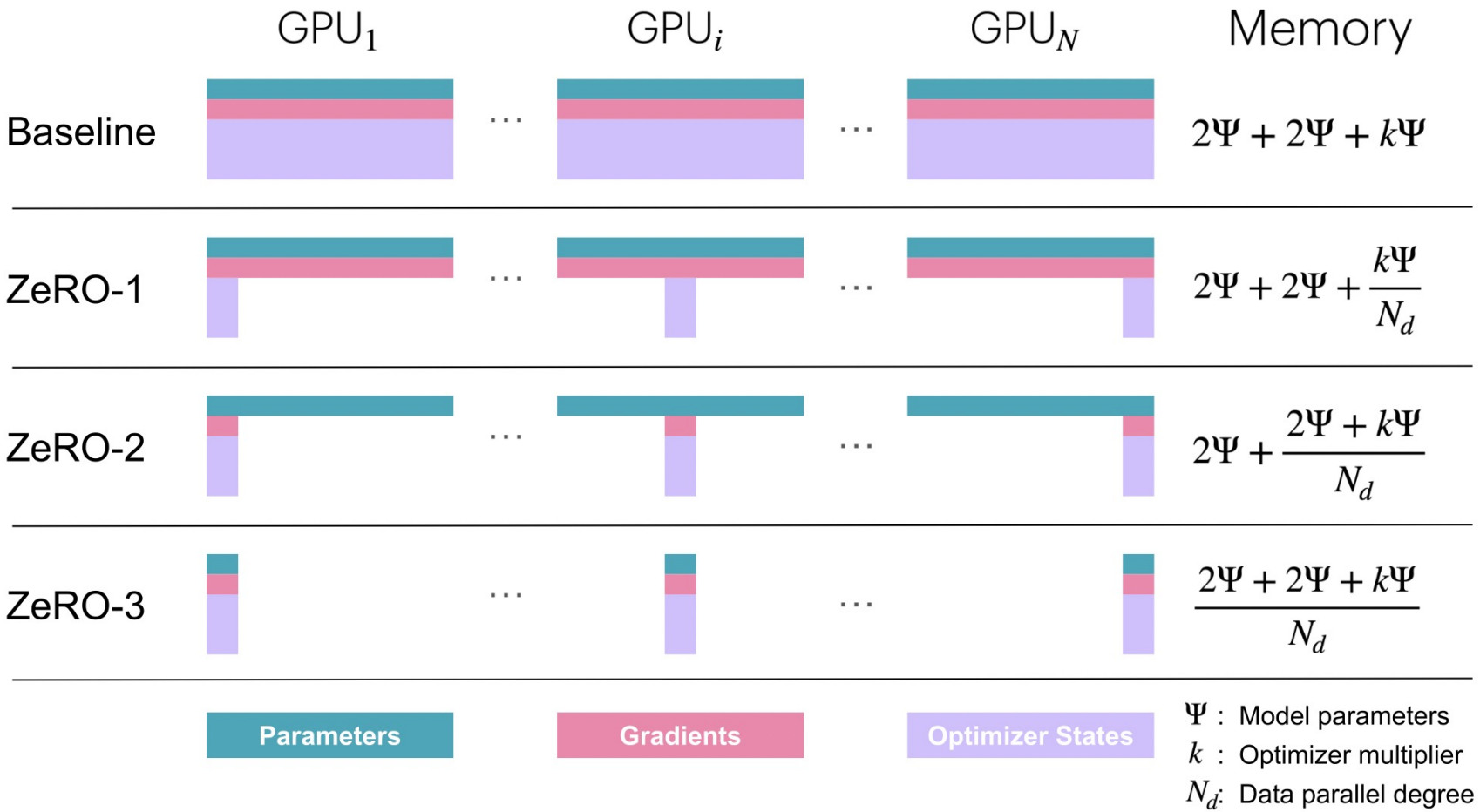
- "기존 DP는 왜 메모리가 많이 들고, **ZeRO는 어떻게 이를 줄이는가**"
- (ZeRO 논문에서) 자주 인용되는 **메모리 사용량 공식(Ψ, k, N_e)**
- Mixed Precision 기준 예시:

기호	의미
Ψ	# Params
k	Optimizer state의 메모리 배수 (Adam은 k=12)
N_d	DP degree = GPU 개수

구성 요소	precision	크기(단위: bytes × Ψ)
Parameters	BF16 (2B)	2Ψ
Gradients	BF16 (2B)	2Ψ
Optimizer	FP32 (4B)	$4\Psi + 4\Psi$
FP32 Parms	FP32 (4B)	4Ψ (master weights)
FP32 grad accum 버퍼	FP32 (4B)	4Ψ (OPTIONAL)

3-6. Zero Redundancy Optimizer

B) Memory Usage Revisited



3-6. Zero Redundancy Optimizer

C) Zero-1: Optimizer State

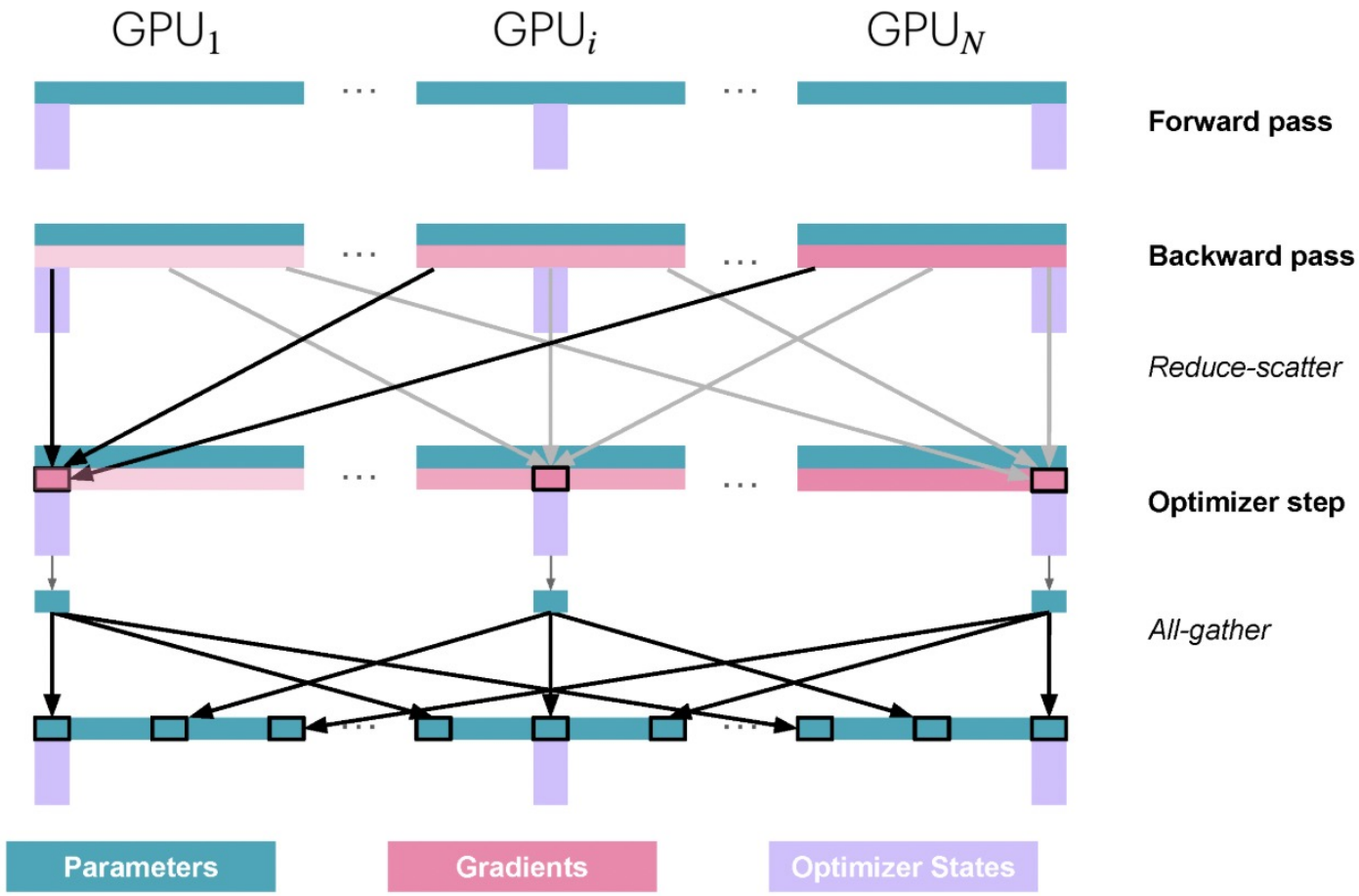
- 한 줄 요약: **Optimizer state**는 각 GPU가 전부 들고 있을 필요가 없으므로, DP 축으로 나눠 저장하자!
  - **GPU_1**은 optimizer state의 **첫 번째 $1/N_n$** 조각,
  - **GPU_2**는 optimizer state의 **두 번째 $1/N_n$** 조각,
  - …
  - **GPU_n**은 optimizer state의 **마지막 $1/N_n$** 조각,

→ GPU당 optimizer state 메모리 = 기존의 $1/N_n$ 수준으로 감소.

3-6. Zero Redundancy Optimizer

C) Zero-1: Optimizer State

- Procedure



3-6. Zero Redundancy Optimizer

C) Zero-1: Optimizer State

- **Reduce-scatter**
  - **"All-reduce의 절반만 수행"**
  - Gradient를 "합치면서" 동시에 각 GPU가 **"일부만"** 보관.
  - 속도: All-reduce보다 2배 빠름!
- **All-Gather**
  - **"분산된 파라미터를 다시 모으는 통신"**
  - Forward pass 전에 **full model 복원** 필요

3-6. Zero Redundancy Optimizer

C) Zero-1: Optimizer State

- Procedure

단계	과정	통신 종류	설명
① Forward pass	**모든 GPU**가 전체 파라미터로 서로 micro-batch를 처리	없음	Vanilla DP와 동일
② Backward pass	**GPU별**로 gradient 계산 완료 후, reduce-scatter 수행	**Reduce-scatter**	**모든 GPU의 grad 합산 후, 조각별로 분산 저장** (각 GPU는 자기 shard의 grad만 가짐)
③ Optimizer step	**GPU별**로 자신이 가진 shard의 optimizer state로만 업데이트 수행	없음	(전체가 아니라) $1/N_n$ 만큼의 optimizer state와 gradient만 사용
④ Parameter all-gather	Update된 조각들을 다시 모아 모든 GPU가 전체 weight를 공유	**All-gather**	Optimizer step 후, forward를 위해 **full parameter 복원**

3-6. Zero Redundancy Optimizer

C) Zero-1: Optimizer State

- All-reduce = Reduce-scatter + All-Gather
  - ZeRO-1은 기존의 “All-reduce”를“Reduce-scatter + All-gather” 로 분리하여 통신 효율을 개선

연산 이름	역할
All-reduce	모든 GPU의 데이터를 합침 (**N개**) → N개 sum/mean → **전체 (N개)**를 각 GPU에 나눠줌
Reduce-scatter	모든 GPU의 데이터를 합침 (**N개**) → N개 sum/mean → **본인 부분 (1개)**을 각 GPU에 나눠줌
All-gather	각 GPU가 가지고 있는 조각(**1개**) → N개 concatenate → **전체 (N개)**를 각 GPU에 나눠줌

3-6. Zero Redundancy Optimizer

C) Zero-1: Optimizer State

- All-reduce = Reduce-scatter + All-Gather
  - ZeRO-1은 기존의 “All-reduce”를“Reduce-scatter + All-gather” 로 분리하여 통신 효율을 개선

연산 이름	입력과 출력의 형태	직관적 비유	통신 비용 관계
All-reduce	- 입력: 각 GPU에 x_i - 출력: 모든 GPU가 동일한 합 Σx_i	“모든 GPU가 서로 결과를 공유”	전체 통신
Reduce-scatter	- 입력: 각 GPU에 x_i - 출력: 각 GPU는 합의 일부분(1/N)만 가짐	“모두가 요리를 만들고, 각자 1/N만 나눠 가지는 것”	All-reduce의 절반
All-gather	- 입력: 각 GPU에서 모아진 데이터 조각 - 출력: 모든 GPU가 전체 벡터를 가짐	“각자 들고 있던 조각 퍼즐을 모아 전체 퍼즐 완성”	Reduce-scatter와 비슷

3-6. Zero Redundancy Optimizer

C) Zero-1: Optimizer State

- Memory Usage

$$\text{Total Memory per GPU} = 2\Psi(\text{params}) + 2\Psi(\text{grads}) + \frac{k\Psi}{N_d}(\text{optimizer})$$

  - 2Ψ: parameters (half precision)
  - 2Ψ: gradients (half precision)
  - $k\Psi/N_d$: optimizer state (partition됨)
  - $k = 12$: Adam 기준

3-6. Zero Redundancy Optimizer

D) Vanilla vs. Zero-1

- (1) Setting

항목	값
W	[w1, w2, w3, w4]=[1.0, 2.0, 3.0, 4.0]
Optimizer	Adam (momentum m, variance v)
Learning rate	0.1
GPU 개수	2대 (DP degree = 2)
입력	$GPU_1 \rightarrow x_1 = 1$, $GPU_2 \rightarrow x_2 = 2$
Target	$y = 0$ (MSE Loss = $\frac{1}{2}(w \cdot x - 0)^2$)

- (2) Vanilla DP

항목	GPU_1	GPU_2
W	[1, 2, 3, 4]	[1, 2, 3, 4]
m	[0, 0, 0, 0]	[0, 0, 0, 0]
v	[0, 0, 0, 0]	[0, 0, 0, 0]

W & (m,v) **모두 동일한 복제본**을 가짐

3-6. Zero Redundancy Optimizer

D) Vanilla vs. Zero-1

- (3) Forward & Backward
  - 손실 $L = \frac{1}{2}(w \cdot x - 0)^2 \rightarrow grad = x \cdot (w \cdot x)$

항목	GPU_1 (x=1)	GPU_2 (x=2)
w·x	(1+2+3+4)*1 = 10	(1+2+3+4)*2 = 20
grad	[10, 10, 10, 10]	[40, 40, 40, 40]

- (4) All-reduce

$$g_{avg} = (g_{gpu1} + g_{gpu2})/2$$

$$g_{avg} = [25, 25, 25, 25]$$

=> 이걸 다시 모든 GPU로 분배

GPU	gradient shard
GPU_1	[25,25,25,25]
GPU_2	[25,25,25,25]

3-6. Zero Redundancy Optimizer

D) Vanilla vs. Zero-1

- (5) Update

GPU 1:

W	계산	업데이트 결과
w_1	$1 - 0.1 \times 25$	-1.5
w_2	$2 - 0.1 \times 25$	-0.5
w_3	$3 - 0.1 \times 25$	0.5
w_4	$4 - 0.1 \times 25$	1.5

GPU 2:

W	계산	업데이트 결과
w_1	$1 - 0.1 \times 25$	-1.5
w_2	$2 - 0.1 \times 25$	-0.5
w_3	$3 - 0.1 \times 25$	0.5
w_4	$4 - 0.1 \times 25$	1.5

3-6. Zero Redundancy Optimizer

D) Vanilla vs. Zero-1

- (1) Setting

항목	값
W	[w1, w2, w3, w4]=[1.0, 2.0, 3.0, 4.0]
Optimizer	Adam (momentum m, variance v)
Learning rate	0.1
GPU 개수	2대 (DP degree = 2)
입력	$GPU_1 \rightarrow x_1 = 1$, $GPU_2 \rightarrow x_2 = 2$
Target	y = 0 (MSE Loss = $\frac{1}{2}(w \cdot x - 0)^2$)

- (2) Vanilla DP (X), Zero-1 (O)

항목	GPU_1	GPU_2
W	[1, 2, 3, 4]	[1, 2, 3, 4]
m	[0, 0, 0, 0]	[0, 0, 0, 0]
v	[0, 0, 0, 0]	[0, 0, 0, 0]

항목	GPU_1	GPU_2
W	[1, 2, 3, 4]	[1, 2, 3, 4]
m	[m1, m2, -, -]	[-, -, m3, m4]
v	[v1, v2, -, -]	[-, -, v3, v4]

3-6. Zero Redundancy Optimizer

D) Vanilla vs. Zero-1

- (3) Forward & Backward
  - 손실 $L = ½(w \cdot x - 0)^2 \rightarrow grad = x \cdot (w \cdot x)$

항목	GPU_1 (x=1)	GPU_2 (x=2)
w·x	(1+2+3+4)*1 = 10	(1+2+3+4)*2 = 20
grad	[10, 10, 10, 10]	[40, 40, 40, 40]

- (4) All-reduce (X), **Reduce-Scatter (O)**

$$g_{avg} = (g_{gpu1} + g_{gpu2})/2$$

$$g_{avg} = [25, 25, 25, 25]$$

~~=> 이걸 다시 모든 GPU로 분배~~

GPU	gradient shard
GPU_1	[25,25,-,-]
GPU_2	[-,-,25,25]

~~| GPU | gradient shard |~~
~~| GPU_1 | [25,25,25,25] |~~
~~| GPU | [25,25,25,25] |~~

3-6. Zero Redundancy Optimizer

D) Vanilla vs. Zero-1

- (5) Update

GPU 1:

W	계산	업데이트 결과
w_1	1 – 0.1×25	–1.5
w_2	2 – 0.1×25	–0.5
w_3	3 – 0.1×25	0.5
w_4	4 – 0.1×25	1.5

GPU 2:

W	계산	업데이트 결과
w_1	1 – 0.1×25	–1.5
w_2	2 – 0.1×25	–0.5
w_3	3 – 0.1×25	0.5
w_4	4 – 0.1×25	1.5

→

W	계산	업데이트 결과
w_1	1 – 0.1×25	–1.5
w_2	2 – 0.1×25	–0.5

W	계산	업데이트 결과
w_3	3 – 0.1×25	0.5
w_4	4 – 0.1×25	1.5

3-6. Zero Redundancy Optimizer

D) Vanilla vs. Zero-1

- (6) All-gather

GPU 1:

W	계산	업데이트 결과
w_1	$1 - 0.1 \times 25$	-1.5
w_2	$2 - 0.1 \times 25$	-0.5

GPU 2:

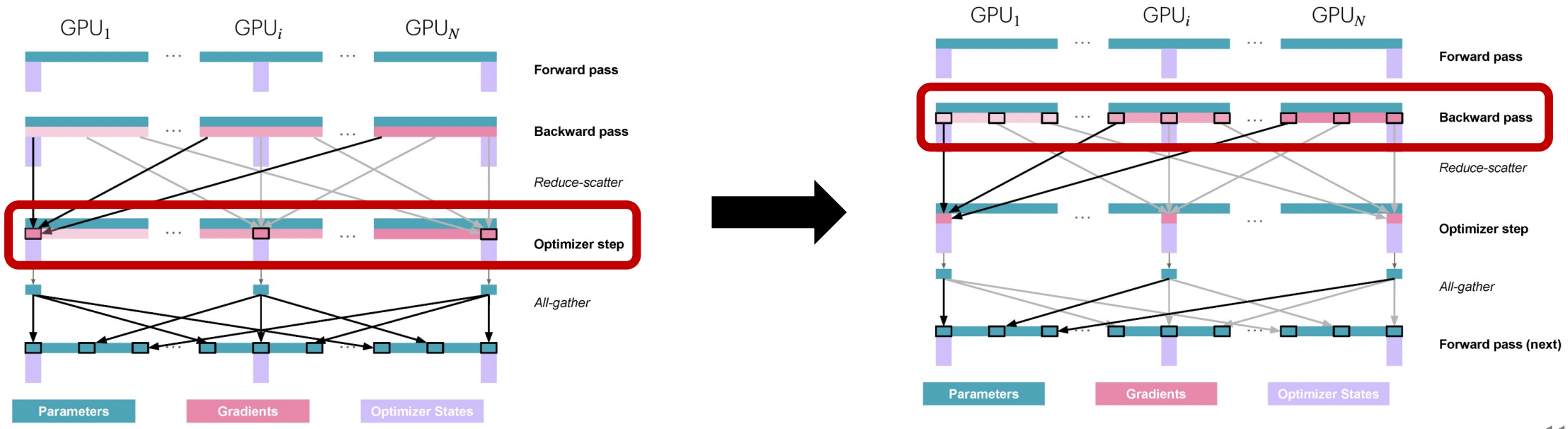
W	계산	업데이트 결과
w_3	$3 - 0.1 \times 25$	0.5
w_4	$4 - 0.1 \times 25$	1.5

W	계산	업데이트 결과
w_1	$1 - 0.1 \times 25$	-1.5
w_2	$2 - 0.1 \times 25$	-0.5
w_3	$3 - 0.1 \times 25$	0.5
w_4	$4 - 0.1 \times 25$	1.5

3-6. Zero Redundancy Optimizer

E) Zero-2: Optimizer State + Gradient

- 한 줄 요약: Optimizer state & Gradient DP 축으로 나눠 저장하자!
  - “Optimizer state를 shard했다면, 그에 대응되는 Gradient도 shard하는 게 논리적이지 않겠는가?”



3-6. Zero Redundancy Optimizer

E) Zero-2: Optimizer State + Gradient

- Procedure

단계	수행 내용	통신 연산	설명
① Forward pass	**모든** GPU가 **동일한 param**로 자기 micro-batch 처리	없음	기존과 동일
② Backward pass	**각** GPU가 local grad 계산 → 즉시 reduce-scatter	**Reduce-scatter**	전체 gradient 합치면서, 각 GPU는 자기 shard만 유지
③ Optimizer step	GPU는 자기 **gradient** shard + **optimizer state** shard 만 사용하여 업데이트	없음	메모리 사용 1/N_d 수준
④ All-gather	업데이트된 parameter shard들을 모아 전체 parameter 복원	All-gather	forward 재개 전, full 모델 복원

3-6. Zero Redundancy Optimizer

E) Zero-2: Optimizer State + Gradient

- Memory Usage

$$M_{\text{ZeRO-2}} = 2\Psi + \frac{2\Psi + k\Psi}{N_d}$$

항목	설명
2Ψ	Parameters (BF16, 전체 보유해야 함)
$\frac{2\Psi}{N_d}$	Gradients (Shard됨)
$\frac{k\Psi}{N_d}$	Optimizer states (Shard됨)

3-6. Zero Redundancy Optimizer

E) Zero-2: Optimizer State + Gradient

- 통신구조: Zero-1과 사실상 동일
  - Backward: **Reduce-scatter** (Zero-1: Optimizer step, Zero-2: Gradients)
  - Optimizer step 후: **All-gather** (Parameters)

→ ZeRO-1과 통신 패턴은 동일하지만

→ 메모리 점유는 더 작고, 통신 중간에 불필요한 Gradient는 즉시 해제 가능.

3-6. Zero Redundancy Optimizer

E) Zero-2: Optimizer State + Gradient

3-5. Zero Redundancy Optimizer

D) Vanilla vs. Zero-1

- (3) Forward & Backward
  - 손실 L = ½(w·x − 0)² → grad = x·(w·x)

항목	GPU₁ (x=1)	GPU₂ (x=2)
w·x	(1+2+3+4)*1 = 10	(1+2+3+4)*2 = 20
grad	[10, 10, 10, 10]	[40, 40, 40, 40]

- (4) All-reduce (X), **Reduce-Scatter (O)**

$$g_{avg} = (g_{gpu1} + g_{gpu2})/2$$

$$g_{avg} = [25, 25, 25, 25]$$

=> 이걸 다시 모든 GPU로 분배

GPU	gradient shard
GPU₁	[25,25,-,-]
GPU₂	[-,-,25,25]

GPU	gradient shard
GPU₁	[25,25,25,25]
GPU₂	[25,25,25,25]

여기서 [25,25,25,25] 조차도 나눠서 계산!

즉, [25,25,-,-] & [-,-,25,25]

3-6. Zero Redundancy Optimizer

F) Zero-3: Optimizer State + Gradient + Parameter

- **ZeRO-3** == **Parameter** Partitioning == **FSDP** (Fully Sharded Data Parallel)
- Parameter까지 shard(분할) 해서 forward/backward 중에 필요할 때만 불러옴
- 이제 각 GPU는:
  - Model param 전체를 저장하지 않음
  - **자신에게 할당된 param 조각만** 들고 있음
  - 계산 시에는 “필요한 param 조각을 **all-gather**로 모아서 사용한 뒤 flush”

연산 이름	역할
All-reduce	모든 GPU의 데이터를 합쳐서, 각 GPU에 동일한 결과를 나눠줌
Reduce-scatter	모든 GPU의 데이터를 합친 뒤, 그 합을 조각으로 나눠 각 GPU에 분배
All-gather	각 GPU가 가지고 있는 조각(1/N)을 서로 교환해서 모두가 전체를 갖도록 복원

3-6. Zero Redundancy Optimizer

F) Zero-3: Optimizer State + Gradient + Parameter

- **Forward pass** 과정
  - Parameter가 GPU마다 나뉘어 있으므로, forward 중에는 **각 layer의 param을 모아야** 함
  - 각 layer 계산 전 다음 과정을 수행:
    - 아래 과정을 layer별로 반복
    - 즉, GPU는 한 시점에 현재 layer의 weight만 들고 있음

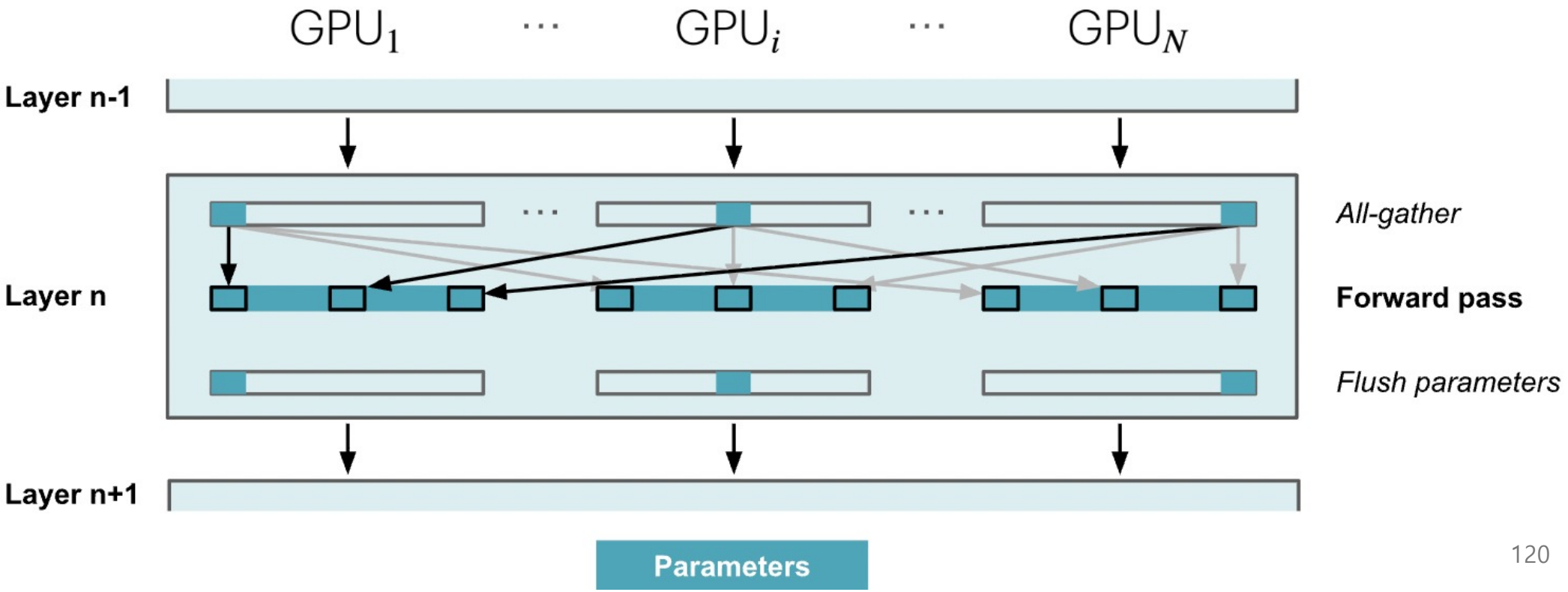
단계	동작	통신
(1) **All-gather**	해당 layer의 파라미터 shards를 전부 **모음** → full weight 복원	GPU ↔ GPU
(2) Forward pass	Layer 연산 수행	Local compute
(3) **Flush** parameters	해당 layer 계산이 끝나면 weight를 메모리에서 **삭제**	Free memory

3-6. Zero Redu

연산 이름	역할
All-reduce	모든 GPU의 데이터를 합침 (N개) → N개 sum/mean → 전체 (N개)를 각 GPU에 나눠줌
Reduce-scatter	모든 GPU의 데이터를 합침 (N개) → N개 sum/mean → 본인 부분 (1개)을 각 GPU에 나눠줌
All-gather	각 GPU가 가지고 있는 조각(1개) → N개 concatenate → 전체 (N개)를 각 GPU에 나눠줌

F) Zero-3: Optimizer State ·

- Forward pass 과정



3-6. Zero Redundancy Optimizer

F) Zero-3: Optimizer State + Gradient + Parameter

- **Backward pass** 과정
  - Forward의 반대 방향으로 진행
  - 각 layer 계산 전 다음 과정을 수행:
    - 아래 과정을 layer별로 반복
    - 필요없는 gradient는 즉시 해제 (flush)

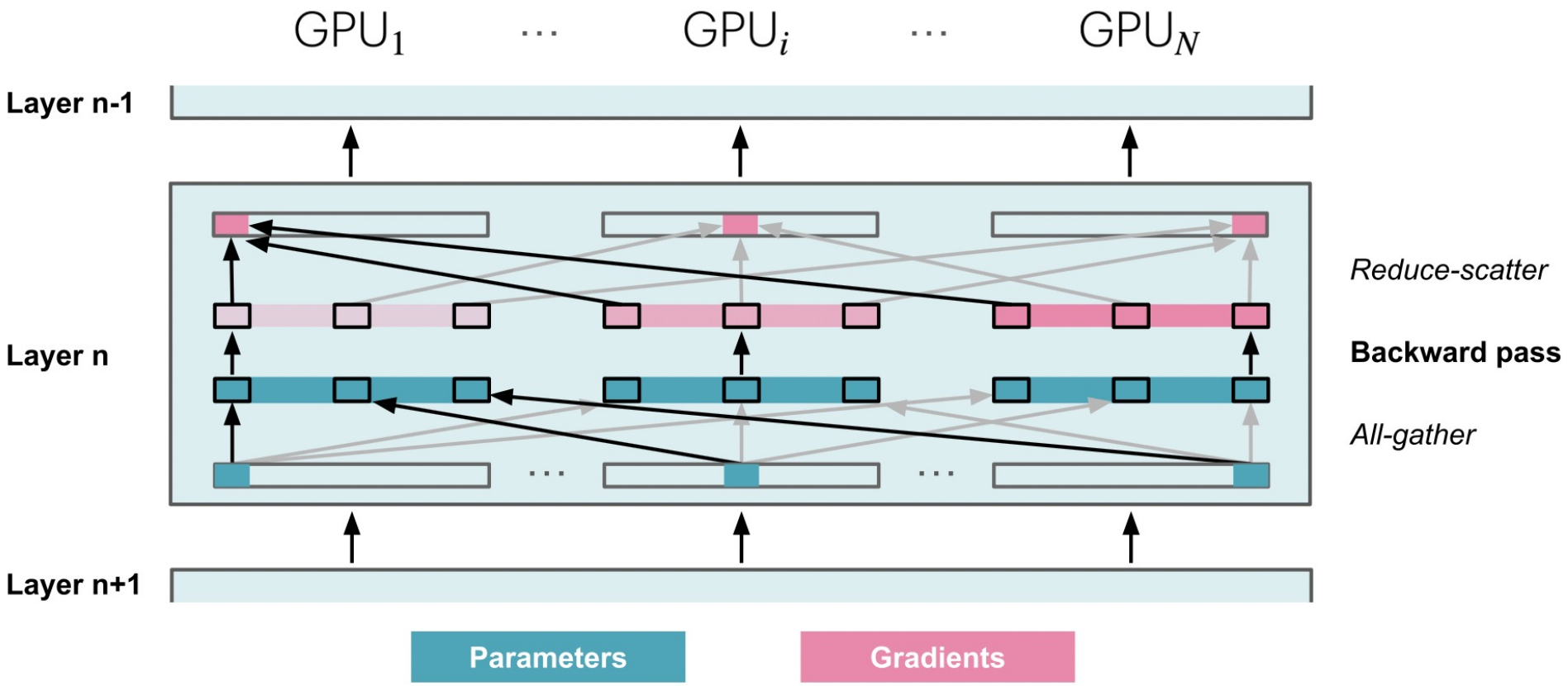
단계	동작	통신
(1) **All-gather**	Backward 시 필요한 layer의 파라미터 shards 복원	GPU ↔ GPU
(2) Backward pass	Gradient 계산	Local compute
(3) **Reduce-scatter**	Gradient shard를 합산 후 각 GPU가 자기 shard만 저장	GPU ↔ GPU

3-6. Zero Redu

연산 이름	역할
All-reduce	모든 GPU의 데이터를 합침 (N개) → N개 sum/mean → 전체 (N개)를 각 GPU에 나눠줌
Reduce-scatter	모든 GPU의 데이터를 합침 (N개) → N개 sum/mean → 본인 부분 (1개)을 각 GPU에 나눠줌
All-gather	각 GPU가 가지고 있는 조각(1개) → N개 concatenate → 전체 (N개)를 각 GPU에 나눠줌

F) Zero-3: Optimizer State ·

- **Backward pass** 과정



3-6. Zero Redundancy Optimizer

F) Zero-3: Optimizer State + Gradient + Parameter

- Memory 절감

$$M_{\text{ZeRO-3}} = \frac{2\Psi + 2\Psi + k\Psi}{N_d}$$

	설명
2Ψ	Parameters (BF16, shard)
2Ψ	Gradients (shard)
kΨ	Optimizer states (shard)
N_d	DP degree (GPU 개수)

하지만! Activation 메모리는 여전히 줄어들지 않음.

→ Sequence length, batch size가 크면 여전히 병목!

3-6. Zero Redundancy Optimizer

F) Zero-3: Optimizer State + Gradient + Parameter

- 통신 최적화: **Pre-fetching**
  - ZeRO-3/FSDP는 **"통신과 계산을 겹치는(pre-fetch)"** 방식으로 통신 비용을 줄임
  - 아래와 같이 하면, 통신이 연산에 숨겨져(**Overlapped**) 실질적인 성능 저하가 거의 없음
    - 단, 너무 많은 GPU (예: DP > 512)에서는 통신 지연(latency)이 커져 overlap 효과가 줄어듬

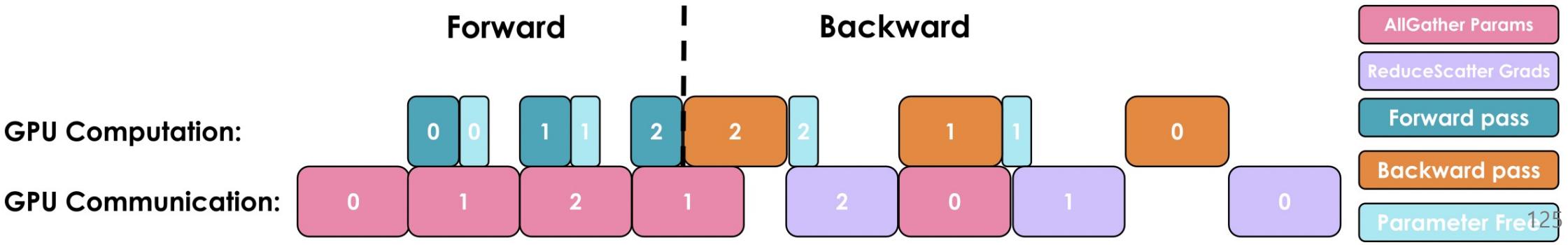
	수행 동작
Forward 중	**Layer n**을 계산하면서 **Layer n+1**의 파라미터를 미리 all-gather
Backward 중	**Layer n**의 backward를 하면서 **Layer n−1**의 파라미터를 미리 all-gather

3-6. Zero Redundancy Optimizer

F) Zero-3: Optimizer State + Gradient + Parameter

- 통신 최적화: **Pre-fetching**

	수행 동작
Forward 중	**Layer n**을 계산하면서 **Layer n+1**의 파라미터를 미리 all-gather
Backward 중	**Layer n**의 backward를 하면서 **Layer n−1**의 파라미터를 미리 all-gather



3-6. Zero Redundancy Optimizer

F) Zero-3: Optimizer State + Gradient + Parameter

- Summary

항목	설명
Sharded Objects	**Optimizer + Gradient + Parameter**
통신 패턴	**All-gather (weight)** + **Reduce-scatter (grad)**
통신량	약 3Ψ
계산 구조	Layer-by-layer: 필요 시 All-gather → 계산 → Flush
메모리 절감	ZeRO-2보다 약 2× 추가 절감
실제 구현	PyTorch **FSDP** (Fully Sharded Data Parallel)
남은 한계	Activation 메모리는 여전히 커서 별도 대책 필요 (예: checkpointing)

3-6. Zero Redundancy Optimizer

G) Summary: Zero-1 vs. 2 vs. 3

- **통신 비용** 비교
  - 나누지 않은 것이 곧 통신의 비용이라고 생각하면 됨 (메모리 & 통신의 trade-off)

단계	통신 비용
ZeRO-1	$\approx 2\Psi$
ZeRO-2	$\approx 2\Psi$
ZeRO-3	$\approx 3\Psi$

3-6. Zero Redundancy Optimizer

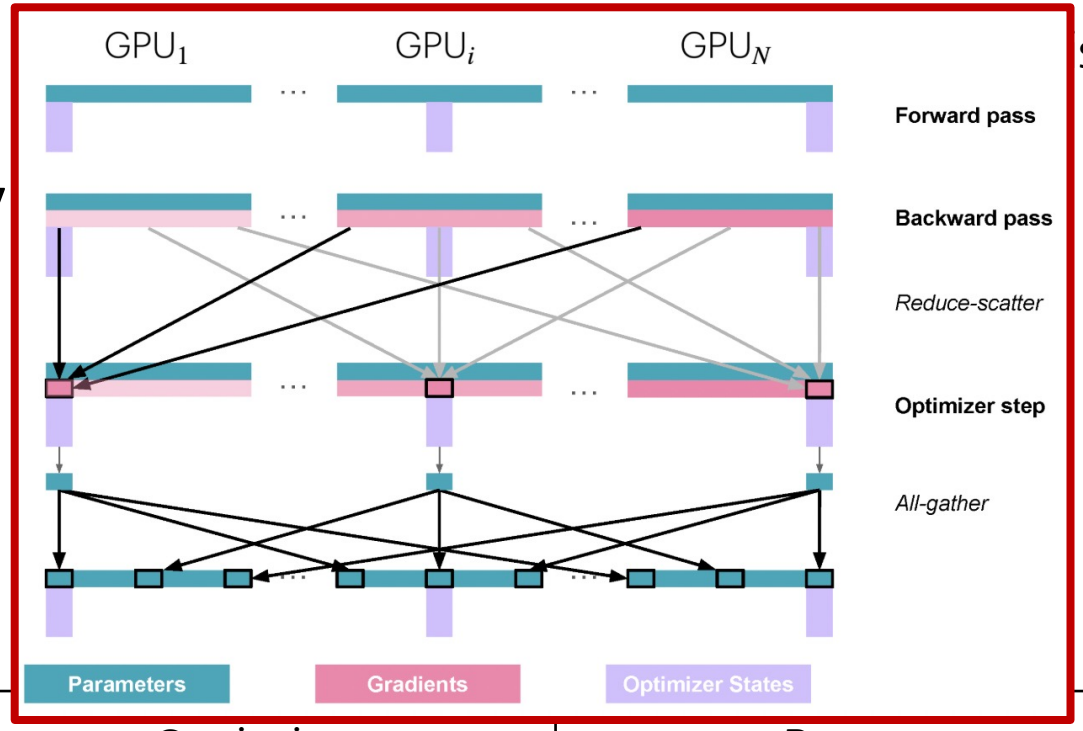
G) Summary: Zero-1 vs. 2 vs. 3

- **통신 비용** 비교
  - 나누지 않은 것이 곧 통신의 비용이라고 생각하면 됨 (메모리 & 통신의 trade-off)

단계	Gradient	Optimizer	Parameter
ZeRO-1	Reduce-scatter ($\approx \Psi$) → Backward 중 grad 합산 후 각 GPU에 shard로 분배	All-gather ($\approx \Psi$) → Optimizer step 후 파라미터 업데이트 결과를 전체로 복원	없음 (모델은 full copy 유지)
ZeRO-2	Reduce-scatter ($\approx \Psi$) → Grad를 합치며 shard로 저장(기존 all-reduce 대체)	통신 없음 (optim state shard 로컬 저장)	All-gather ($\approx \Psi$) → optimizer step 후 파라미터 전체 복원
ZeRO-3	Reduce-scatter ($\approx \Psi$) → Backward 중 grad 합산 및 shard 저장	통신 없음 (로컬 shard 관리)	All-gather ($\approx \Psi \times 2$) → 각 layer for ward 및 backward 시 필요할 때 on-demand 복원 (Flush 후 재수집)

3-6. Zero Redundancy

sm



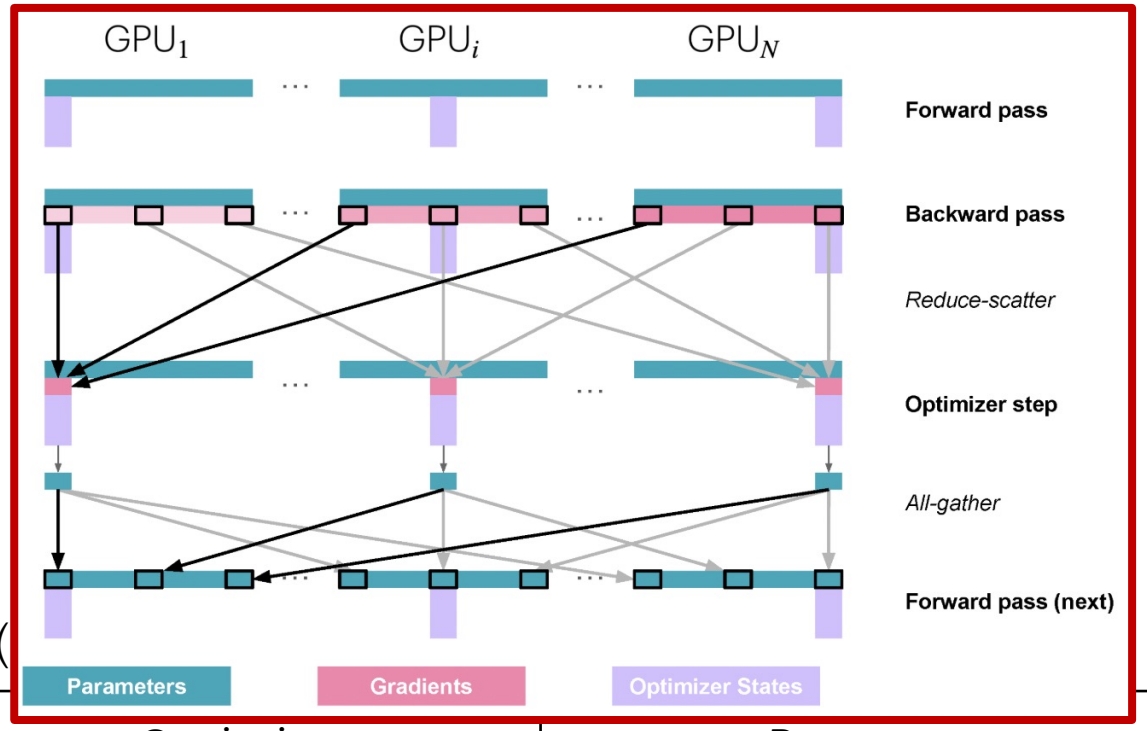
G) Summary: Zero-1 vs. 2 vs. 3

- **통신 비용** 비교
  - 나누지 않은 것이 곧 통신의 비용이라고 생각하면 됨

단계	Gradient	Optimizer	Parameter
ZeRO-1	**Reduce-scatter** ($\approx \Psi$) → Backward 중 grad 합산 후 각 GPU에 shard로 분배	**All-gather** ($\approx \Psi$) → Optimizer step 후 파라미터 업데이트 결과를 전체로 복원	없음 (모델은 full copy 유지)
ZeRO-2	Reduce-scatter ($\approx \Psi$) → Grad를 합치며 shard로 저장(기존 all-reduce 대체)	통신 없음 (optim state shard 로컬 저장)	All-gather ($\approx \Psi$) → optimizer step 후 파라미터 전체 복원
ZeRO-3	Reduce-scatter ($\approx \Psi$) → Backward 중 grad 합산 및 shard 저장	통신 없음 (로컬 shard 관리)	All-gather ($\approx \Psi \times 2$) → 각 layer forward 및 backward 시 필요할 때 on-demand 복원 (Flush 후 재수집)

3-6. Zero Redundancy

G) Summary: Zero-1 vs. 2 vs. 3



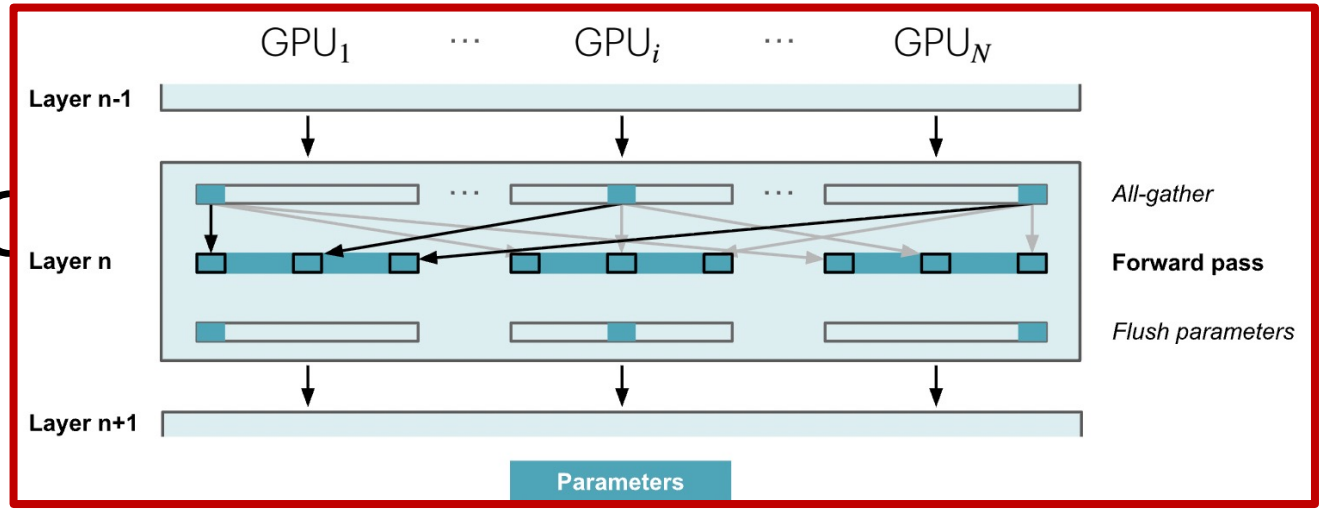
- **통신 비용** 비교
  - 나누지 않은 것이 곧 통신의 비용이라고 생각하면 됨 (

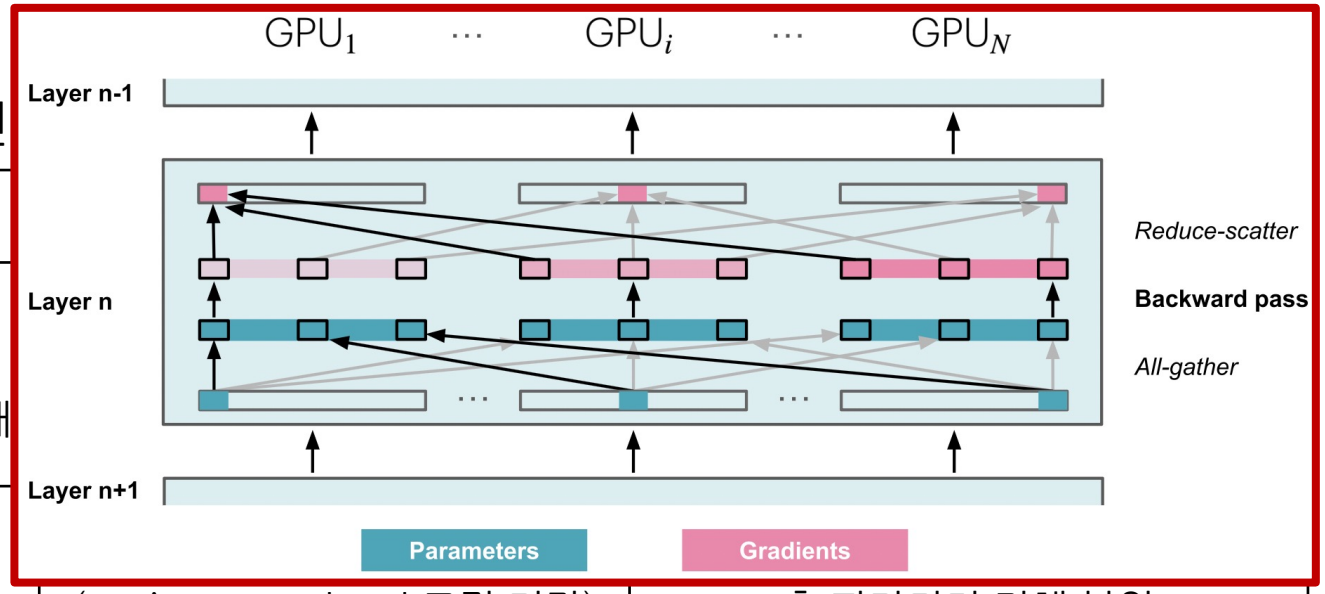
단계	Gradient	Optimizer	Parameter
ZeRO-1	**Reduce-scatter** ($\approx \Psi$) → Backward 중 grad 합산 후 각 GPU에 shard로 분배	**All-gather** ($\approx \Psi$) → Optimizer step 후 파라미터 업데이트 결과를 전체로 복원	없음 (모델은 full copy 유지)
ZeRO-2	Reduce-scatter ($\approx \Psi$) → Grad를 합치며 shard로 저장(기존 all-reduce 대체)	통신 없음 (optim state shard 로컬 저장)	All-gather ($\approx \Psi$) → optimizer step 후 파라미터 전체 복원
ZeRO-3	Reduce-scatter ($\approx \Psi$) → Backward 중 grad 합산 및 shard 저장	통신 없음 (로컬 shard 관리)	All-gather ($\approx \Psi \times 2$) → 각 layer for ward 및 backward 시 필요할 때 on-demand 복원 (Flush 후 재수집)

3-6. Zero Redundanc

G) Summary: Zero-1 vs. 2 vs. 3

- **통신 비용** 비교
  - 나누지 않은 것이 곧 통신의 비용이라고 생각하면



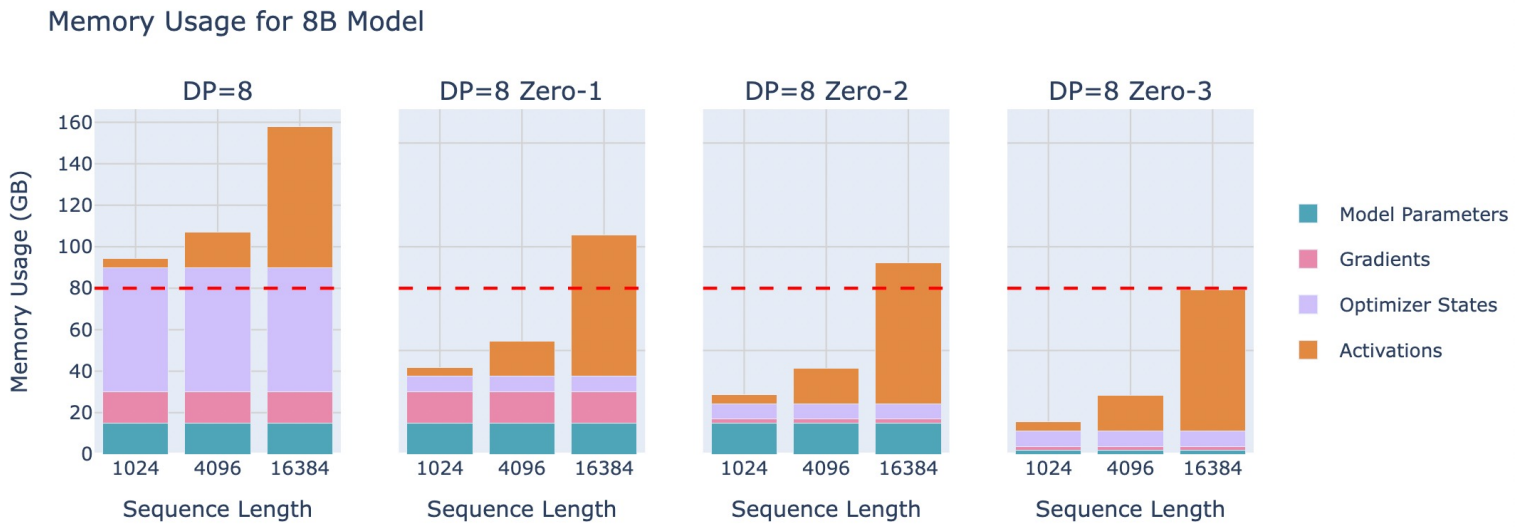


단계	Gradient		
ZeRO-1	Reduce-scatter ($\approx \Psi$) → Backward 중 grad 합산 후 각 GPU에 shard로 분배		
ZeRO-2	Reduce-scatter ($\approx \Psi$) → Grad를 합치며 shard로 저장(기존 all-reduce 대체)		(optim state shard 로컬 저장) 후 파라미터 전체 복원
ZeRO-3	Reduce-scatter ($\approx \Psi$) → Backward 중 grad 합산 및 shard 저장	통신 없음 (로컬 shard 관리)	All-gather ($\approx \Psi \times 2$) → 각 layer forward 및 backward 시 필요할 때 on-demand 복원 (Flush 후 재수집)

3-6. Zero Redundancy Optimizer

G) Summary: Zero-1 vs. 2 vs. 3

- **Memory** 비교
  - ZeRO-3: 8B 모델조차 한 GPU 80GB 내에서 안정적으로 학습 가능



DP	설명
(기본)	모든 GPU가 full 모델 복사본 → 메모리 폭발 (약 150GB 이상)
ZeRO-1	Optimizer만 shard → 메모리 약 **100GB**
ZeRO-2	Optimizer + Gradient shard → 메모리 약 **70GB**
ZeRO-3	Parameter까지 shard → 메모리 약 **40GB**

4. Tensor Parallelism

4-1. Tensor Parallelism

4-1. Tensor Parallelism

A) Overview

- **Tensor Parallelism (TP)**의 개념:
  - 행(row) or 열(column) 방향으로 **Tensor를 쪼개어** 여러 GPU에서 동시에 **행렬 곱(matmul)**
  - **한 layer의 연산**을 여러 GPU가 나누어서 동시에 수행하는 병렬화
- ZeRO (of DP) vs. TP:
  - **ZeRO**: 모델 전체의 파라미터를 분할 (= **"layer 간"**의 메모리 중복 제거)
  - **TP**: 하나의 layer 내부의 행렬 곱 연산까지도 분할 (= **"layer 내부"**의 연산을 병렬화)

4-1. Tensor Parallelism

B) Tensor Parallelism의 필요성

- **ZeRO-3**로도 activation memory는 줄일 수 없었음
- **TP**는 weight, gradient, optimizer state뿐만 아니라, **activation까지 GPU 간에 나눠 계산**

→ GPU당 연산 + 메모리 부담을 동시에 줄임

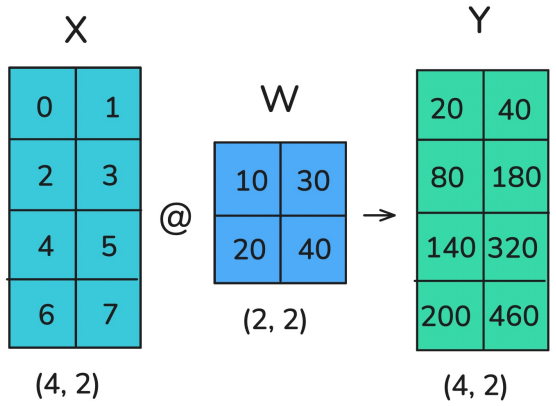
4-1. Tensor Parallelism

C) 행렬곱의 분해

- 아래와 같이 **행렬곱**을 수행
- 두 가지로 나뉠 수 있음
  - (1) **Column** parallelism
  - (2) **Row** parallelism

$$A \times B = [A_1, A_2, \ldots] \begin{bmatrix} B_1 \\ B_2 \\ \vdots \end{bmatrix} = \sum_i A_i B_i$$

- **입력 X**는 모든 GPU에 **broadcast** 필요
- **출력 Y**는 각 GPU가 부분 결과를 **all-gather**로 모음

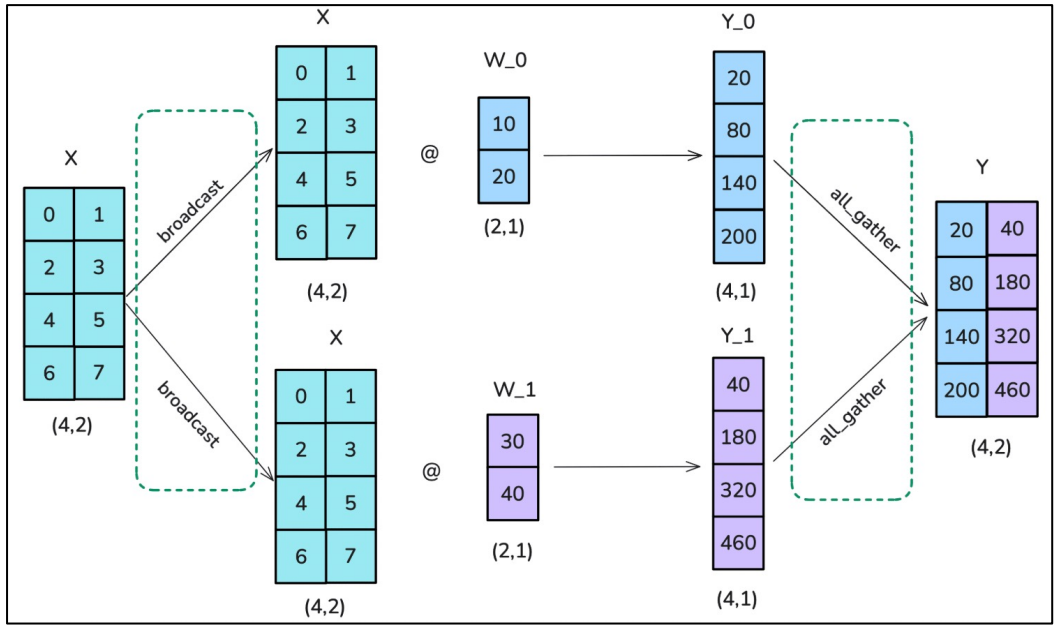


	연산	통신	설명
(1)	**입력** X를 **broadcast**	Broadcast	모든 GPU가 동일한 입력을 받음
(2)	**각 GPU**가 **자기 열 shard인 W_i**와 곱: $X \times W_i$	**Local compute**	각 GPU가 partial output Y_i 생성
(3)	Partial 결과들을 **all-gather**	All-gather	모든 Y_i를 합쳐 최종 Y 완성

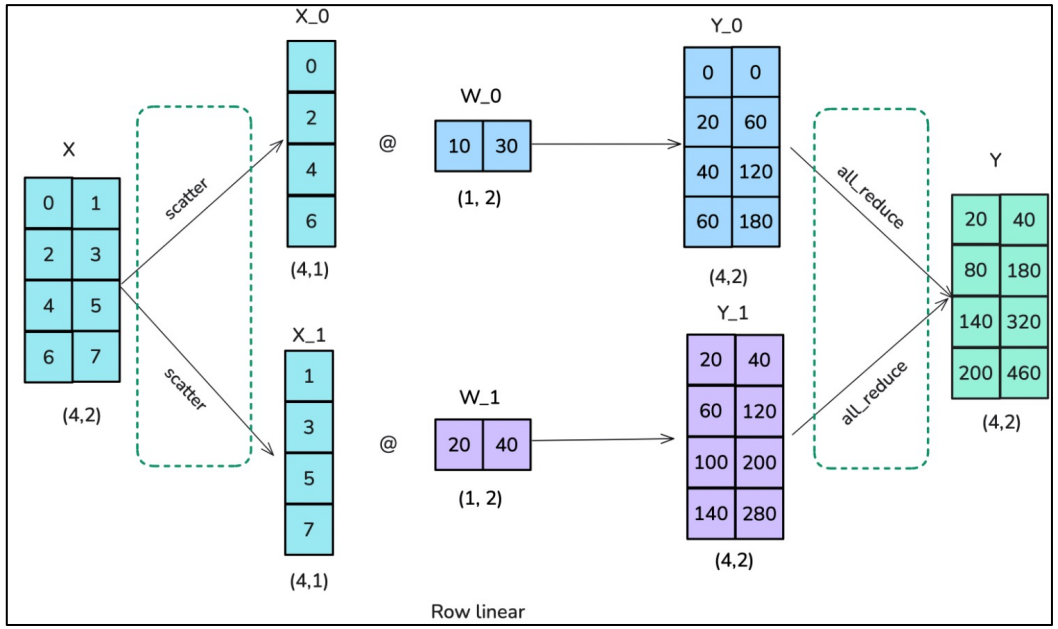
4-1. Tensor Pa

연산 이름	역할
All-reduce	모든 GPU의 데이터를 합침 (**N개**) → N개 sum/mean → **전체 (N개)**를 각 GPU에 나눠줌
Reduce-scatter	모든 GPU의 데이터를 합침 (**N개**) → N개 sum/mean → **본인 부분 (1개)**을 각 GPU에 나눠줌
All-gather	각 GPU가 가지고 있는 조각(**1개**) → N개 concatenate → **전체 (N개)**를 각 GPU에 나눠줌

C) 행렬곱의 분해



Column Parallelism



Row Parallelism

4-1. Tensor Parallelism

C) 행렬곱의 분해

	Column Parallelism	Row Parallelism
Weight 분할 기준	열(column)	행(row)
입력 X	Broadcast (모두 동일 입력 사용)	Scatter (입력 일부만)
출력 Y	all-gather (결과 합치기)	all-reduce (결과 더하기)
메모리 효율	X 중복 저장 있음	X 분할로 더 효율적
통신 형태	broadcast + all-gather	scatter + all-reduce
주요 예시	Transformer의 MLP (W_2)	Transformer의 Attention (W_QKV)

4-1. Tensor Parallelism

C) 행렬곱의 분해

	설명
Column Parallelism	Weight를 **열**로 분할 → 각 GPU가 **출력의 일부**를 계산 → **all-gather**로 병합
Row Parallelism	Weight를 **행**으로 분할 → 각 GPU가 **입력의 일부**를 계산 → **all-reduce**로 병합
공통점	**Forward/Backward 계산을 병렬화**하여 GPU당 연산·메모리 부담 감소
차이점	**통신 방향**: Column(출력 병합) / Row(입력 병합)
목표	ZeRO가 줄이지 못한 **activation** 및 연산 병목 해결

4-1. Tensor Parallelism

D) Summary

- 모델의 weight, gradient, optimizer state뿐만 아니라, **activation**까지 GPU 간에 분할(shard)
- ZeRO와 달리 "**계산 자체를 나눔**"
- **All-gather / All-reduce** 통신으로 partial 결과를 병합

4-2. Tensor Parallelism in a Transformer Block

4-2. Tensor Parallelism in a Transformer Block

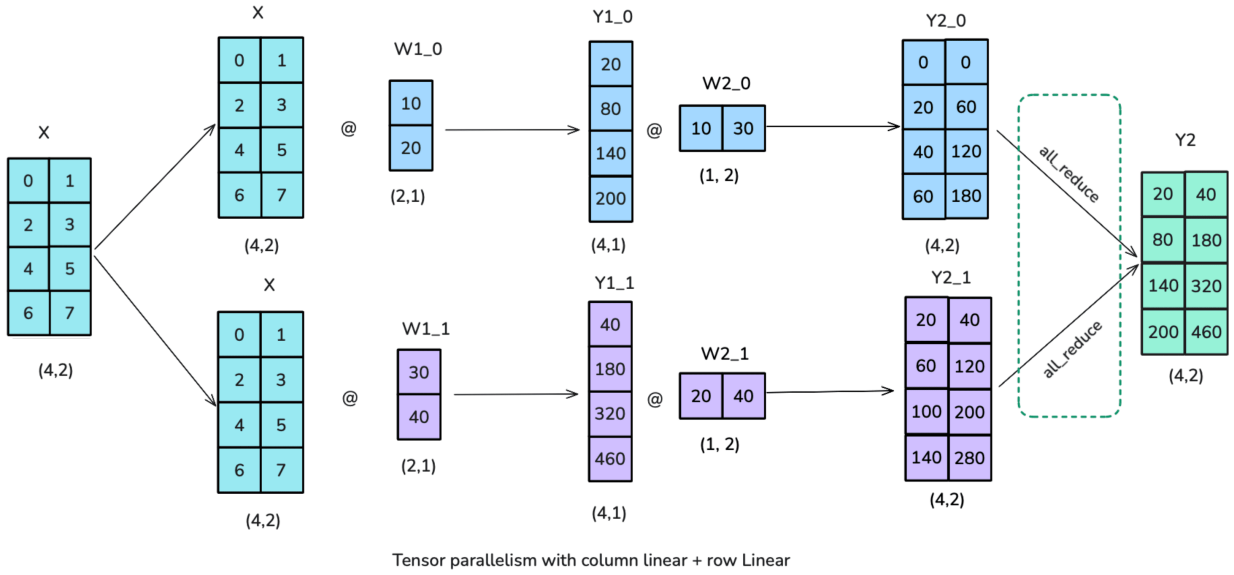
A) Transformer의 두 구성요소에 TP 적용

- (1) **FeedForward (MLP)**
  - 구조: $FC_1 \rightarrow GeLU \rightarrow FC_2$
  - 적용 방식: **Column** Linear → **Row** Linear
- (2) **Multi-Head Attention (MHA)**
  - Query(Q), Key(K), Value(V) projection: **Column** Parallel
  - Output projection: **Row** Parallel

4-2. Tensor Parallelism in a Transformer Block

A) Transformer의 두 구성요소에 TP 적용

- (1) **FeedForward (MLP)**
  - 구조: $FC_1 \rightarrow GeLU \rightarrow FC_2$
  - 적용 방식: **Column** Linear → **Row** Linear



4-2. Tensor Parallelis k

	Column Parallelism	Row Parallelism
Weight 분할 기준	열(column)	행(row)
입력 X	broadcast (모두 동일 입력 사용)	scatter (입력 일부만)
출력 Y	all-gather (결과 합치기)	all-reduce (결과 더하기)
메모리 효율	X 중복 저장 있음	X 분할로 더 효율적
통신 형태	broadcast + all-gather	scatter + all-reduce
주요 예시	Transformer의 MLP (W_2)	Transformer의 Attention (W_QKV)

A) Transformer의 두 구성요소에 TP 적

- (1) **FeedForward (MLP)**
  - 구조: FC_1 → GeLU → FC_2
  - 적용 방식: **Column** Linear → **Row** Linear

Column → Row 조합을 쓰는 이유?

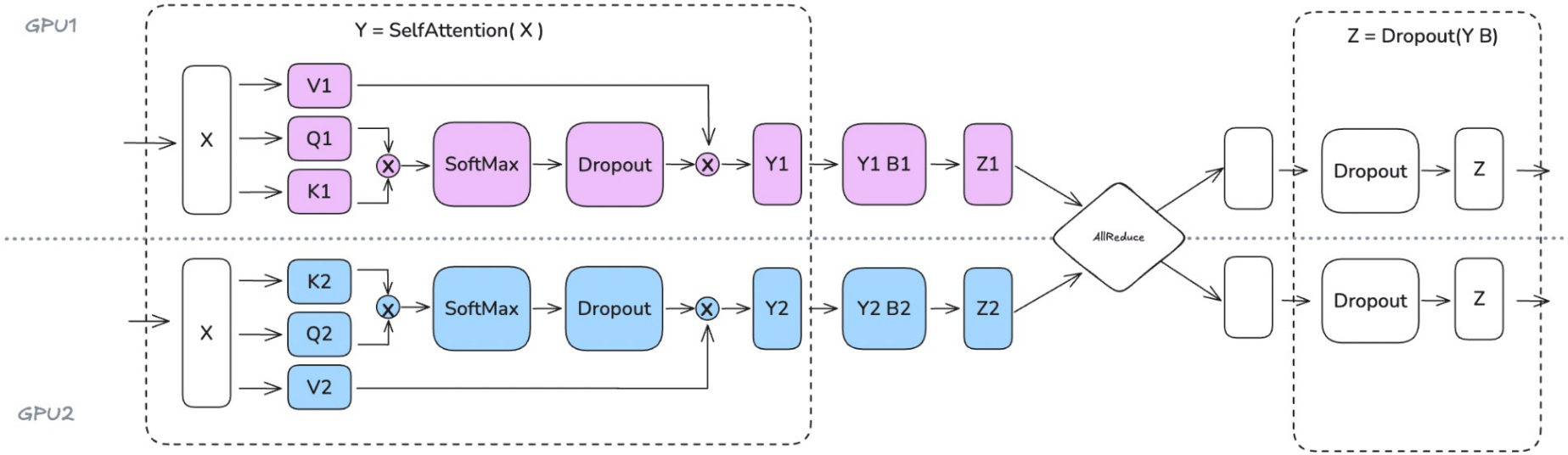
- 중간 **all-reduce를 생략**할 수 있어서 더 효율적!

단계	분할 방식	통신	설명
FC_1	weight의 **column** 방향	**Broadcast** (X) + All-gather (Y)	각 GPU가 **출력의 일부(Y_1, Y_2)만** 계산 후, all-gather로 결합
FC_2	weight의 **row** 방향	Scatter (X) + **All-reduce** (Y)	입력 일부만 곱하고, 결과를 **all-reduce로 합산**

4-2. Tensor Parallelism in a Transformer Block

A) Transformer의 두 구성요소에 TP 적용

- (2) **Multi-Head Attention (MHA)**
  - Query(Q), Key(K), Value(V) projection: **Column** Parallel
  - Output projection: **Row** Parallel



4-2. Tensor Parallelis k

	Column Parallelism	Row Parallelism
Weight 분할 기준	열(column)	행(row)
입력 X	broadcast (모두 동일 입력 사용)	scatter (입력 일부만)
출력 Y	all-gather (결과 합치기)	all-reduce (결과 더하기)
메모리 효율	X 중복 저장 있음	X 분할로 더 효율적
통신 형태	broadcast + all-gather	scatter + all-reduce
주요 예시	Transformer의 MLP (W_2)	Transformer의 Attention (W_QKV)

A) Transformer의 두 구성요소에 TP 적

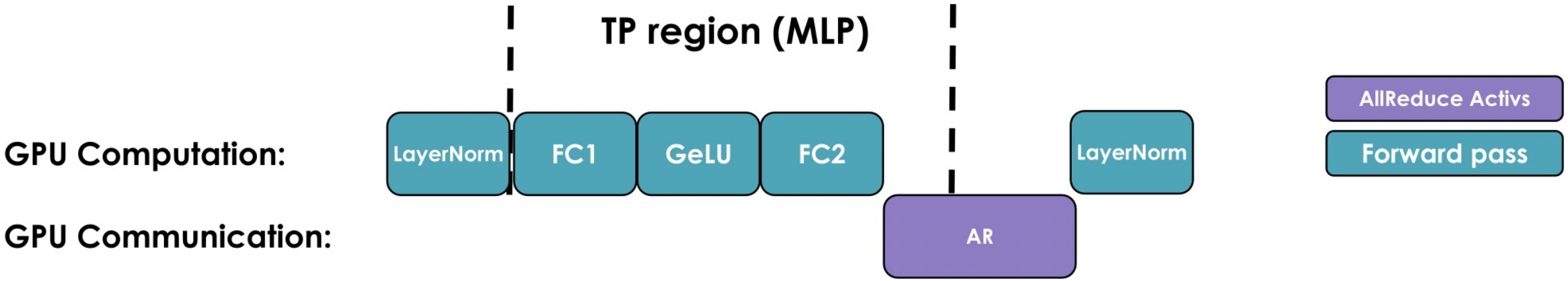
- (2) **Multi-Head Attention (MHA)**
  - Query(Q), Key(K), Value(V) projection: **Column** Parallel
  - Output projection: **Row** Parallel

- Attention head들이 독립적이므로 **head 단위로 나누기 쉬움**
- TP degree ≤ num_heads (예: 32 heads → TP 최대 32)
- MQA/GQA에서는 KV head 수가 적으므로 동기화에 주의 필요

구성 요소	분할 방식	설명
Q, K, V	**Column** Parallel	각 GPU가 **일부 head**만 담당 (예: $GPU_1 \rightarrow Head_{1-4}$, $GPU_2 \rightarrow Head_{5-8}$)
Output (W^O)	**Row** Parallel	각 GPU의 attention 결과를 **all-reduce**로 합침

4-2. Tensor Parallelism in a Transformer Block

B) TP Timeline

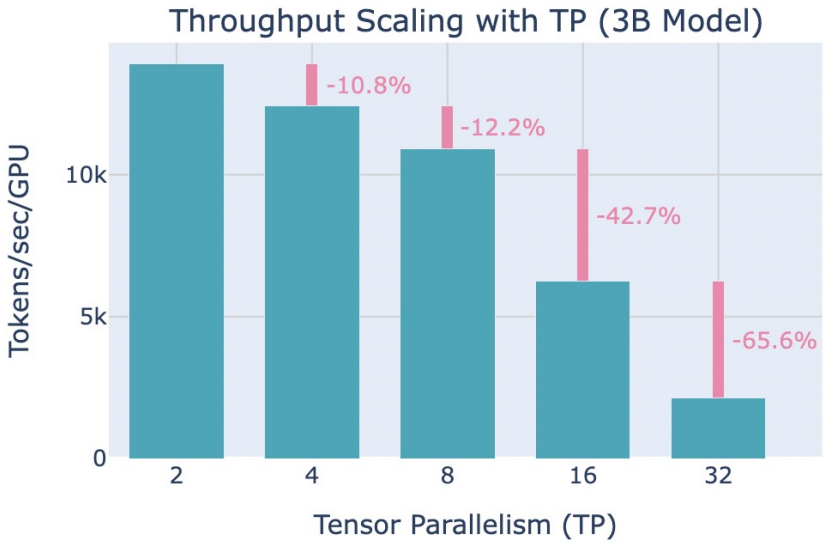


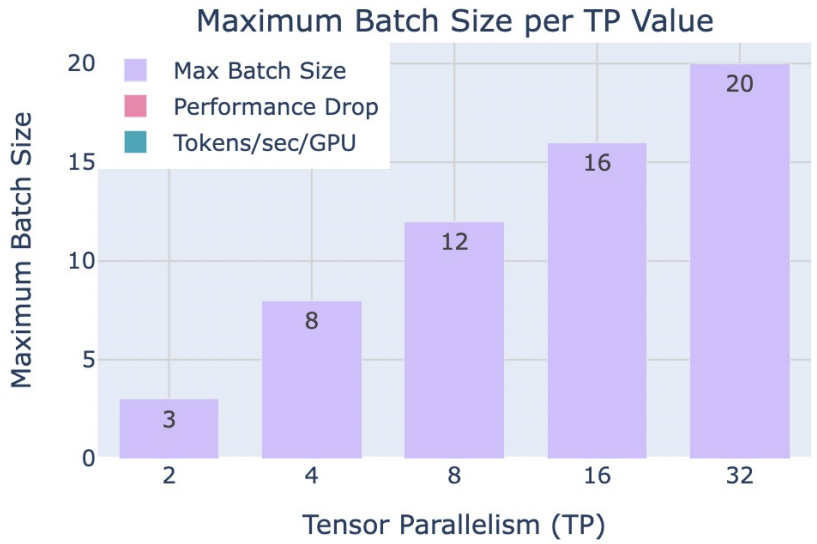
구간	GPU 연산	GPU 통신
LayerNorm	Local	-
FC_1, GeLU, FC_2 (TP Region)	Parallel	AllReduce Activs
LayerNorm	Local	-

4-2. Tensor Parallelism in a Transformer Block

C) Performance Trade-off

	TP 증가 시	설명
Throughput (속도)	감소	통신(AllReduce, AllGather) 증가
Max Batch Size	증가	GPU당 메모리 부담 감소
Activation Memory	감소	activations가 GPU 간 분산 저장
Network 부하	증가	통신 횟수·크기 늘어남



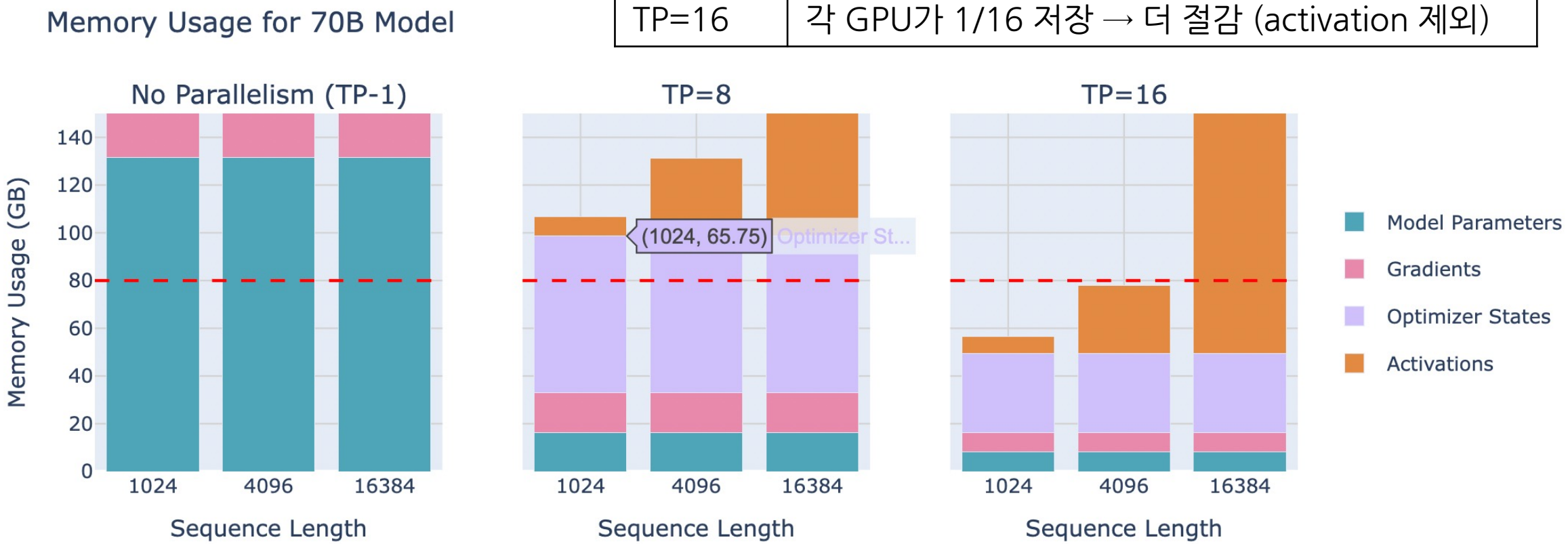


NVLink(노드 내)는 빠르지만, InfiniBand(노드 간)로 넘어가면 통신 병목 급증.

4-2. Tensor Parallelism in a Transformer Block

D) Memory 절감 효과

	메모리 효과
TP=1	모든 GPU가 full weight, grad, optimizer 저장
TP=8	각 GPU가 1/8만 저장 → 메모리 약 8배 절감
TP=16	각 GPU가 1/16 저장 → 더 절감 (activation 제외)



4-2. Tensor Parallelism in a Transformer Block

E) TP의 한계 & Sequence Parallelism의 필요성

- 하지만, TP에서 Activation은 LayerNorm, Dropout 등에 대해서 all-gather가 필요하다!
- 따라서, 완전한 절감은 아니다 => Sequence Parallelism의 필요성!

4-2. Tensor Parallelism in a Transformer Block

E) TP의 한계 & Sequence Parallelism의 필요성

- Q)왜 "**all-gather이 필요**"하다고 하나?
- A) TP는 hidden dim 을 쪼갬: 한 레이어의 중간 activation **들은 GPU별로 h/TP 만큼만** 가지고 있음
  - 예시) MLP 한 블록: 배치×토큰×히든 = (B, S, H), TP=2
    - FC1 (**Column**-parallel)
      - 각 GPU: 입력 (B,S,H) × 가중치 조각 (H, **4H/2**) → 부분 출력 (B,S, **4H/2**)
      - GeLU 등은 부분 출력에 대해 Local로 수행 가능 (문제 없음).
    - FC2 (**Row**-parallel)
      - 각 GPU: 입력 조각 (B,S, **4H/2**) × 가중치 조각 (**4H/2**, H) → 부분 합 (B,S,H)
      - 그리고 **all-reduce (합산)** 를 해서 최종 출력 (B,S,H)을 **각 GPU가 전부 갖게** 됩니다.

4-2. Tensor Parallelism in a Transformer Block

E) TP의 한계 & Sequence Parallelism의 필요성

- Q)왜 "**all-gather이 필요**"하다고 하나?
- A) TP는 hidden dim 을 쪼갬: 한 레이어의 중간 activation **들은 GPU별로 h/TP 만큼만** 가지고 있음
  - 예시) MLP 한 블록: 배치×토큰×히든 = (B, S, H), TP=2
    - 이 시점에서 **GPU마다** 다시 **전체 hidden dim**의 activation (B,S,**H**)가 생김
    - 그 다음 오는 Residual add / LayerNorm / Dropout 은 이 전체 텐서를 입력으로 사용
    - 그래서 **"중간에는 잘게 쪼개 저장했지만, 블록 경계에서 전체가 한 번 재구성"**됨!

=> 이것이 TP가 activation 메모리를 "완전히" 줄이지 못하는 이유.

4-2. Tensor Parallelism in a Transformer Block

F) TP에서 LayerNorm & Dropout 나누기 어려운 이유

- **(1) LayerNorm**
  - 각 토큰의 **hidden dim 전체(H)** 에 대해 **평균/분산**을 계산
  - Hidden dim을 TP로 쪼갠 상태라면 각 GPU는 **부분** 평균/분산만 계산할 수 있음 :(
    - 방법 A) **부분 통계를 all-reduce** 해서 전역 평균/분산을 구하고, shard 위에서 정규화(메모리는 늘지 않음).
    - 방법 B) 구현 단순화를 위해 **일시적으로 all-gather**로 전체 activation 을 모아 한 번에 LN (이때 피크 메모리↑).
  - 많은 프레임워크가 성능/구현 편의 때문에 방법 B를 사용하나, FC2의 all-reduce로 어차피 (B,S,H)가 생기는 설계라, 결과적으로 블록 경계에서 full 크기 activation이 존재

4-2. Tensor Parallelism in a Transformer Block

F) TP에서 LayerNorm & Dropout 나누기 어려운 이유

- **(2) Dropout**
  - 수학적으로는 요소별 연산이라 **shard 위에서 해도 무방**하기는 함!
  - 다만, 모든 TP rank에서 같은 값을 재현하려면, **RNG seed를 동기화해야** 하고,
    구현에 따라선 **모여진 tensor에 dropout**을 적용하기도 해서 피크 메모리가 커질 수 있습니다.

4-2. Tensor Parallelism in a Transformer Block

G) Sequence Parallelism의 등장

- (앞선 TP의 문제를 해결하기 위해) **Megatron-LM**이 쓰는 해법:
  - FC2 뒤에 바로 all-reduce(전부 보유) 대신 **reduce-scatter (sequence 축으로 분할)**

    → 각 GPU가 (B, **S/TP**, H) (**토큰 일부** + 히든 전체) 를 갖도록!

    → 그러면 LN/Dropout/Residual을 각 GPU가 **로컬로 처리** 가능 (전체 복원 불필요)
  - 다음 layer의 **FC1 들어가기 직전**에만 **all-gather(sequence 축)** 로 다시 합침
- 이렇게 하면 **블록 경계에서 전체 (B,S,H)를 오래 들고 있을 필요가 없어**,

  TP의 activation 절감 효과를 크게 살릴 수 있음!

4-2. Tensor Parallelism in a Transformer Block

G) Sequence Parallelism의 등장

- Summary:
  - TP는 **layer 내부**의 matmul 중간값들은 잘게 나눠 메모리를 아끼지만,
  - **블록 경계(Residual/LN 등)**에서 종종 전체 activation을 잠깐 복원해야함

    → 메모리를 “완전히” 줄이지는 못합니다.
  - 이를 줄이려면 **Sequence Parallelism**!

4-3. Sequence Parallelism

4-3. Sequence Parallelism

A) Sequence Parallelism의 개념

- **Sequence Parallelism (SP)**
  - TP로 분할하지 못했던 연산(**LayerNorm**, **Dropout** 등)이 존재!
  - **SP = 시퀀스** 차원(**sequence** dimension) 으로 나누어 각 GPU가 독립적으로 계산
- Summary
  - TP: **히든 차원 (hidden dim)** 을 나눔
  - SP: **시퀀스 차원 (sequence length)** 을 나눔

→ 두 방법을 결합하면, 모든 주요 연산을 GPU 여러 대에 **완전히** 분산 가능

4-3. Sequence Parallelism

B) Sequence Parallelism의 필요성

- **Tensor Parallelism (TP)**만 쓸 경우:
  - LayerNorm, Dropout 같은 연산은 **전체 hidden vector (특정 token의 모든 hidden dim)이 필요**함
  - 따라서 GPU 간에 activation 전체를 복원(**all-gather**)이 불가피함

  → 메모리 절약 효과 반감

- Sequence Parallelism(SP)는 이 문제를 해결!
  - LayerNorm과 Dropout을 **"토큰 단위 (Sequence)"**로 쪼개면?

  → 각 GPU가 자기 시퀀스 chunk만 처리할 수 있음 → full 복원 불필요

4-3. Sequence Parallelism

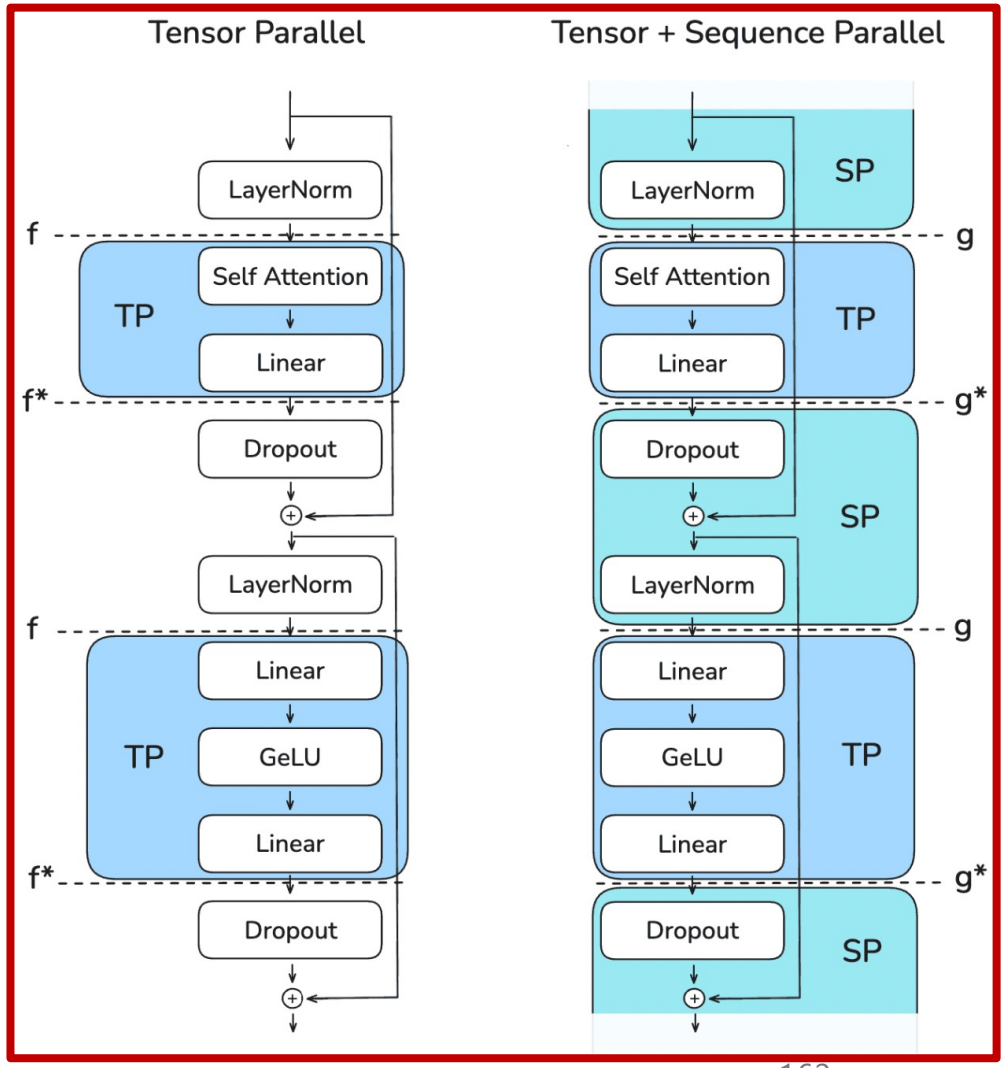
C) TP vs. TP + SP

구분	Tensor Parallel (TP)	Tensor + Sequence Parallel (TP+SP)
분할 기준	Hidden dimension **h**	Hidden **h** + Sequence **s**
LayerNorm / Dropout	전체 시퀀스 복원 필요 (**All-gather**)	시퀀스 단위로 **분리 처리 가능**
Activation Memory	b x s x **(h / tp)**	b x **(s / tp)** x **(h / tp)**
통신 연산	2 × **All-Reduce**	2 × (**All-Gather + Reduce-Scatter**) ≈ 동일 비용
통신 오버랩	일부 가능	일부 가능 (비슷한 수준)

4-3. Sequence Parallelism

C) TP vs. TP + SP

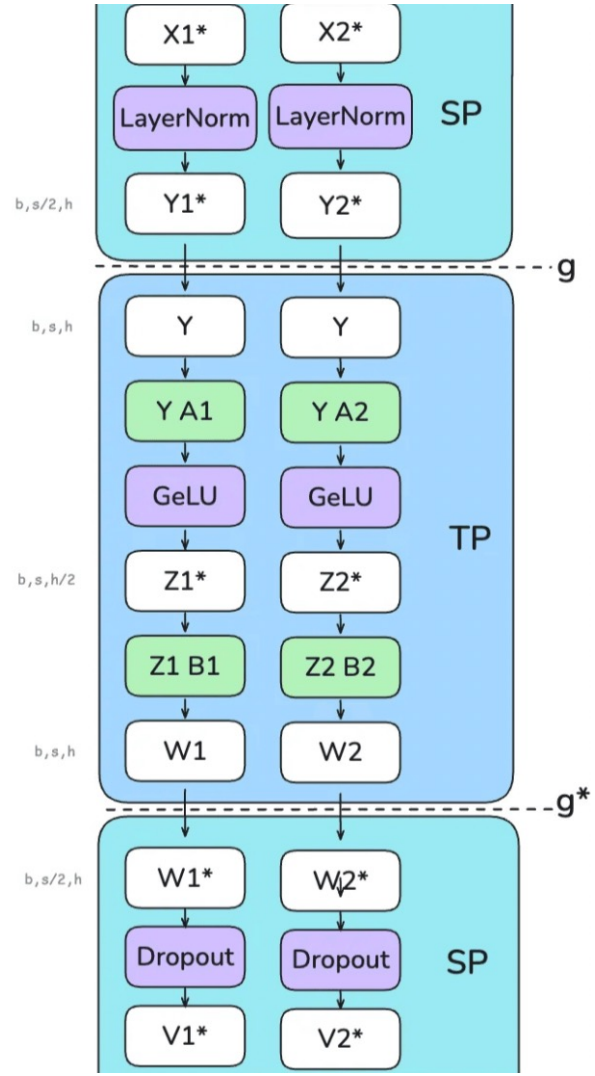
Region	TP only	TP with SP
Enter TP (column-linear)	h: sharded (`weight_out` is sharded) s: full	h: sharded (`weight_out` is sharded) s: **all-gather** to full
TP region	h: sharded s: full	h: sharded s: full
Exit TP (row-linear)	h: full (`weight_out` is full + **all-reduce** for correctness) s: full	h: full (`weight_out` is full + **reduce-scatter** for correctness) s: **reduce-scatter** to sharded
SP region	h: full s: full	h: full s: sharded



4-3. Sequence Parallelism

D) Forward Pass of SP

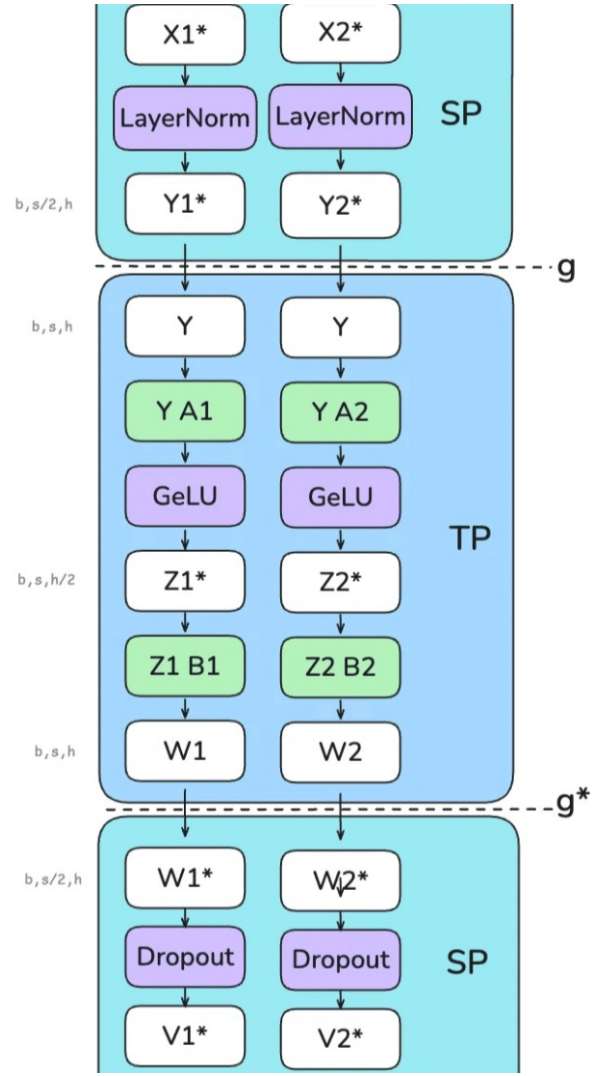
- [1] **SP** region (LayerNorm)
  - 입력 X는 sequence 차원으로 나뉨:
  - GPU_1, GPU_2 → 각 X_1* (b, **s/2**, h)
  - 각 GPU가 자기 chunk에 대해 LN→ Y_1*, Y_2*.
- [2] **SP** → **TP** 전환 (g 연산)
  - All-Gather 수행 → Y = concat(Y_1*, Y_2*) = (b, s, h).
  - 이제 TP 구간(Linear, GeLU 등)에서는 full sequence 가 필요하므로 복원.



4-3. Sequence Parallelism

D) Forward Pass of SP

- [3] **TP** region (MLP, Attention 등)
  - Hidden dimension 기준으로 분할 (TP standard)
  - FC_1: column parallel → GeLU → FC_2: row parallel
  - FC_2 끝에서 Reduce-Scatter로 TP 구간 마무리.
- [4] **TP** → **SP** 전환 (g* 연산)
  - Reduce-Scatter 수행 → W_1*, W_2* = (b, **s/2**, h)
  - SP region으로 돌아가 Dropout, Residual, LayerNorm 수행.



4-3. Sequence Parallelism

E) Backward Pass of SP

- Forward의 conjugate operation을 수행
  - 이 때문에 f / f* 와 g / g* 를 **“conjugate pair”**라고 부름

→ Forward에서 No-op이면, Backward에서는 All-reduce (반대 역할).

구간	Forward 연산	Backward 연산
SP region	g (All-Gather)	g* (Reduce-Scatter)
TP region	g* (Reduce-Scatter)	g (All-Gather)

4-3. Sequence Parallelism

F) Memory 절감

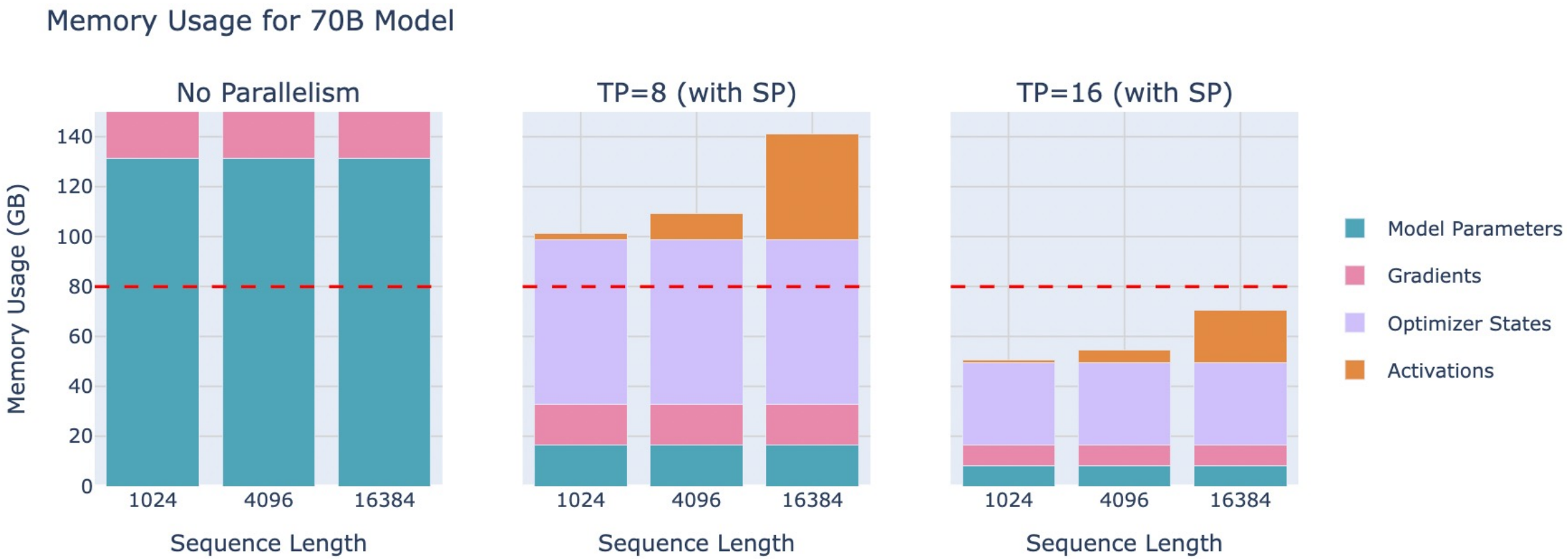
- [1] **Activation Memory 감소**
  - TP: (b, s, h) 전체 activation을 저장해야 함.
  - TP+SP: 각 GPU가 **(b, s/TP, h)** 또는 **(b, s, h/TP)** 일부만 보유.

  → 피크 메모리 ≈ 1/TP 수준으로 감소.

  → LayerNorm / Dropout까지 분산 처리 가능.
- [2] **Throughput Trade-off**
  - TP와 **통신량은 동일** 수준이지만, **통신 횟수(2× 더 많음)** 때문에 통신 지연(latency) 는 약간 증가.
  - 일반적으로 TP ≤ 8 (node 내부) 수준에서는 효율 유지.
  - TP > 8 (cross-node) 부터 급격히 속도 저하 발생.

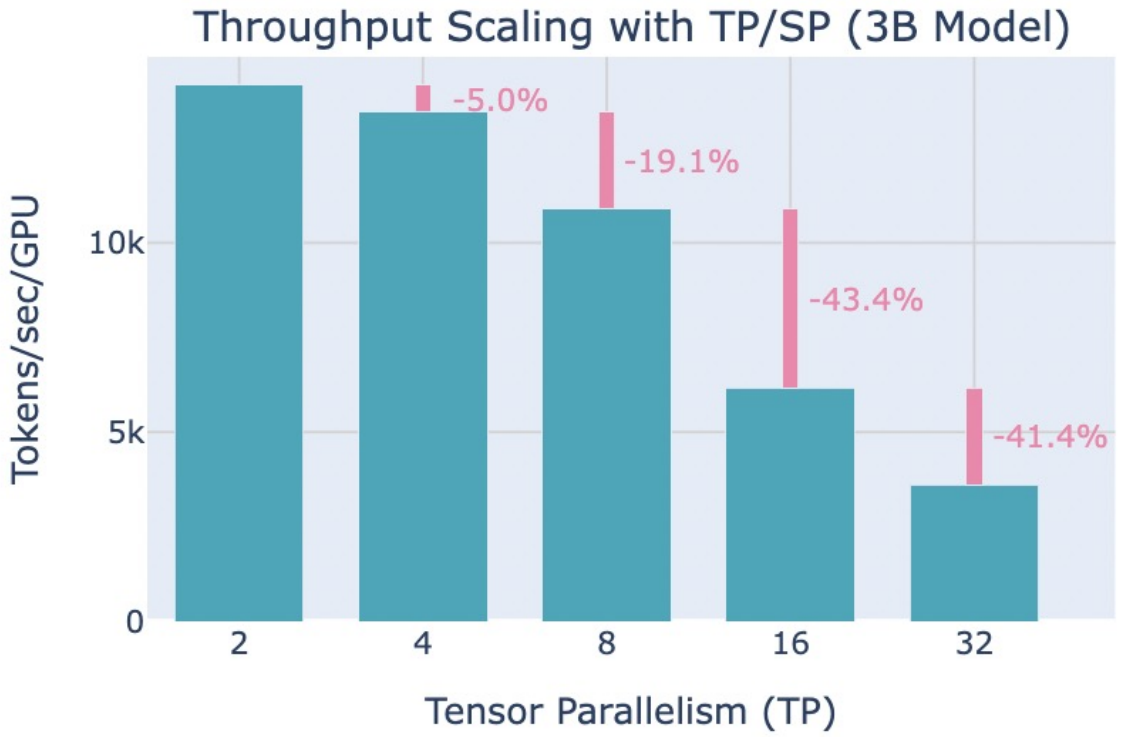
4-3. Sequence Parallelism

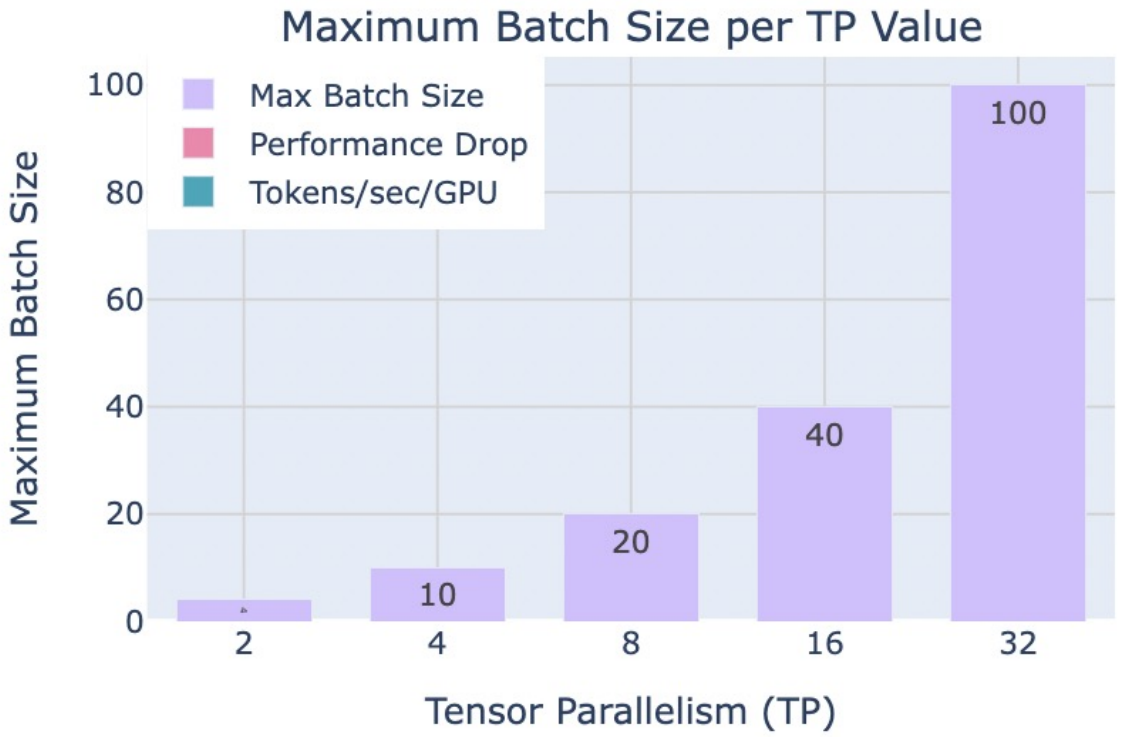
F) Memory 절감



4-3. Sequence Parallelism

F) Memory 절감





4-3. Sequence Parallelism

F) Memory 절감

항목	TP=8	TP=8 + SP	TP=16 + SP
Throughput	baseline	-5~10%	-40%↓
Max Sequence (70B 모델)	8k	16k	16k
GPU 메모리 (활성화 포함)	초과	80GB 이내	80GB 이내

동일한 GPU 메모리 내에서도 **sequence length를 2배 이상** 늘릴 수 있음.

4-3. Sequence Parallelism

G) Summary

- Tensor Parallelism(TP)의 보완 기술로, **LayerNorm·Dropout** 등의 연산을 **sequence 차원**으로 분산(shard) 하여 activation memory를 추가로 줄이고 더 긴 시퀀스(batch)를 처리할 수 있게 함
- 통신 비용은 동일하지만 횟수가 늘어나기 때문에,
- TP+SP는 보통 노드 내부(≤8GPU) 에서 가장 효율적

4-3. Sequence Parallelism

H) TP+SP의 한계점

- (1) **시퀀스 길이 (sequence length)가 너무 길어지면** 여전히 activation 폭발
  - 만약 token들 간에 독립적이지 않고, 서로 **연관**이 있다면? (feat. Attention)
  - SP가 LayerNorm, Dropout 등의 activation을 분할해 주긴 하지만,
  - TP region (Linear, GeLU, Attention) 의 activation은 여전히 히든 차원 기준으로 shard 되어 있음 (i.e., TP 구간 내부의 activation은 여전히 **sequence full 상태에서 계산돼야** 함)

  → 즉, O(b x s x h) 항목 중 **s(시퀀스) 가 커지는 만큼 activation memory도 그대로 커짐**

  → 긴 문장(예: 32K, 64K 토큰)을 다룰 땐 activation memory가 다시 폭발(OOM)!

4-3. Sequence Parallelism

H) TP+SP의 한계점

- (2) **TP=8 이상으로** 스케일링 시 **massive slowdown (통신 병목)**
  - TP는 주로 **노드 내** 병렬화에 최적화되어 있음
    - 예: **1 노드 = 8 GPU** → TP=8까지는 NVLink로 빠르게 통신 가능
  - 하지만 TP=16, 32로 넘어가면 **노드 간 통신(InfiniBand, EFA)** 이 필요

→ 통신 지연(latency)이 급증,

→ Throughput 급감 (-40~60%)

→ GPU가 계산보다 통신을 기다리는 시간이 더 김

모델을 너무 크게 만들거나,
여러 노드에 걸쳐 TP를 확장하려 하면
속도는 떨어지고 이득은 줄어든다!

4-3. Sequence Parallelism

I) CP & PP의 등장 배경

문제	원인	해결책
(1) **긴 시퀀스**로 인한 activation 폭발	TP region에서 **sequence가 full** 상태	**Context** Parallelism (CP)
(2) TP 확장 시 **속도 저하**	노드 간 **통신 병목**	**Pipeline** Parallelism (PP)

CP: 시퀀스 전체를 한 GPU에 두지 않고, **context(문맥 단위) 로 나누어** attention 계산을 분산.

5. Context Parallelism

5-1. Overview

5-1. Overview

A) CP & PP의 등장 배경

- **(1) Context Parallelism (CP)**
  - SP와의 공통점/차이점
    - (공통점) Sequence 전체를 한 GPU에 두지 않고, **context(문맥 단위) 로 나누어** attention 계산을 분산.
    - (차이점) Sequence를 여러 GPU에 나눠 **attention**도 분할 수행 → 다만, 독립적 X이므로 **정보교환 필요**
  - 각 GPU가 **자기 구간의 token**만 처리하되, 필요한 **attention context** 정보만 주고받음
    - **Ring** Attention, **Flash** Attention 기반
  - 이걸 통해 SP가 처리하지 못한 긴 sequence 문제를 해결

    → 긴 문장(예: 32K~128K token)도 메모리 폭발 없이 처리 가능

5-1. Overview

A) CP & PP의 등장 배경

- **(2) Pipeline Parallelism (PP)**
  - 모델의 **layer 단위**로 GPU를 나눔
    - 예: 96-layer 모델 → GPU **8개** x layer **12개**씩
  - 각 GPU가 모델 일부만 담당하므로, 단일 노드에 너무 많은 weight가 몰리지 않음
  - 메모리와 연산량을 layer 단위로 분산
  - **ZeRO/TP/SP**와 동시에 사용 가능
  - PP는 (TP가 감당하지 못할) **거대한 모델** 크기 문제를 해결

5-1. Overview

B) Sequence Parallelism vs. Context Parallelism

- CP vs. SP: **"무엇을 나누는가"** 와 **"왜 나누는가"** 에서 다름
- 공통점

항목	공통점
공통 목적	**Sequence length**가 길 때 GPU 메모리 부담을 줄이기 위함
사용 시점	**TP로도 감당 안 되는** 긴 sequence에서!
방식	**Sequence** 차원을 여러 GPU에 나누어 병렬 처리
차이점 요약	- SP는 "**작은** 연산 최적화(LayerNorm·Dropout)"에 초점 - CP는 "**큰** 연산(Attention) 분할"에 초점

5-1. Overview

B) Sequence Parallelism vs. Context Parallelism

구분	Sequence Parallelism (SP)	Context Parallelism (CP)
분할 축	**Sequence** dim	**Sequence** dim: 세부적으로는 **"attention context"** 기준
적용 위치	**LayerNorm, Dropout** 등 "TP 영역 밖의 연산"	**Self-Attention** 내부 연산 ($QK^T \cdot V$ 계산)
통신 패턴	**All-gather + Reduce-scatter** (sequence 전환용)	**Ring Attention** 등을 사용하여 cross-GPU context 전달
활용 사례	Megatron-LM, GPT-NeoX (TP와 결합)	Long Context LLMs (Mistral, DeepSeek, Llama3 Long Context)
중요 포인트	Sequence를 GPU 간 "분할하여 독립 연산"	Sequence를 "연속 context로 나누되, **GPU 간 attention 공유**"
해결 문제	TP로 못 분할하던 LayerNorm/Dropout의 메모리	8K·32K·128K 같은 초장문 context에서 OOM 문제

5-1. Overview

B) Sequence Parallelism vs. Context Parallelism

- Summary
  - [TP] **hidden** dim 분할 → MLP, Attention 내부 연산 분산
  - [SP] **seq** dim 분할 → LN, Dropout 등 경량 연산 분산
  - [CP] **seq** dim 중 **context** dim 분할 → Attention 문맥 전체를 여러 GPU로 분산

5-1. Overview

B) Sequence Parallelism vs. Context Parallelism

- Summary
  - [TP] **hidden** dim 분할 → MLP, Attention 내부 연산 분산
  - [SP] **seq** dim 분할 → LN, Dropout 등 경량 연산 분산
  - [CP] **seq** dim 중 **context** dim 분할 → Attention 문맥 전체를 여러 GPU로 분산

5-1. Overview

C) KV 교환

- 여기서 Q(query) 는 각 토큰별로 계산되고, 각 query는 **"모든" K(key) 와 V(value) 를 참조해야** 합니다.
- 문제점: CP에서는 sequence를 GPU마다 나누니까,

GPU	담당 시퀀스	보유 Key/Value
GPU1	토큰 0~4K	K_1, V_1
GPU2	토큰 4K~8K	K_2, V_2
GPU3	토큰 8K~12K	K_3, V_3
$GPU4_4$	토큰 12K~16K	K_4, V_4

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$

- 이때 GPU1의 query는 GPU2~ GPU4 의 key/value도 참조해야! → **GPU 간 Key/Value 교환(통신) 필요!**

5-1. Overview

D) Ring Attention

- 필요성: **GPU 간 Key/Value 교환(통신) 필요!**
- Ring Attention은 **CP의 핵심**
  - GPU1 → GPU2 → GPU3 → GPU4 → GPU1 형태의 **ring 구조**로 K/V를 순차적으로 교환
  - **각 GPU**는 자기 **local attention**을 수행하
  - 다음 GPU에 필요한 **KV를 전달**
- 위와 같이 “**부분 Attention 결과를 순차적으로 쌓는 방식**” 으로 전체 Attention을 완성

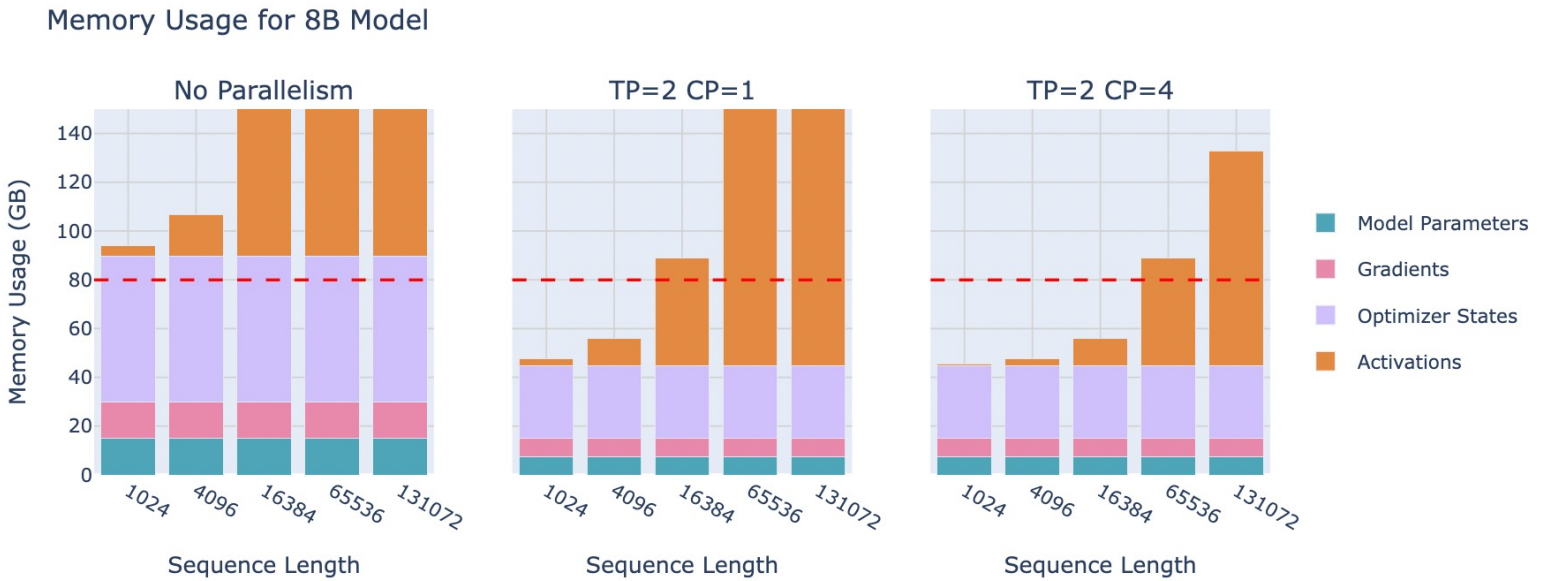
5-1. Overview

D) Ring Attention

- KV 전체를 all-gather하지 않음

  → 통신량 대폭 감소

- 메모리 효율적으로 긴. seq 처리 가능

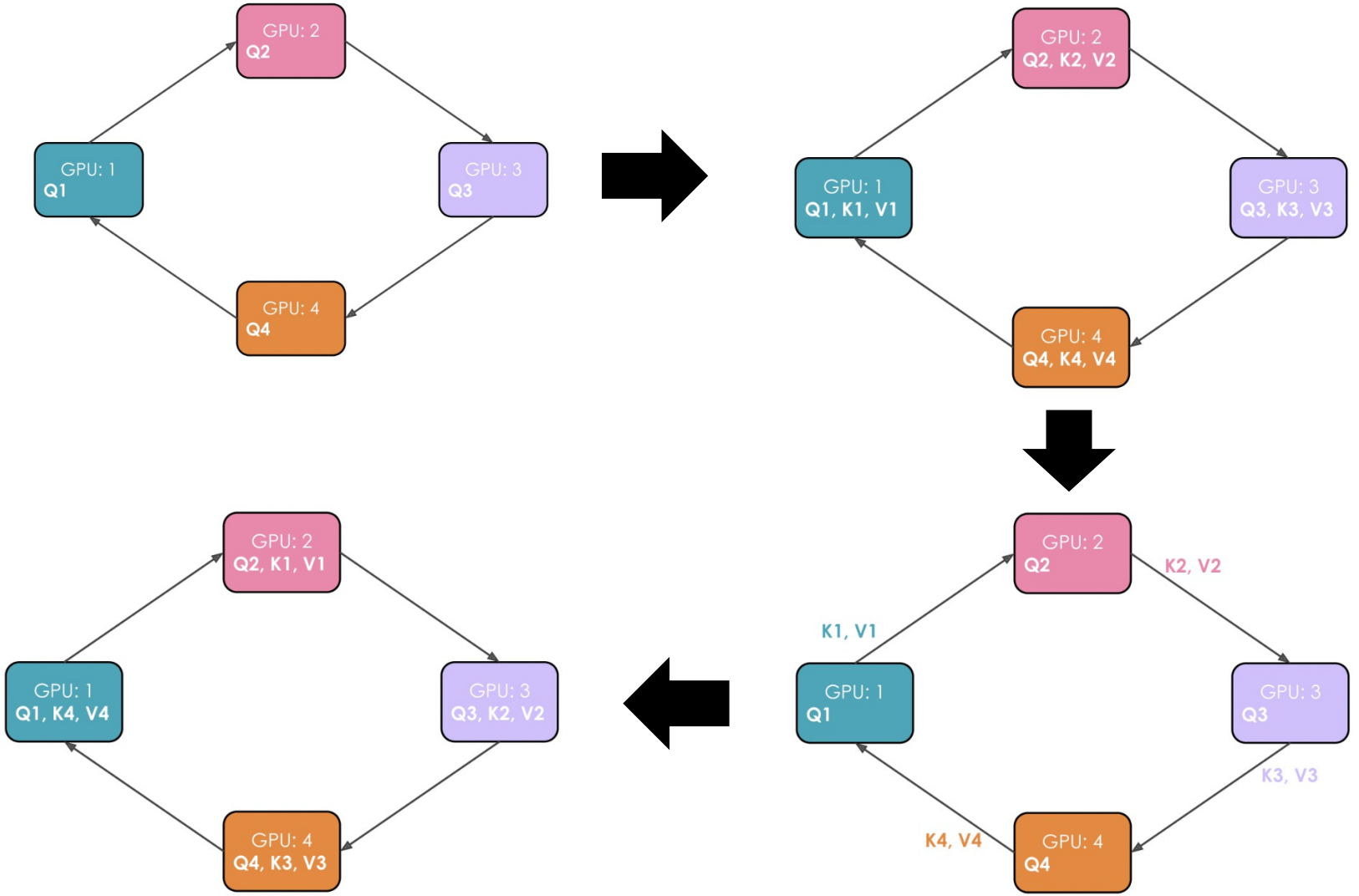


구성	설명
No Parallelism	한 GPU가 모든 토큰(예: 131K)을 전부 처리 → activation 메모리 폭발 (orange 영역 ↑↑)
TP=2, CP=1	Hidden dimension은 TP로 분할, 시퀀스는 분할하지 않음 → TP로 weight 관련 메모리 줄었지만 activation은 여전히 커짐
TP=2, CP=4	시퀀스 dimension까지 GPU 4개에 분산 → activation 메모리(orange)가 현저히 감소

5-2. Ring Attention

5-2. Ring Attention

A) Overview



5-2. Ring Attention

B) Procedure

- 각 GPU는 동시에 아래의 세 작업을 수행함

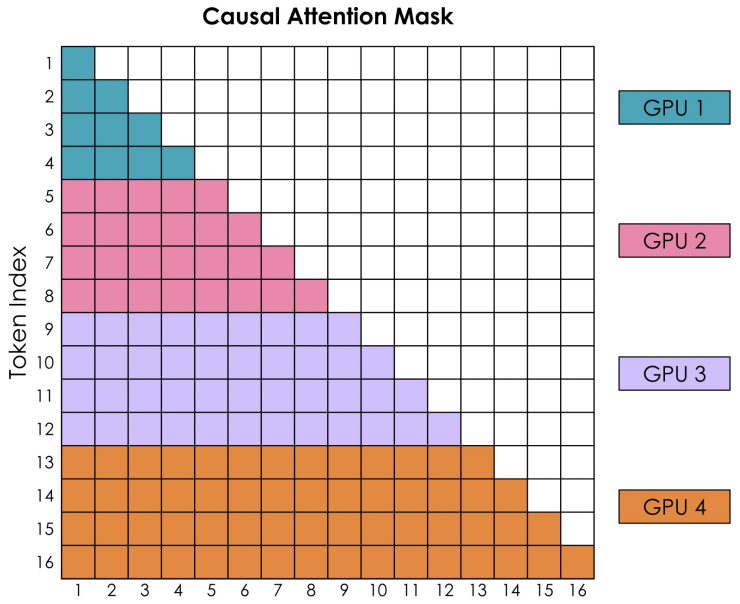
단계	동작	설명
(1) **Send**	자신의 K, V를 다음 GPU로 전송 (비동기)	예: $GPU_1 \rightarrow GPU_2$
(2) **Compute**	로컬 K, V에 대해 $*QK^T / \sqrt{d} \rightarrow$ Softmax $\rightarrow$ V 계산	예: GPU_1은 $Q_1 \times K_1^T$
(3) **Receive**	이전 GPU로부터 K, V 수신	예: $GPU_1 \leftarrow GPU_4$ (K_4, V_4)

5-2. Ring Attention

C) Causal Attention Mask & Load Imbalance

- Causal Mask는 "**앞의 토큰만 attend 가능**"하도록 하는 triangular mask
- 이 마스크 때문에 GPU들이 계산량이 서로 다르게 됨

GPU	담당 토큰	Attention 대상
GPU_1 (토큰 1-4)	자기 이전 토큰 없음	바로 계산 시작 가능
GPU_2 (토큰 5-8)	토큰 1-8 필요	GPU_1의 K_1,V_1 받아야 함
GPU_3 (토큰 9-12)	토큰 1-12 필요	GPU_1,K_2 모두 기다려야 함
GPU_4 (토큰 13-16)	토큰 1-16 필요	가장 오래 기다림



그래서 GPU1은 계산을 시작하고 빨리 끝나지만, **GPU4는 모든 다른 GPU의 데이터를 기다려야**!

→ 이게 바로 **"부하 불균형(load imbalance)"** 문제!

5-2. Ring Attention

D) Load Imbalance의 해결

- 개선 방향: **Load Balancing** Ring Attention
  - GPU별 attention **계산 순서 재배치** 또는 **균등한 work chunk** 분배 (balanced ring scheduling)
- 예시)
  - GPU_1이 끝나고 idle 상태일 때 GPU_2 계산을 일부 분담하거나
  - Q, K, V 블록 단위를 더 잘게 쪼개서 동시에 여러 GPU가 작업 가능하게 하는 식.
- 이렇게 하면 각 GPU가 동시에 일정량의 attention을 계산

→ 전체 GPU 활용률이 높아지고 & 긴 시퀀스에서도 throughput 저하를 최소화

5-2. Ring Attention

E) Summary

- CP의 핵심으로, sequence를 GPU별로 나누어 K/V를 순차적으로 교환하며 attention을 계산
- 하지만 causal mask로 인해 GPU별 계산량이 불균형하게 됨
- 이에 따라, 후속 연구에서는 이를 균형화(load balancing) 하는 기법들이 추가로 도입

항목	설명
핵심 아이디어	K/V를 ring 형태로 순차적으로 전달하면서 attention을 계산
장점	모든 GPU가 전체 시퀀스에 대해 attention 계산 가능 (memory 분산)
통신 방식	Send/Receive (비동기), 순차적 교환
문제점	causal mask 때문에 GPU별 계산량 불균형 발생
해결방향	balanced scheduling, overlapped compute-comm 방식

5-3. Zig-Zag Ring Attention

5-3. Zig-Zag Ring Attention

A) Review: 기존 Ring Attention의 한계점

- 각 GPU가 시퀀스의 연속된 부분(예: GPU1=토큰1-4, GPU2=토큰5-8 …)을 담당
- Causal Mask(자기 이전 토큰만 참조) 때문에 문제가 생김

  → 후반 GPU일수록 계산량이 폭증

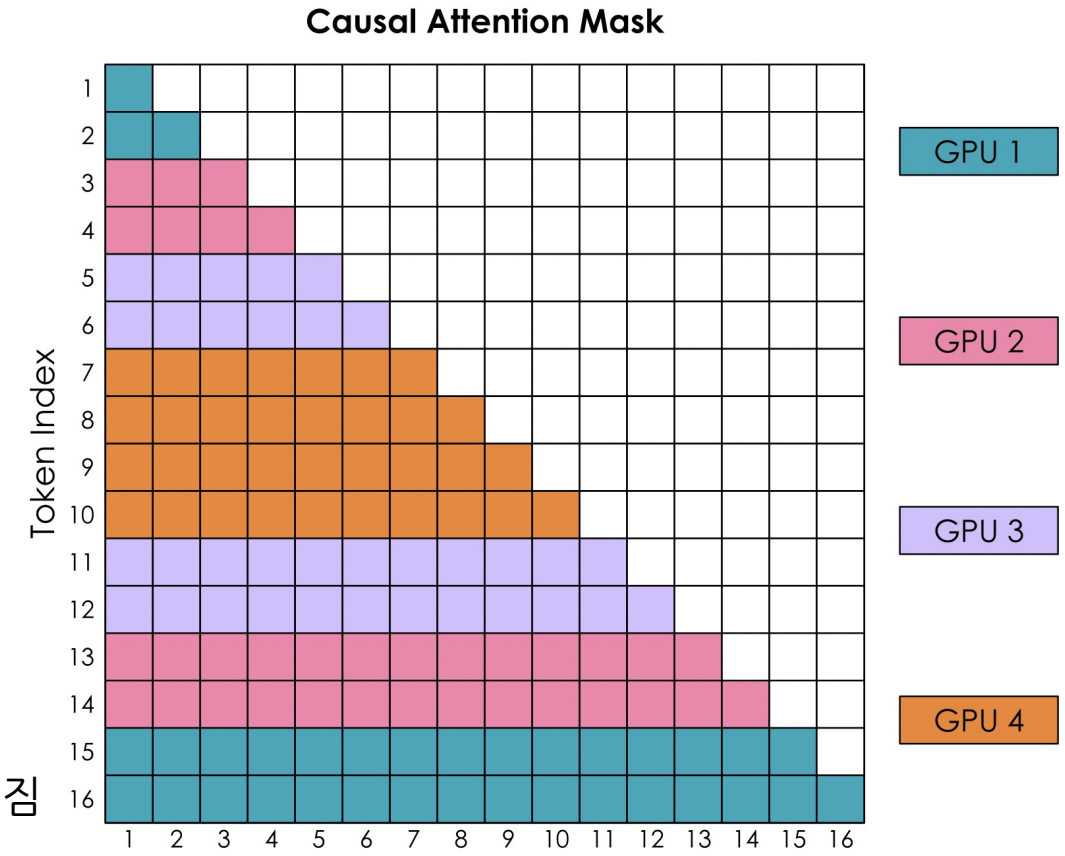
  → GPU1은 금방 끝나고 놀고 있고, GPU4는 계속 계산 중

  → GPU 부하 불균형(load imbalance) 발생.

5-3. Zig-Zag Ring Attention

B) Overview

- Zig-Zag Ring Attention = **균형 잡힌 Context Parallelism**
- 기존 Ring Attention의 "**GPU 부하 불균형**" 문제를 해결
- 해결책: **Zig-Zag 분할** (균형 잡힌 시퀀스 배치)
  - 단순히 토큰을 순서대로 GPU에 배정하지 말고,
  - **"초반 + 후반 토큰"**이 섞이도록 분할하자!



Causal mask를 보면, 각 GPU는 "앞/뒤 토큰"이 섞여 있어서 계산량이 균등해짐

5-3. Zig-Zag Ring Attention

C) 통신방법 비교

- (a) **All-Gather** 방식
  - 모든 GPU가 한 번에 **모든 K/V 데이터**를 모음.
  - 통신은 단 1회 (**All-Gather 1번**)
  - 계산은 순차적: Attn(Qi, Ki, Vi) → Attn(Qi, Ki+1, Vi+1) → …
  - 장) **간단** <-> 단) GPU마다 K/V 전체를 복사해야 하므로 **임시 메모리 폭증**

5-3. Zig-Zag Ring Attention

C) 통신방법 비교

- (b) **All-to-All (Ring)** 방식
  - GPU끼리 **필요한 K/V만** 교환(fetch) (send/recv 1-hop)
  - 각 단계에서 통신과 계산이 **겹침(Overlap)**: Fetch (K_{i+1}, V_{i+1}) 하면서 Attn(Q_i, K_i, V_i) 계산.
  - 장) 메모리 효율적 (한 번에 한 chunk만 보유)
  - 단) 통신 단계가 여러 번 → base latency 약간 증가

5-3. Zig-Zag Ring Attention

C) 통신방법 비교

- Summary
  - All-Gather: 빠름 + 메모리 많이 듬
  - All-to-All (Ring): 느림 + 메모리 적게 듬 (더 scalable)

구분	All-Gather	Ring (All-to-All)
통신 횟수	1회 (모두 모음)	**여러 번** (1-hop씩 순환)
통신 동작	집단 통신 (집합적)	**P2P** 통신 (이웃끼리)
메모리 사용량	높음 (K/V 전체 보관)	**낮음** (K/V 조각만 보관)
통신-계산 오버랩	어려움	**자연스러움**
Latency	낮음	다단계라 약간 높음
적합한 상황	GPU 메모리 넉넉할 때	긴 시퀀스, 메모리 제약 심할 때

5-3. Zig-Zag Ring Attention

D) 최종 효과

- **Context Parallelism**에서의 **GPU 부하 불균형**을 해결하기 위해

시퀀스 토큰을 "**앞/뒤가 섞이도록**" 분배(Zig-Zag)하여 **계산량을 균등화**하고, **Ring(All-to-All)** 통신으로

Key/Value를 **단계적으로 교환**해 GPU 간 계산 부하와 메모리 사용량을 모두 최적화한 Attention 구현 방식

Zig-Zag Ring Attention = 부하 균형 (load-balanced) + 메모리 효율 (memory-efficient) Attention

문제	해결 방식	효과
GPU 부하 불균형	토큰 순서를 섞어(Zig-Zag) 앞뒤 섞인 분할	모든 GPU 계산량 균등
GPU 메모리 폭발	K/V를 순차적으로 교환(Ring)	임시 메모리 감소
통신 오버헤드	계산-통신 오버랩	GPU idle 시간 감소

6. Pipeline Parallelism

6-1. Overview

6-1. Overview

A) CP & PP의 등장 배경

- **(1) Context Parallelism (CP)**
  - SP와의 공통점/차이점
    - (공통점) Sequence 전체를 한 GPU에 두지 않고, **context(문맥 단위) 로 나누어** attention 계산을 분산.
    - (차이점) Sequence를 여러 GPU에 나눠 **attention**도 분할 수행 → 다만, 독립적 X이므로 **정보교환 필요**
  - 각 GPU가 **자기 구간의 token**만 처리하되, 필요한 **attention context** 정보만 주고받음
    - **Ring** Attention, **Flash** Attention 기반
  - 이걸 통해 SP가 처리하지 못한 긴 sequence 문제를 해결

    → 긴 문장(예: 32K~128K token)도 메모리 폭발 없이 처리 가능

6-1. Overview

A) CP & PP의 등장 배경

- **(2) Pipeline Parallelism (PP)**
  - 모델의 **layer 단위**로 GPU를 나눔
    - 예: 96-layer 모델 → GPU **8개** x layer **12개**씩
  - 각 GPU가 모델 일부만 담당하므로, 단일 노드에 너무 많은 weight가 몰리지 않음
  - 메모리와 연산량을 layer 단위로 분산
  - **ZeRO/TP/SP**와 동시에 사용 가능
  - PP는 (TP가 감당하지 못할) **거대한 모델** 크기 문제를 해결

6-1. Overview

B) PP의 필요성

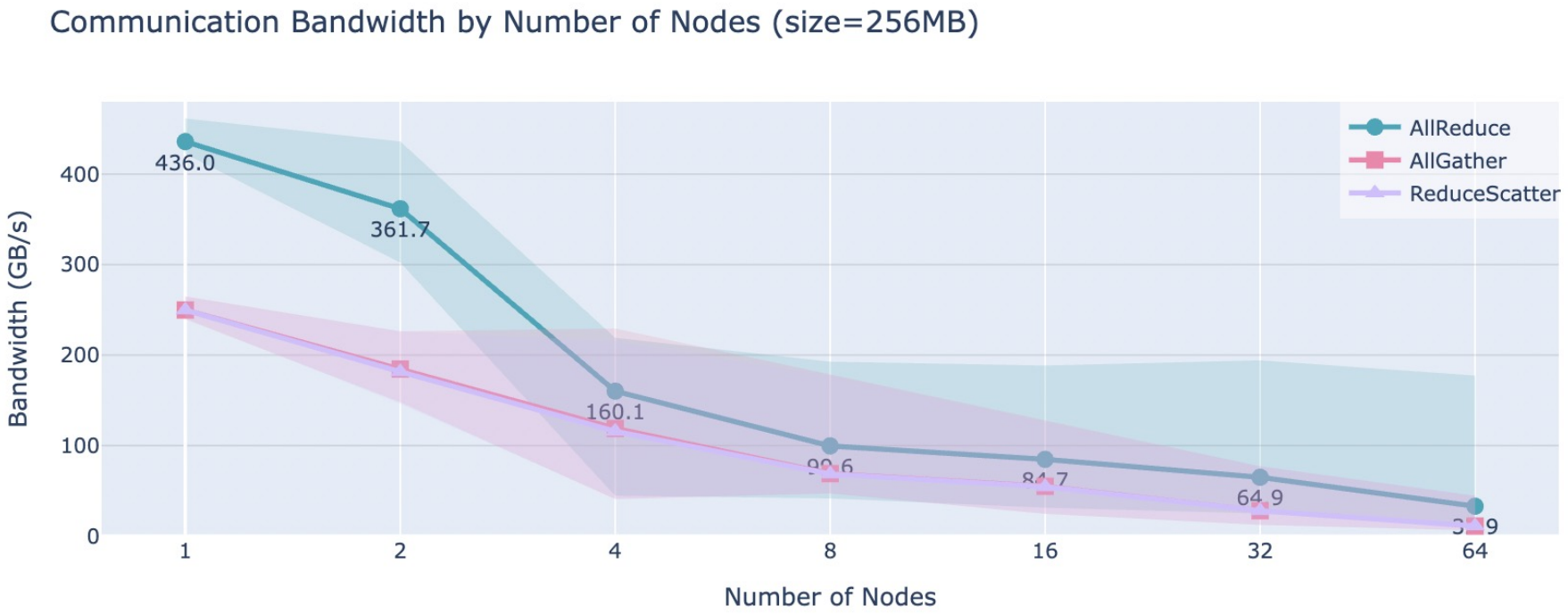
- 이전까지 배운 Parallelism
  - **Tensor** Parallelism (TP): Layer 내부(**hidden dimension**) 단위 분할
  - **Sequence** / **Context** Parallelism (SP, CP): 시퀀스(**sequence**) 단위 분할
- 위 세 방법론들은, 모두 "한 노드(**8 GPU 정도**) 안에서는 효과적"

  → But **노드를 넘어가면** 통신 병목 때문에 효율이 급격히 떨어짐!!

6-1. Overview

B) PP의 필요성

노드 수	AllReduce 대역폭 (GB/s)	설명
1	436 GB/s	NVLink (노드 내) — 매우 빠름
4	160 GB/s	노드 간 EFA 통신 시작 — 급락
8	90 GB/s	네트워크 대역폭 한계
64	39 GB/s	거의 1/10 수준으로 감소



즉, TP를 **여러 노드로 확장**하면 **통신이 병목**이 되어 성능이 붕괴

→ 그래서 **모델 크기가 커질수록 TP만으로는 부족해짐!**

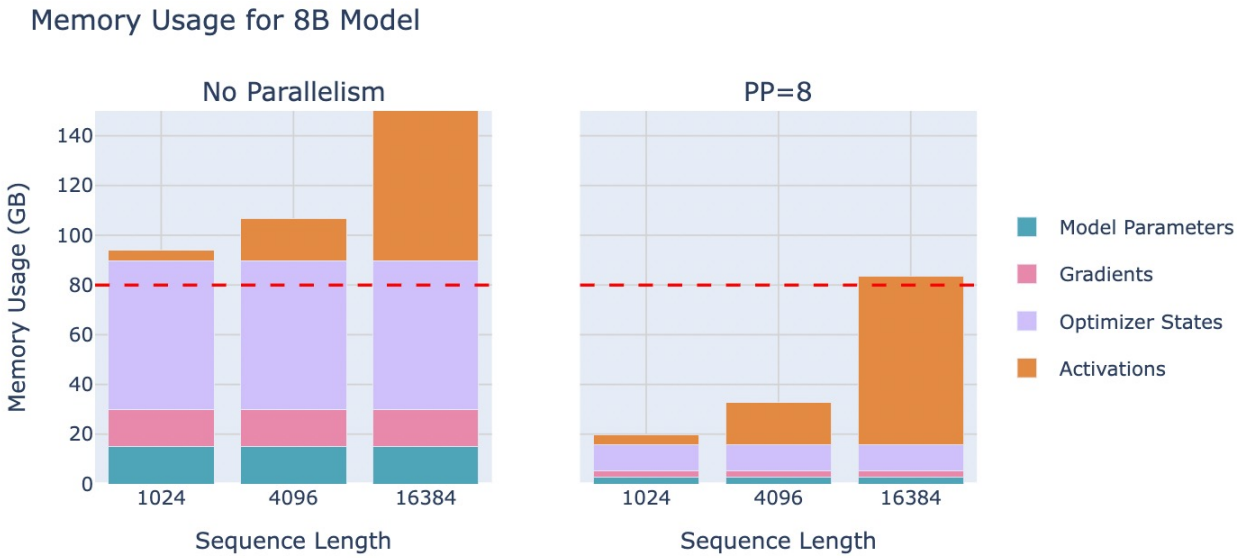
6-1. Overview

C) PP의 핵심 아이디어

- **"모델의 Layer를 여러 GPU에 나눠서 처리하자."**
- 모델에 layer가 32개 있다고 하면:
  - GPU_1 → layer 1-4 ... GPU_8 → layer 29-32

→ 각 GPU는 모델 파라미터의 일부분만 보유

→ 따라서 GPU당 메모리 사용량이 크게 감소



항목	No Parallelism	PP=8
모델 파라미터	한 GPU에 전체 weight 보관	**1/8**로 분할 → 메모리 절감
Optim. / Gradient	동일	**1/8**로 분할
Activation	동일	**변화 없음**

6-1. Overview

D) Activation memory는 줄지 않는 이유

- PP에서는 **Forward와 Backward가 겹치지 X**
  - Forward 전체를 다 돌린 뒤에야, Backward를 시작하기 때문
- Ex) PP with 8 GPU:
  - Forward는 8개의 micro-batch를 **순서대로** GPU를 통과해야 함.
  - 첫 Backward는 Forward가 **모두 끝나야** 가능.
- 따라서 각 GPU는 "**PP 단계 수** × **(activation / PP 단계 수)** ≈ activation 전체"를 저장해야!
  - 비록 layer는 나눴지만, 각 GPU는 여전히 전체 시퀀스의 activation을 저장해야!

6-1. Overview

E) 통신 패턴: Pipeline 구조

- PP에서는 ZeRO처럼 weight를 통신하지 않고, 대신 **"activation을 순차적으로 전달"**
  - Forward 시: 각 GPU가 다음 GPU로 **activation**을 넘김
  - Backward 시: 반대로 **activation gradient**를 이전 GPU로 전달
- 이런 식으로 "pipeline처럼" 데이터(=activation)를 흘려보내서 전체 모델을 학습

구분	Data Parallel / ZeRO	Pipeline Parallel
통신 단위	Parameters / Gradients	**Activations**
통신 패턴	All-Reduce	Send / Receive (1-hop)
병목 원인	Gradient sync	Forward/Backward 간 dependency

6-1. Overview

F) Summary

항목	장점	단점
Weight 메모리	**1/PP** 감소	-
Activation 메모리	-	**Forward 전체**를 버퍼링해야
통신량	**Weight** 통신 X	**Activation** 송수신 O
확장성 (multi-node)	TP보다 유리	(Stage 간 의존성으로) latency 존재
Throughput	큰 모델에 유리	Pipeline “bubble” 발생 가능

PP = 모델의 **layer**를 여러 GPU로 분할하여 **parameter 메모리 부담**을 줄임

→ 단점: 하지만 **activation 메모리는 그대로**이며, GPU 간 activation을 **순차적**으로 전달해야!

→ 문제점: 통신 오버헤드와 파이프라인 **“bubble”**이 생길 수 있음

6-1. Overview

F) Summary

- DP vs. TP vs. SP/CP vs. PP

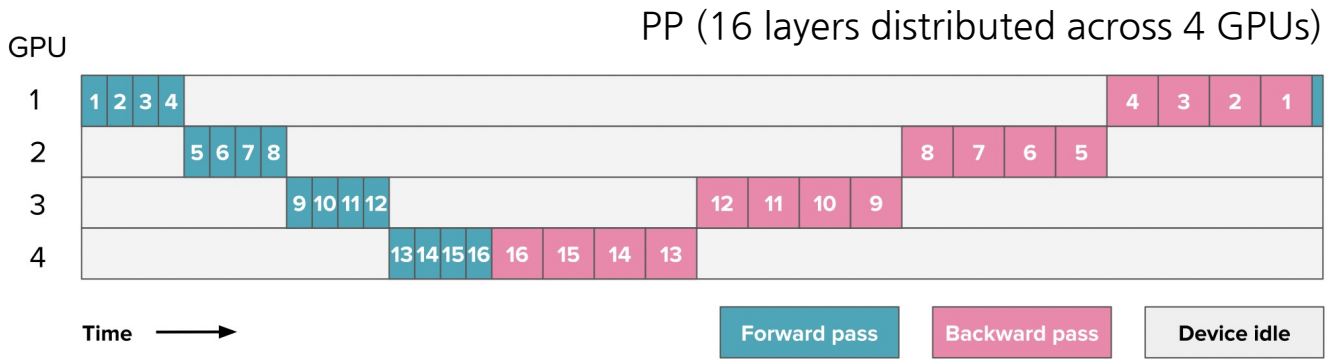
	병렬화 기법	분할 대상	주요 효과	한계
Data Parallel (DP)		Data 배치(batch)	간단, scale-up 용이	메모리 중복
Model Parallel	**Tensor** Parallel (TP)	Layer 내부 (hidden dim)	Layer 계산 분산	노드 간 확장 시 통신 병목
	Sequence/Context Parallel (SP/CP)	Sequence dimension	Activation 절감	Attention 통신 오버헤드
	Pipeline Parallel (PP)	Layer (depth)	모델 파라미터 메모리 절감	Activation 동일, 파이프라인 지연 존재

6-2. Splitting layers: All forward, All backward

6-2. Splitting layers: All forward, All backward

A) Forward Pass 개요

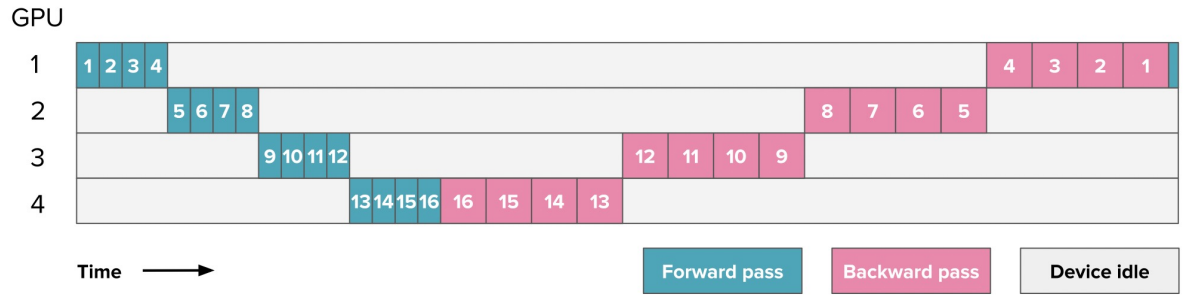
- PP의 기본 아이디어:
  - 모델의 **layer**들을 여러 GPU에 나누어 **"순서대로"** 연결
  - Batch를 **순차적으로 전달**하며 Forward + Backward 수행
- **All-Forward All-Backward** (**AFAB**) 스케줄: PP의 **가장 단순한** 스케줄링 방식
- 아래 그림: **Forward Pass** 스케줄



(숫자: Layer의 index)

- 파란색: Forward pass
- 분홍색: Backward pass
- 회색: idle 상태 (bubble)

6-2. Splitting layers: All forward, All backward



B) Pipeline Bubble

Procedure

- Step 1) **GPU 1**이 먼저 레이어 1~4를 통해 micro-batch 1을 처리
- Step 2) 결과(Activation)를 **GPU 2**로 전달
- Step 3) **GPU 2** 는 micro-batch 1의 Forward를 진행하는 동안, **GPU 1**은 micro-batch 2를 시작

- 이런 식으로 데이터가 **“pipeline”**처럼 흐름

→ 하지만, Backward는 Forward가 전부 끝난 후에만 시작!

→ 따라서, 후반부 GPU들이 오랫동안 idle 상태 == **“Pipeline bubble”**

6-2. Splitting layers: All forward, All backward

C) Pipeline Bubble의 수식적 이해

$$r_{bubble} = \frac{(p-1)(t_f+t_b)}{m(t_f+t_b)} = \frac{p-1}{m}$$

	의미
p	파이프라인 단계 수 (GPU 개수)
m	micro-batch 개수
t_f, t_b	forward / backward 시간

- **GPU**를 많이 쓰면 → Bubble **증가** (병렬 비효율)
- **Micro-batch**를 늘리면 → Bubble **감소** (효율 향상)

6-2. Splitting layers: All forward, All backward

D) All-Forward All-Backward (AFAB)

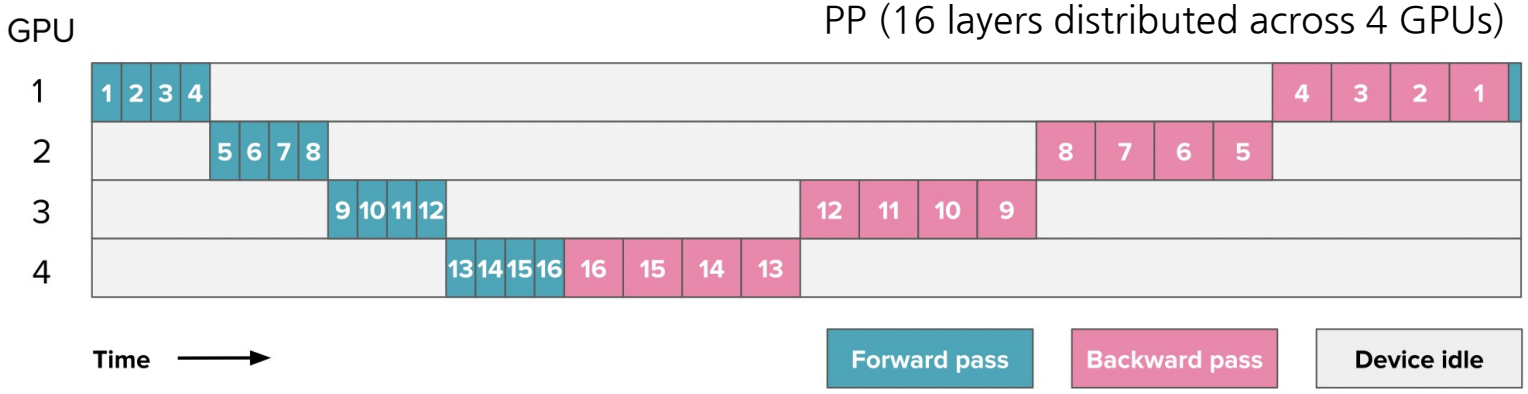
- **All-Forward All-Backward** (**AFAB**) 스케줄: **PP의 가장 단순한 스케줄링 방식**
- 한 줄 요약: Forward → Backward 순서 전체를 **"한 번에"** 반복하는 구조
- Procedure
  - Step 1) 한 GPU가 micro-batch 1~8의 forward를 순차적으로 처리
  - Step 2) Forward가 끝나면 backward를 시작
  - Step 3) Backward 또한 순차적으로 진행

6-2. Splitting layers: All forward, All backward

D) All-Forward All-Backward (AFAB)

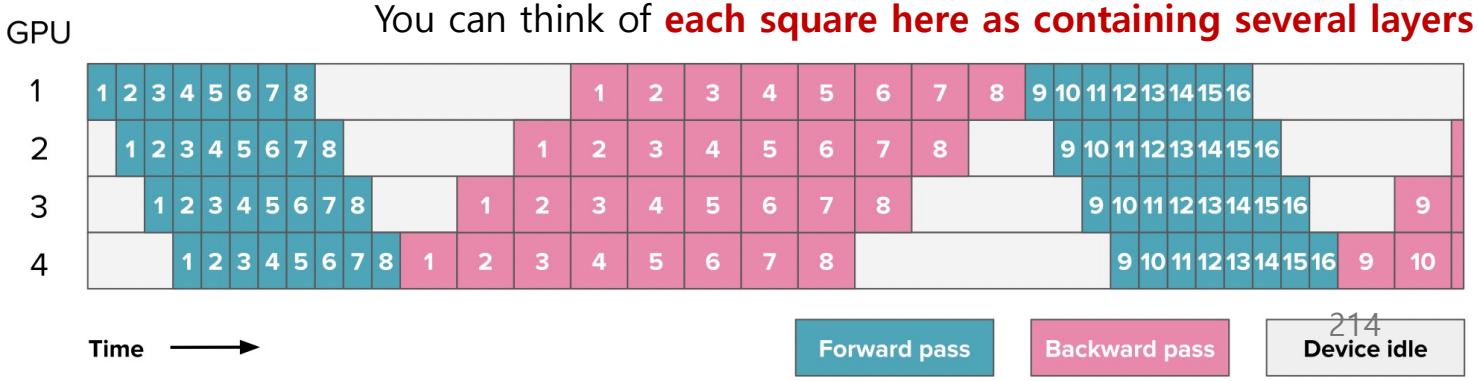
Forward only

PP (16 layers distributed across 4 GPUs)



All-Forward All-Backward (AFAB)

Before, the numbers in the diagram indicated the layers,
but in all pipeline parallel plots from here on they indicate **micro-batches**.
You can think of **each square here as containing several layers**



6-2. Splitting layers: All forward, All backward

E) AFAB로 인한 Bubble 감소

- AFAB에서는 micro-batch가 여러 개 존재하기 때문에, GPU 간 pipeline이 아래처럼 겹침

GPU	Forward 처리	Backward 처리
GPU_1	micro-batch **1~8**	micro-batch **8~1**
GPU_2	micro-batch **1~8**	micro-batch **8~1**
GPU_3	micro-batch **1~8**	micro-batch **8~1**
GPU_4	micro-batch **1~8**	micro-batch **8~1**

- 이렇게 되면 Forward/Backward 간 **pipeline 겹침**이 생김

  → **Bubble 비율이 약 1/m 만큼 줄어듬** (즉, micro-batch 8개면 bubble 약 1/8로 감소)

6-2. Splitting layers: All forward, All backward

F) AFAB의 단점

- Bubble은 줄었지만, **Activation 메모리 폭발**!
  - Forward시 계산된 activation은 Backward 전까지 모두 저장해야 함
  - 즉, micro-batch가 많아질수록 저장해야 할 activation도 많아짐
- 해결책: **1F1B (One Forward, One Backward)**
  - Forward와 Backward를 **교차**시키는 방식으로 메모리까지 절약 가능

6-2. Splitting layers: All forward, All backward

G) Summary

- Pipeline Parallelism은 모델 **Layer**를 여러 GPU에 분할하여 메모리를 절약

→ 한계: **Sequential 실행**으로 인해 "**bubble**"이 생김

- 이를 완화하기 위해 All-Forward All-Backward (**AFAB**) 스케줄을 사용해 **여러 micro-batch를 동시에** 처리

→ 한계: **Activation** 메모리 부담!

항목	설명
기본 개념	모델 레이어를 여러 GPU에 분할하여 순차적으로 처리
통신 단위	Activation (앞 GPU → 다음 GPU)
문제점	Sequential 실행으로 인한 GPU idle → Bubble 발생
해결 (AFAB)	Micro-batch 병렬화로 bubble 감소
부작용	Activation 저장량 증가 → 메모리 폭발
다음 단계	1F1B 스케줄 (Forward와 Backward를 교차)

6-3. One Forward One Backward

6-3. One Forward One Backward

A) 1F1B (One Forward One Backward)의 개요

- **Pipeline Parallelism(PP)** 의 고급 스케줄링 기법 중 하나
- **Llama 3.1** 같은 초대형 모델에서도 사용
- 기존의 AFAB:
  - 모든 Forward를 다 수행한 후
  - 모든 Backward를 순차적으로 수행

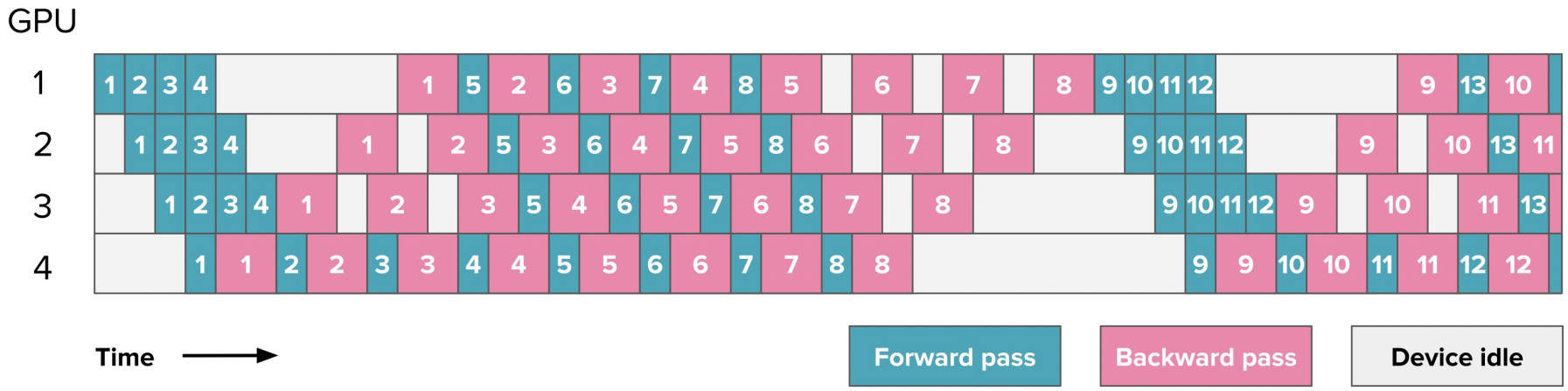
  → Bubble (대기 시간)을 줄이기 위해 micro-batch를 늘릴 수 있지만,

  → 그만큼 **activation을 전부 저장해야** 해서 메모리 폭발
- 해결책: **1F1B = One Forward, One Backward** (**interleaved pipeline scheduling**)

6-3. One Forward One Backward

B) Procedure

- 핵심 아이디어: "**가능한 한 빨리 backward를 시작**해서, activation을 오래 들고 있지 말자."
- 즉, **Forward와 Backward를 교차(interleave)** 시켜, Activation을 빨리 버림
- GPU idle 시간도 줄이는 구조

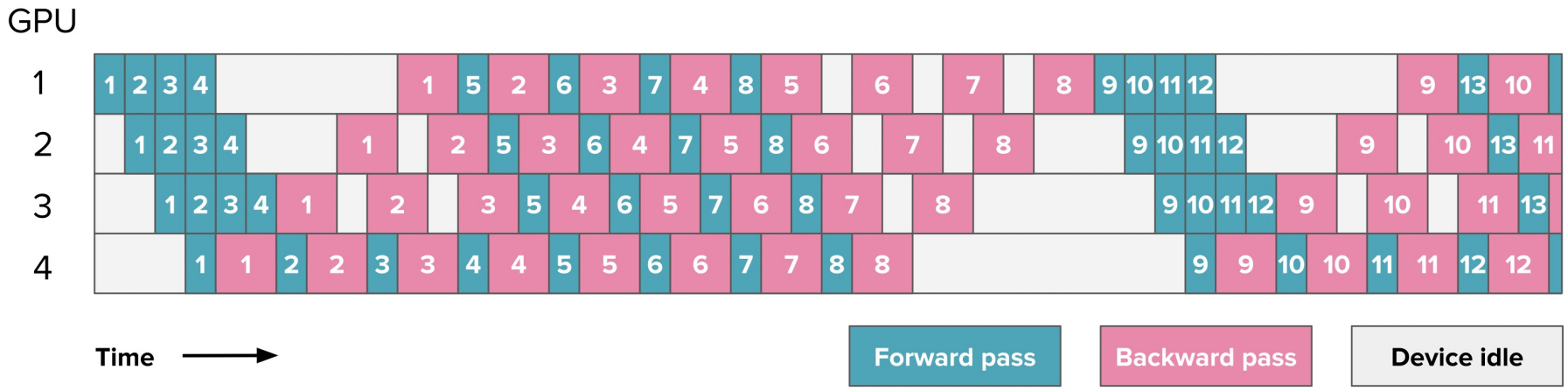


6-3. One Forward One Backward

B) Procedure

1F1B에서는 “마지막 output layer까지 다 통과할 필요 없이”
각 pipeline stage 단위로 backward를 바로 시작할 수 있음!

- Forward와 Backward가 **동시**에 일어남
  - 각 GPU는 forward **한 번**, backward **한 번**씩 번갈아 수행 (1F1B)
- Bubble이 여전히 존재하지만, **Activation 저장 개수 = GPU 수 (p)** 로 감소
  - AFAB에서는 micro-batch 개수 m 만큼 저장해야 했음

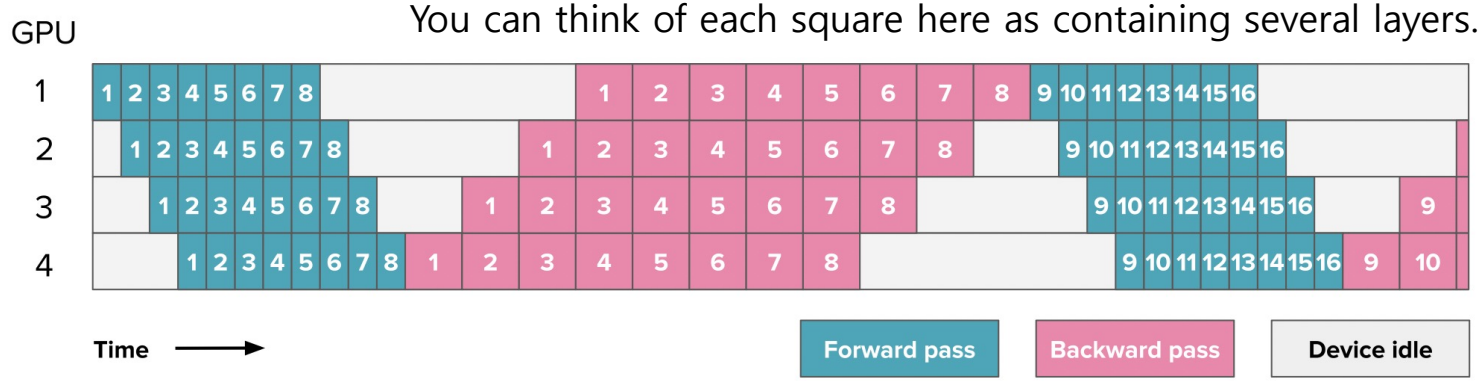


6-3. One Forward One Backward

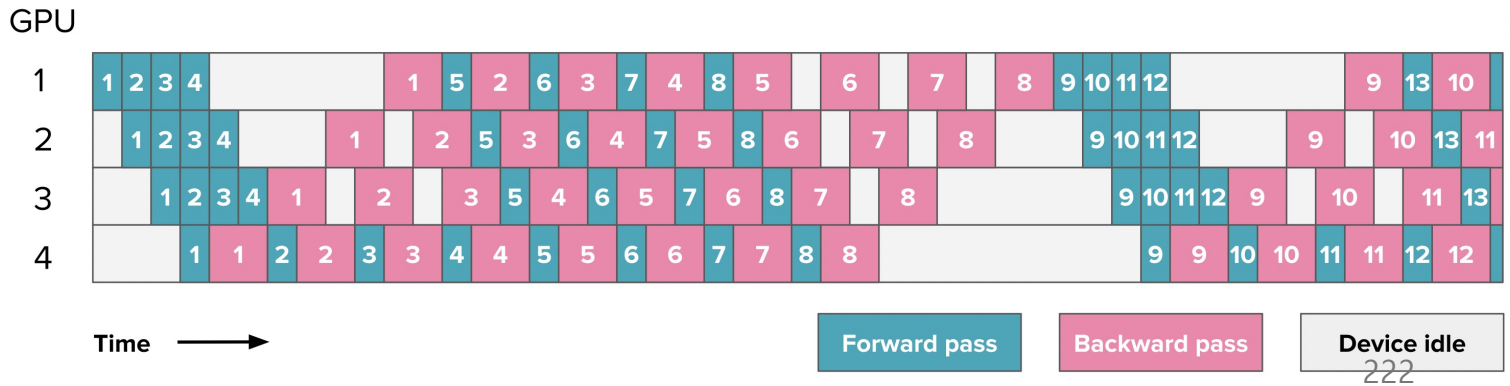
C) AFAB vs. 1F1B

All-Forward All-Backward (AFAB)

Before, the numbers in the diagram indicated the layers,
but in all pipeline parallel plots from here on they indicate **micro-batches**.
You can think of each square here as containing several layers.



One Forward One Backward (1F1B)



6-3. One Forward One Backward

C) AFAB vs. 1F1B

- **AFAB**: 전체 모델(**마지막 레이어**)까지 forward가 끝나야 backward를 시작
  - Forward → Backward 순서가 **철저히 분리**됨
  - 마지막 GPU (출력 레이어 담당)가 forward를 끝낼 때까지 다른 GPU는 backward를 기다림

  → Backward는 모델 전체의 output이 준비된 뒤에야 시작 가능

  → 결과: **bubble 크고**, **activation 메모리 많이 필요**

  (모든 micro-batch의 activation을 backward 전까지 다 저장해야 함)

6-3. One Forward One Backward

C) AFAB vs. 1F1B

- **1F1B**: **해당 stage의 forward**가 끝난 micro-batch에 대해 **곧바로 backward**
  - "한 stage의 forward가 끝나면, 그 stage는 즉시 backward를 시작 가능"
  - 파이프라인이 steady state에 들어가면, 각 GPU(stage)는 아래처럼 교차로 일함

시간	GPU1	GPU2	GPU3	GPU4
t_1	F1	—	—	—
t_2	F2	F1	—	—
t_3	F3	F2	F1	—
t_4	B1	F3	F2	F1
t_5	F4	B2	F3	F2
t_6	…	…	…	…

구분	AFAB	1F1B
Backward 시작 시점	마지막 GPU의 forward 완료 이후	각 stage의 forward 완료 직후
병렬성	없음 (순차)	있음 (forward/backward interleave)
Activation 메모리	매우 큼 (모든 micro-batch 저장)	적음 (stage별로만 저장)
GPU Idle 시간	큼 (bubble 많음)	적음 (steady state 유지)

6-3. One Forward One Backward

D) Bubble/Memory 크기 비교

- Bubble 크기는 동일하지만
- Activation 메모리만 GPU 수 (p)배 줄어듬

→ 그래서 실제로는 micro-batch를 더 많이 넣을 수 있음

→ 결과적으로 bubble 비율까지 줄어드는 효과가 생김

스케줄	Bubble 비율	Activation 메모리
AFAB	$(p-1)/m$	$\propto m$
1F1B	$(p-1)/m$	$\propto p$

6-3. One Forward One Backward

E) 구현 복잡성

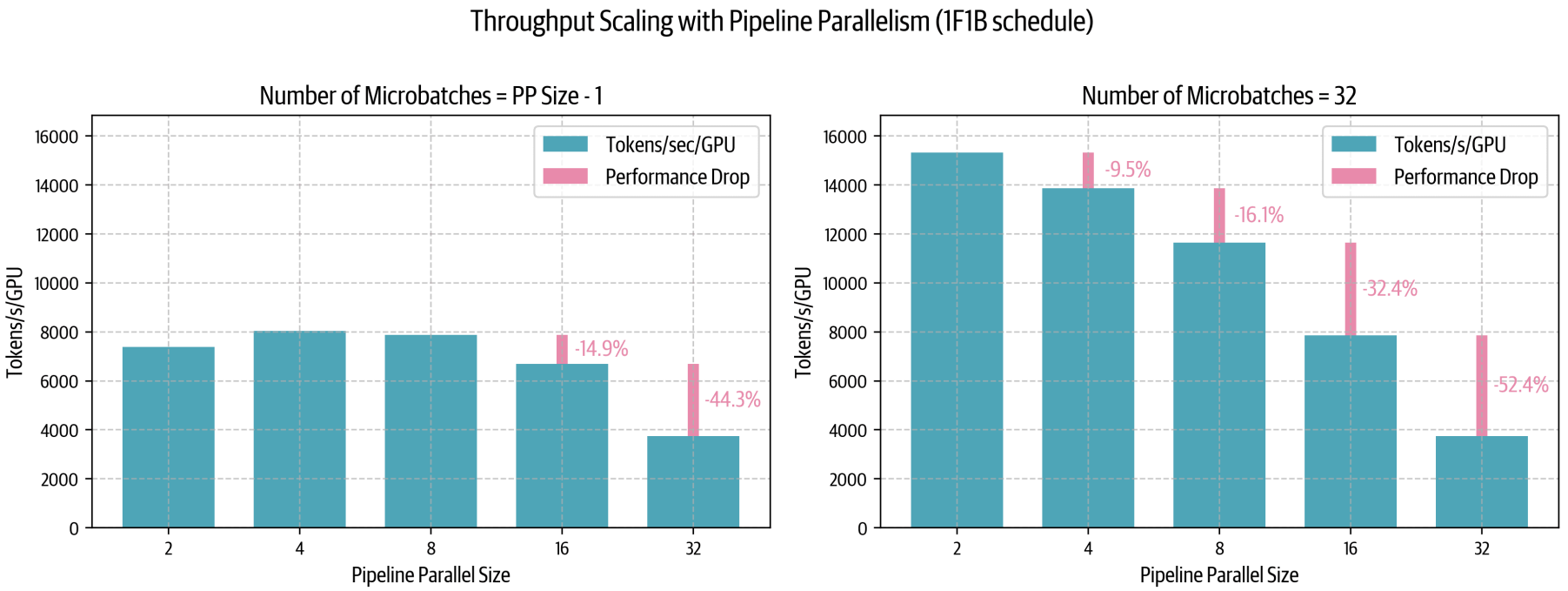
- 하지만, 1F1B는 Forward/Backward를 GPU별로 교차 수행하므로, **단순한 "한 루프" 구조로는 X**
- **GPU 1은 이미 backward 중인데, GPU 2는 forward 중일 수 있음**.
- 각 GPU가 **독립적**으로 **forward ↔ backward 전환** 시점을 관리해야 함.

→ 따라서 구현 복잡도가 급격히 증가

→ **DeepSpeed**나 **Megatron-LM** 같은 프레임워크가 이를 자동 스케줄링

6-3. One Forward One Backward

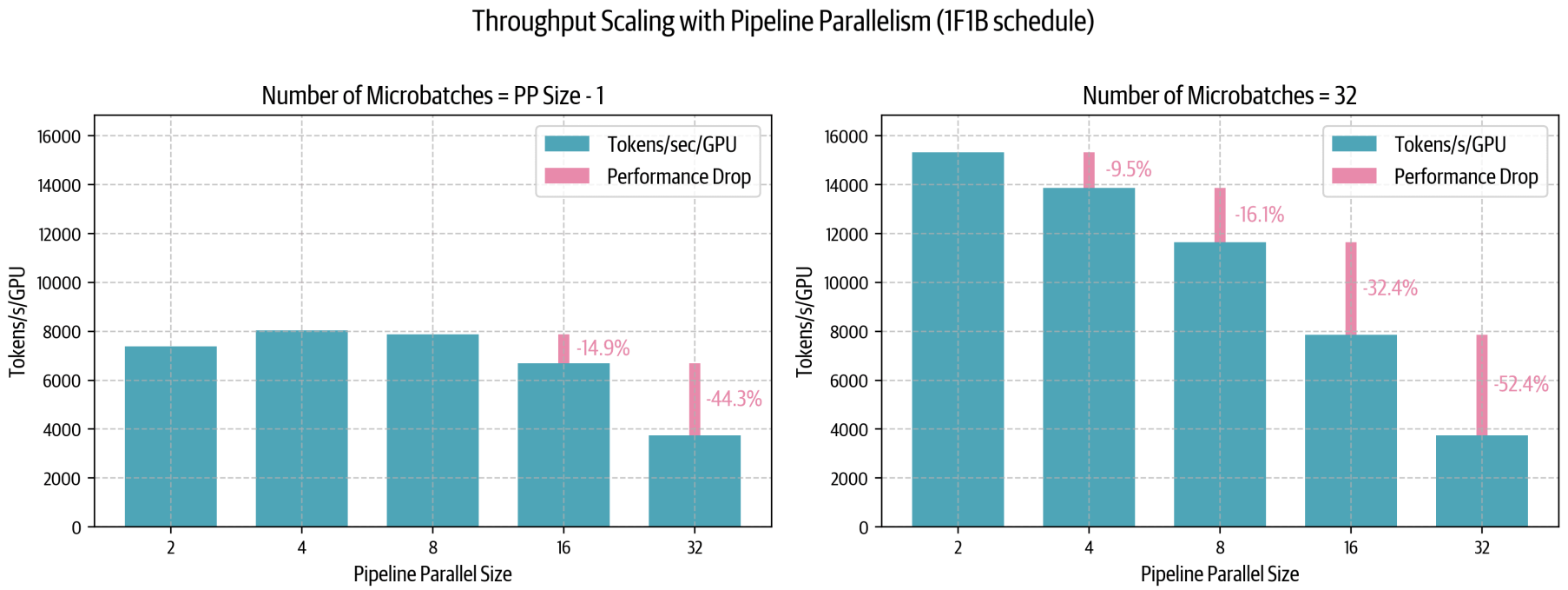
F) 성능



그래프	왼쪽	오른쪽
설명	micro-batch 수 = PP size - 1	micro-batch 수 = 32
결과 요약	작은 m에서는 bubble 영향 ↑	큰 m에서는 throughput 향상 ↑

6-3. One Forward One Backward

F) 성능

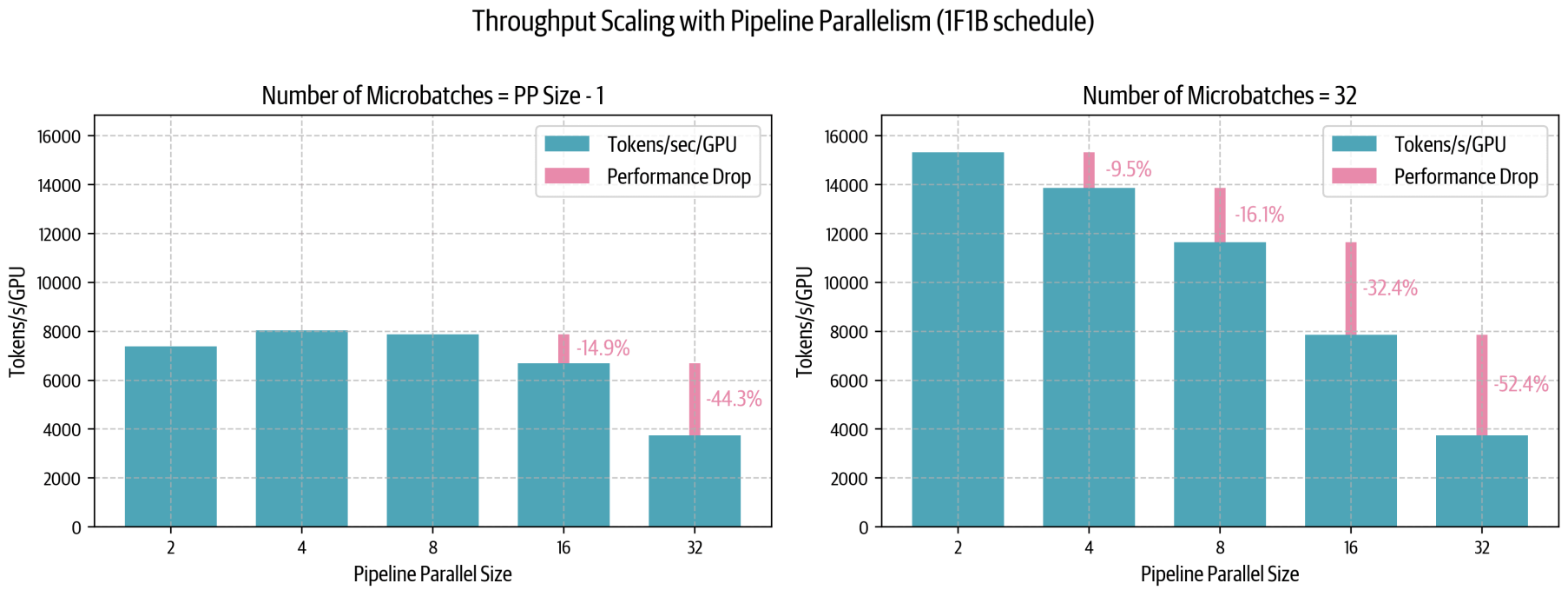


왼쪽 그래프 (m = p−1)

- micro-batch 수가 적을 때는 bubble 비율이 커서 성능이 낮음.
- 예: PP=32일 때 성능이 44% 감소.

6-3. One Forward One Backward

F) 성능



오른쪽 그래프 ($m = 32 \gg p-1$)

- micro-batch를 늘리면 bubble 영향 ↓
- throughput이 거의 linear하게 증가.
- 단, micro-batch를 무한히 늘릴 순 없음 (global batch 한계 때문에).

6-4. Interleaving Stages

6-4. Interleaving Stages

A) Interleaved Pipeline Parallelism

- 목적: **1F1B** 스케줄에서 **남아 있는 bubble**을 더 줄이기 위해!
- **Llama 3.1**에서 사용되는 고급 파이프라인 병렬화 스케줄
- 1F1B 스케줄의 한계점:
  - 비록 activation 메모리를 크게 줄여줬지만, **완벽하게 bubble을 줄인 것은 X**
  - 이유: 각 GPU는 forward → backward를 번갈아 수행하지만,

    forward/backward가 "순서대로" 연결되어 있어서 **여전히 일부 GPU**는 다른 GPU가 끝날 때까지 기다림.
- 해결책: **Interleaved Pipeline Parallelism (Interleaving Stages)**

6-4. Interleaving Stages

A) Interleaved Pipeline Parallelism

- 기본 아이디어: “GPU 안에서도 pipeline을 나누자”
- 기존 pipeline 은 모델을 GPU 단위로만 분할했었음
  - GPU1: layer 1-4
  - GPU2: layer 5-8
  - GPU3: layer 9-12
  - GPU4: layer 13-16

→ 즉, 하나의 GPU = 하나의 연속된 stage

- **하지만 Interleaving에서는 오른쪽과 같이 나눔** =>

한 GPU가 모델의 두 구간(stage)을 동시에 담당!

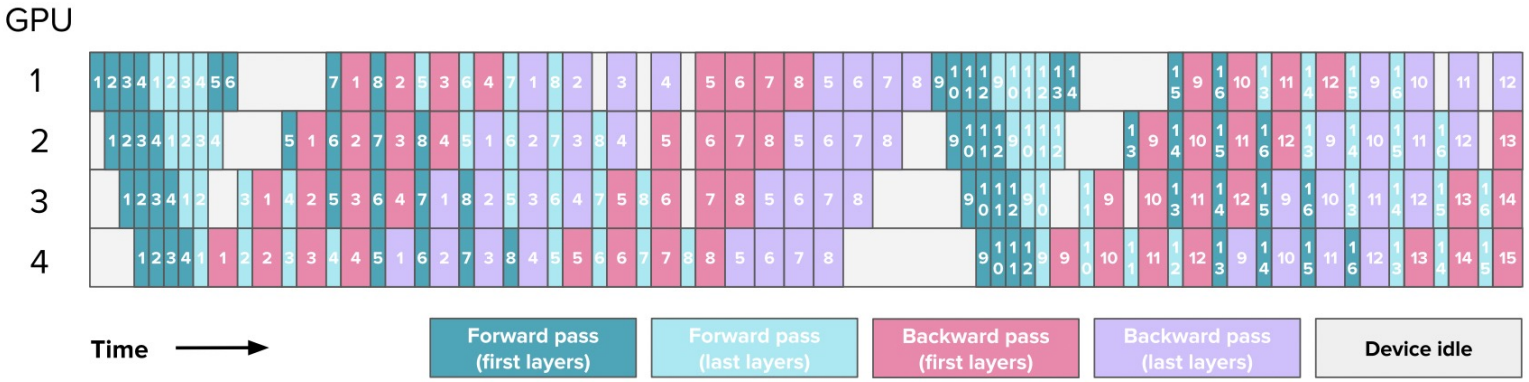
(GPU1은 모델의 “**앞부분**”과 “**뒷부분**” 둘 다 담당)

GPU	Stage 1	Stage 2
GPU1	**Layer 1-2**	**Layer 9-10**
GPU2	**Layer 3-4**	**Layer 11-12**
GPU3	**Layer 5-6**	**Layer 13-14**
GPU4	**Layer 7-8**	**Layer 15-16**

6-4. Interleaving Stages

A) Interleaved Pipeline Parallelism

- 도식화



구간	색상
진한 파랑	Forward (초기 레이어)
옅은 파랑	Forward (후반 레이어)
진한 분홍	Backward (초기 레이어)
보라색	Backward (후반 레이어)

한 GPU가 첫 stage의 forward를 수행하다가, 다른 GPU가 backward를 수행하는 동안,

자신은 두 번째 stage의 forward를 시작

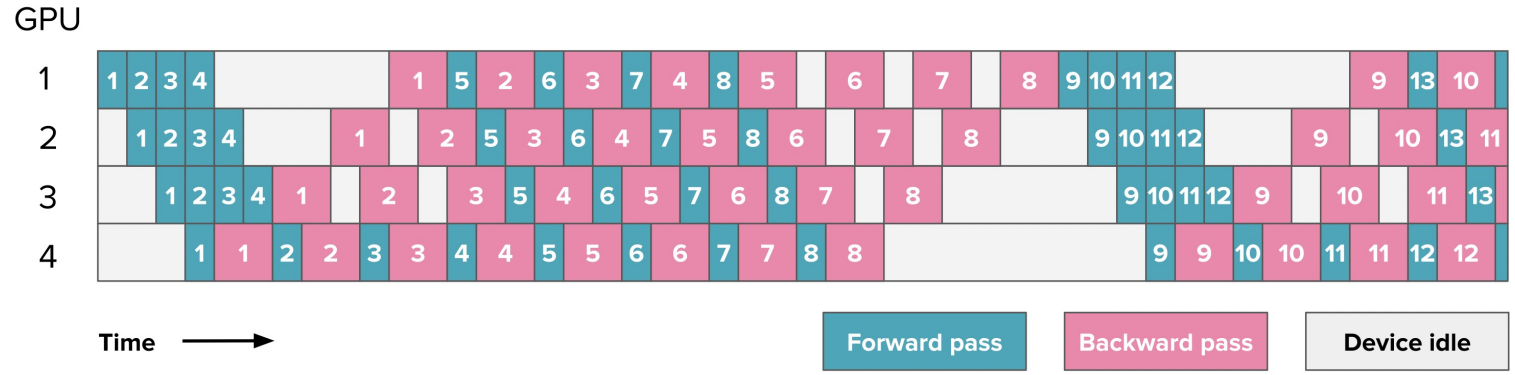
→ Forward와 Backward가 더 세밀하게 interleave됨

→ GPU idle 구간이 더 짧아짐 → bubble 감소!

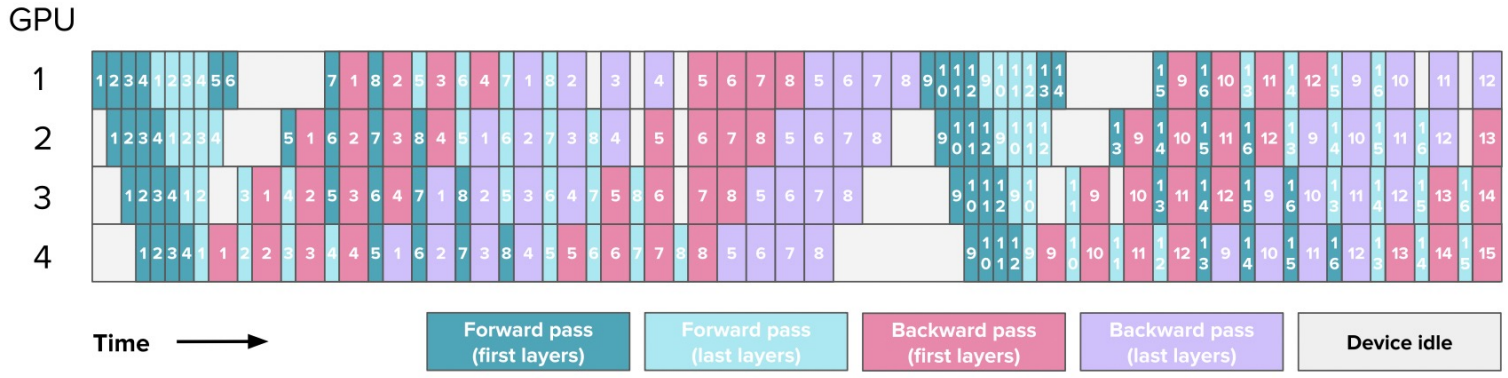
6-4. Interleaving Stages

A) Interleaved Pipeline Parallelism

One Forward One Backward (1F1B)



Interleaving Stages



6-4. Interleaving Stages

B) 수식으로 보는 Bubble 감소 효과

$$t_{pb} = \frac{(p-1)(t_f+t_b)}{v}$$

$$r_{bubble} = \frac{p-1}{v \times m}$$

기호	의미
p	pipeline 단계 수 (GPU 개수)
v	각 GPU 내의 interleaved stage 수
m	micro-batch 수
r_{bubble}	bubble 비율

구성	Bubble 비율
일반 1F1B	(p-1) / m
Interleaved v=2	(p-1) / (2m)
Interleaved v=4	(p-1) / (4m)

interleaved stage 수 v 가 늘어날수록 bubble이 1/v로 줄어듬!

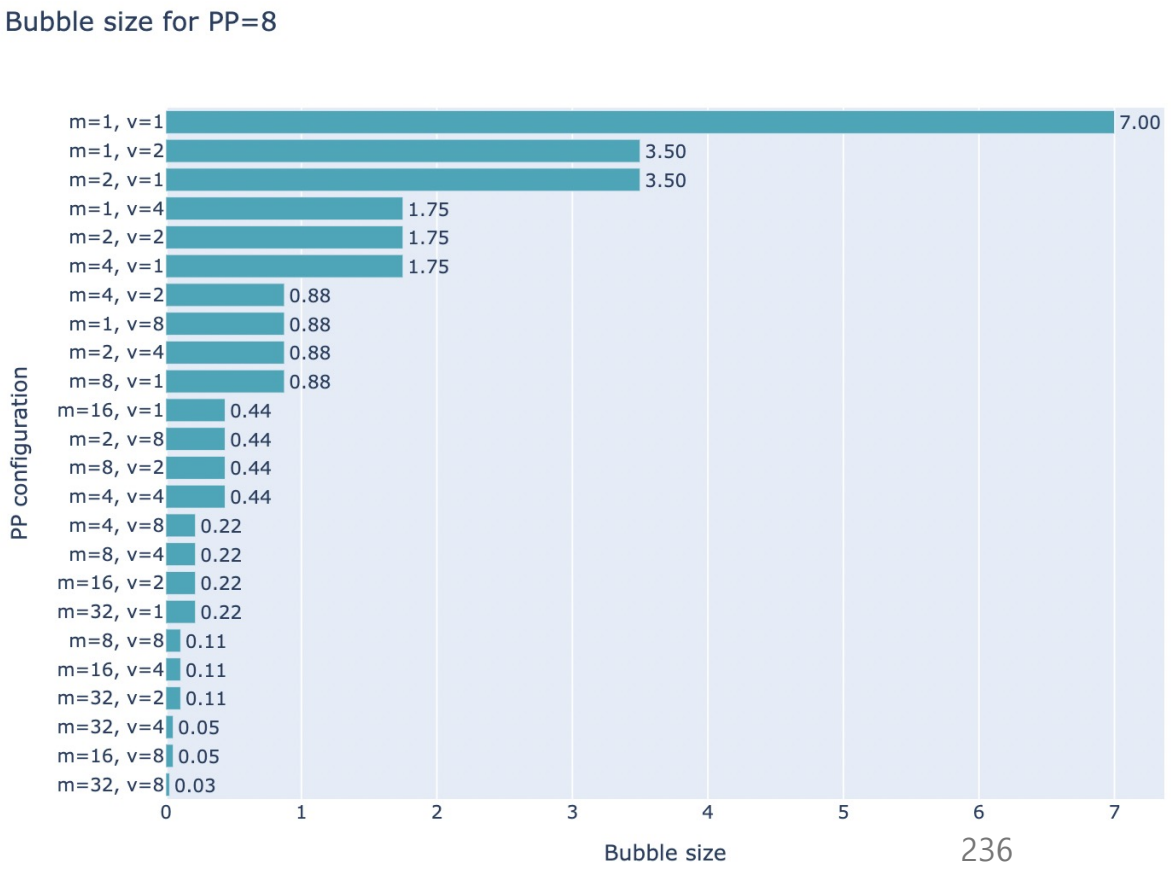
6-4. Interleaving Stages

(m=micro-batch, v=interleaved stage 수)

C) Bubble 크기 비교

- 세팅: 여기서 PP=8 (pipeline 단계 8개)로 고정
- 결론: Bubble이 작아짐 (GPU idle 시간 거의 0).
  - micro-batch 수가 많을수록 (m ↑),
  - interleaving stage가 많을수록 (v ↑),

구성	Bubble 크기
m=1, v=1	7.0
m=1, v=2	3.5
m=1, v=4	1.75
m=8, v=4	0.22
m=32, v=8	0.03



6-4. Interleaving Stages

D) Llama의 실제 구현

- Setting
  - 총 8개 pipeline stage (0~7)
  - 4개의 GPU rank (PP rank 0~3)
- **1F1B + interleaved (v=2)** 형태
- 결론:
  - Bubble을 절반으로 줄이면서도
  - Activation 메모리도 1F1B 수준으로 유지

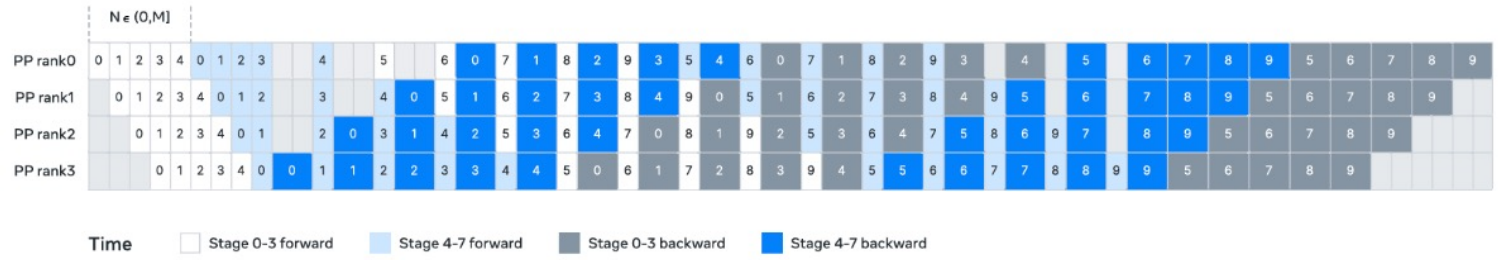


Figure 6 Illustration of pipeline parallelism in Llama 3. Pipeline parallelism partitions eight pipeline stages (0 to 7) across four pipeline ranks (PP ranks 0 to 3), where the GPUs with rank 0 run stages 0 and 4, the GPUs with P rank 1 run stages 1 and 5, *etc.* The colored blocks (0 to 9) represent a sequence of micro-batches, where M is the total number of micro-batches and N is the number of continuous micro-batches for the same stage's forward or backward. Our key insight is to make N tunable.

Rank	Stage 담당
Rank 0	Stage 0, 4
Rank 1	Stage 1, 5
Rank 2	Stage 2, 6
Rank 3	Stage 3, 7

6-5. Zero Bubble and DualPipe

6-5. Zero Bubble and DualPipe

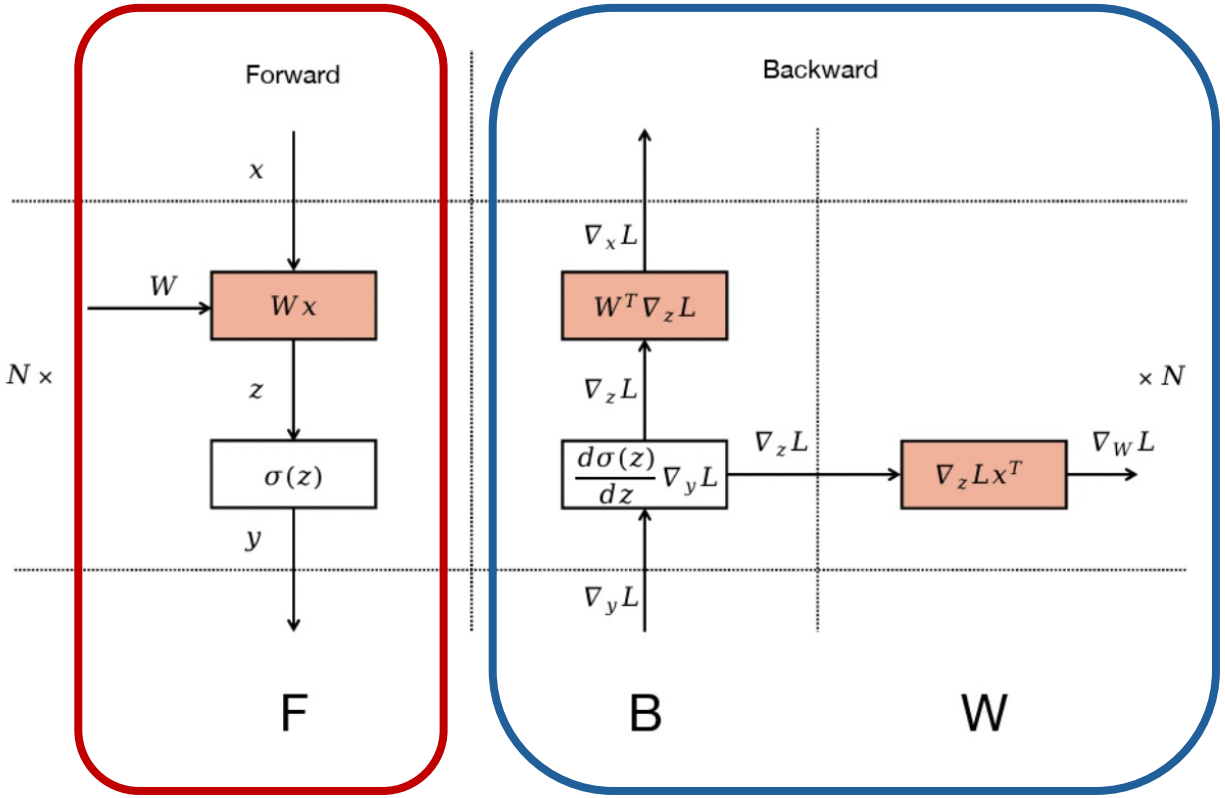
A) Computation Graph for MLP

Forward

- 입력 $x \rightarrow$ 가중치 W와 곱: $z = Wx$
- 활성화 $\sigma(z)$ 적용 $\rightarrow$ 출력 y
- 즉: $x \xrightarrow{W} z \xrightarrow{\sigma} y$

Backward

- Loss L의 gradient $\nabla_y L$ 로부터 역전파
- $\nabla_z L = \frac{d\sigma(z)}{dz} \nabla_y L$
- $\nabla_x L = W^T \nabla_z L$ 아래 layer로 전달할 gradient
- $\nabla_W L = \nabla_z L \, x^T$ weight 업데이트용 gradient



6-5. Zero Bubble and DualPipe

B) Review of 1F1B

- 한 micro-batch의 **forward 1 단계**를 수행한 뒤, **backward 1 단계**를 바로 interleave 해서 pipeline을 채움
- (그림) **4개의 GPU**가 서로 다른 micro-batch를 순서대로 처리
  - 각 칸이 하나의 micro-batch 처리 단계
- 문제점: 맨 앞/뒤 구간에서 GPU들이 기다리는 "bubble"이 존재 → 효율 ↓

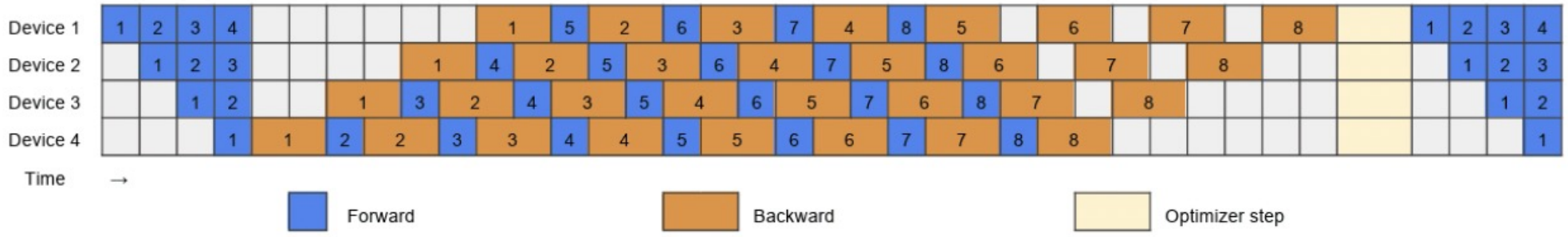


Figure 2: 1F1B pipeline schedule.

6-5. Zero Bubble and DualPipe

C) Zero Bubble (ZB)

- ZB-H1 / ZB-H2: “Zero Bubble” 논문에서 제안된 **세밀한 Backward 분할** 스케줄
- Backward를 **두 부분**으로 나눔:
  - 🟦 B = Backward for inputs ($\nabla_x L$)
  - 🟩 W = Backward for weights ($\nabla_W L$)
- **두 연산(B, W)을 분리**해서 다른 시간 슬롯에 배치
- 결론: **B (입력 gradient)는 다음 stage 역전파에 필요**하므로 **빨리 계산**

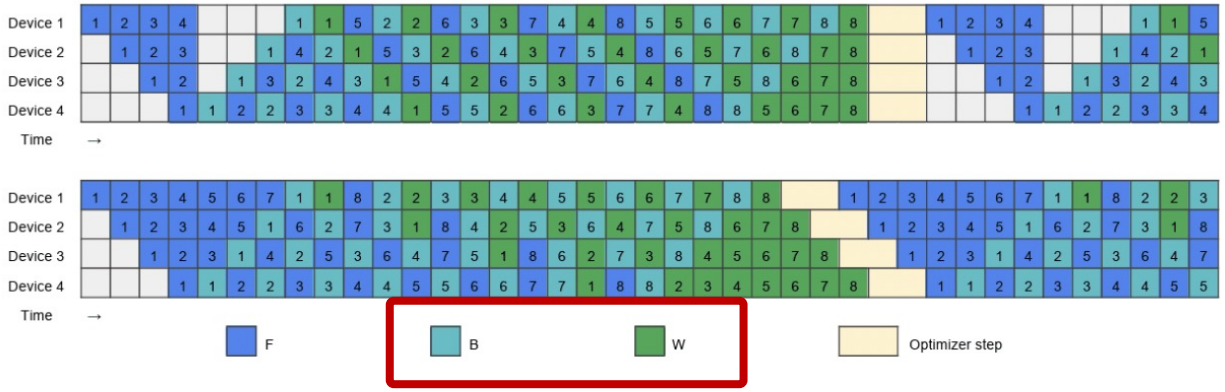


Figure 3: Handcrafted pipeline schedules, top: ZB-H1; bottom: ZB-H2

Backward를 둘로 나눔

6-5. Zero Bubble and DualPipe

C) Zero Bubble (ZB)

- **B (입력 gradient)**: 다음 stage 역전파에 필요하므로 **빨리** 계산
- **W (가중치 gradient)**: optimizer step전까지만 계산하면 되므로 유연하게 **뒤로 미룸**

→ Bubble에 W 연산을 채워넣어 GPU 유휴 시간 거의 0 으로 만듦.

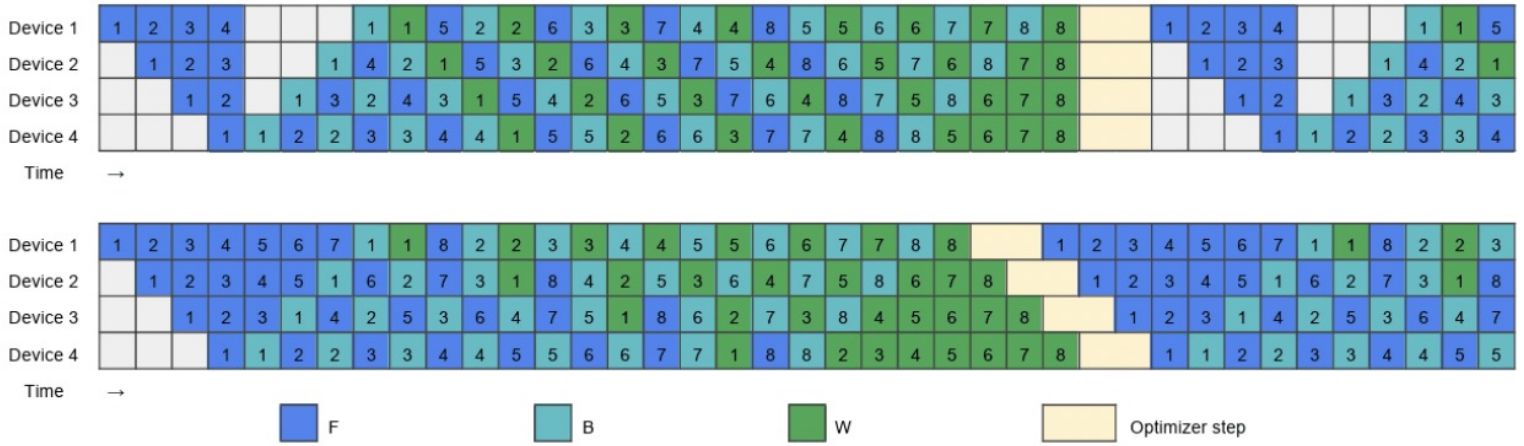


Figure 3: Handcrafted pipeline schedules, top: ZB-H1; bottom: ZB-H2

6-5. Zero Bubble and DualPipe

D) DualPipe (feat. DeepSeek-V3/R1)

- ZB를 확장한 최신 스케줄
- 두 방향으로 Pipeline을 흘려보내며 (위→아래, 아래→위) forward/backward를 서로 겹쳐서 진행.
- 결과: 거의 “zero-bubble” 상태 달성 (+ 통신(all-to-all) 오버헤드도 최소화)
  - DeepSeek-V3: “near-zero all-to-all communication overhead”

Figure 5 | Example DualPipe scheduling for 8 PP ranks and 20 micro-batches in two directions. The micro-batches in the reverse direction are symmetric to those in the forward direction, so we omit their batch ID for illustration simplicity. Two cells enclosed by a shared black border have mutually overlapped computation and communication.

6-5. Zero Bubble and DualPipe

E) Summary

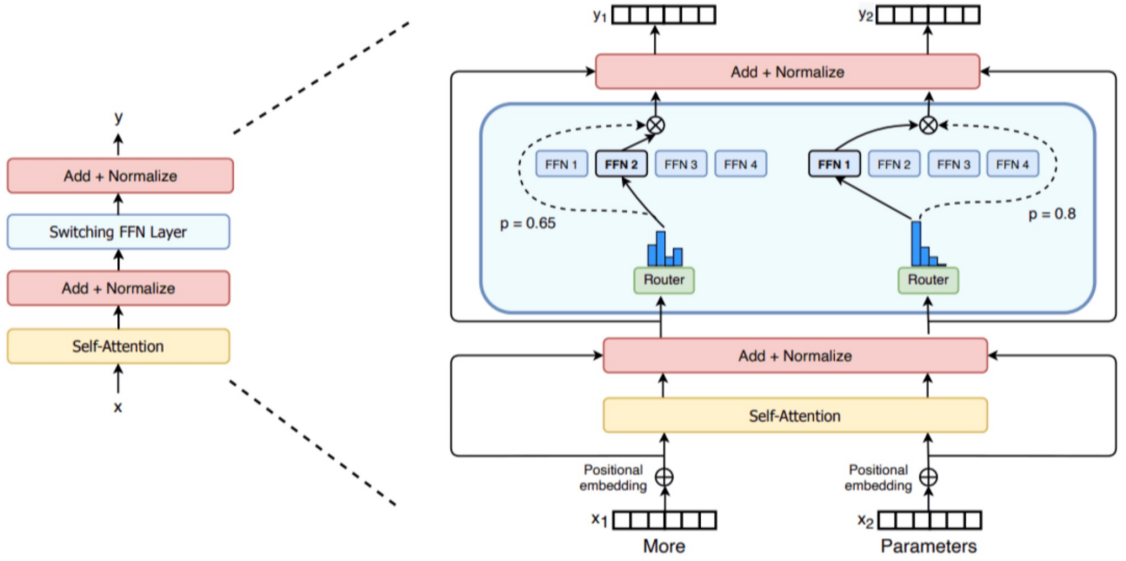
스케줄	아이디어	특징
1F1B	Forward 와 Backward 교차	간단하지만 bubble 존재
ZB-H1/H2	Backward 를 (B, W) 로 분리	bubble 거의 0 (이론적 zero)
DualPipe	양방향 파이프 + (B/W) 분리 겹침	실제 zero-bubble 가까움 + 통신 효율 ↑

7. Expert Parallelism

7. Expert Parallelism

A) Mixture of Experts (MoE) 모델

- MoE (X): 각 층의 FFN이 단 하나뿐
- MoE (O): 한 층에 여러 개의 전문가 네트워크(Experts, FFN1, FFN2, …)
  - 입력 token마다 “어떤 expert에게 보낼지”를 router (게이트) 가 결정



7. Expert Parallelism

B) Router의 역할

- FFN1, FFN2, FFN3, FFN4: 각각 다른 expert
- Router가 각 입력 token x_i 에 대해 “어떤 expert로 보낼지” 확률을 계산
  - x_1 → FFN2 (p=0.65),
  - x_2 → FFN1 (p=0.8)
- Top-k gating: 각 token은 가장 높은 확률의 k개 expert로만 라우팅 (예: Top-1 또는 Top-2).

7. Expert Parallelism

C) Expert Parallelism (EP)

- 각 Expert를 다른 GPU에 분산시켜 병렬 실행
- 기본 원리
  - MoE layer 내부의 각 Expert는 독립적 (Parameter sharing X)
  - 따라서, 서로 다른 GPU에 두어도 ok
- 그러면 Router는 각 토큰을 “적절한 GPU로” 라우팅
- 즉, 각 GPU는 자신에게 배정된 Expert의 FFN만 계산하면 됨!

GPU	Expert
GPU1	FFN1
GPU2	FFN2
GPU3	FFN3
GPU4	FFN4

7. Expert Parallelism

C) Expert Parallelism (EP)

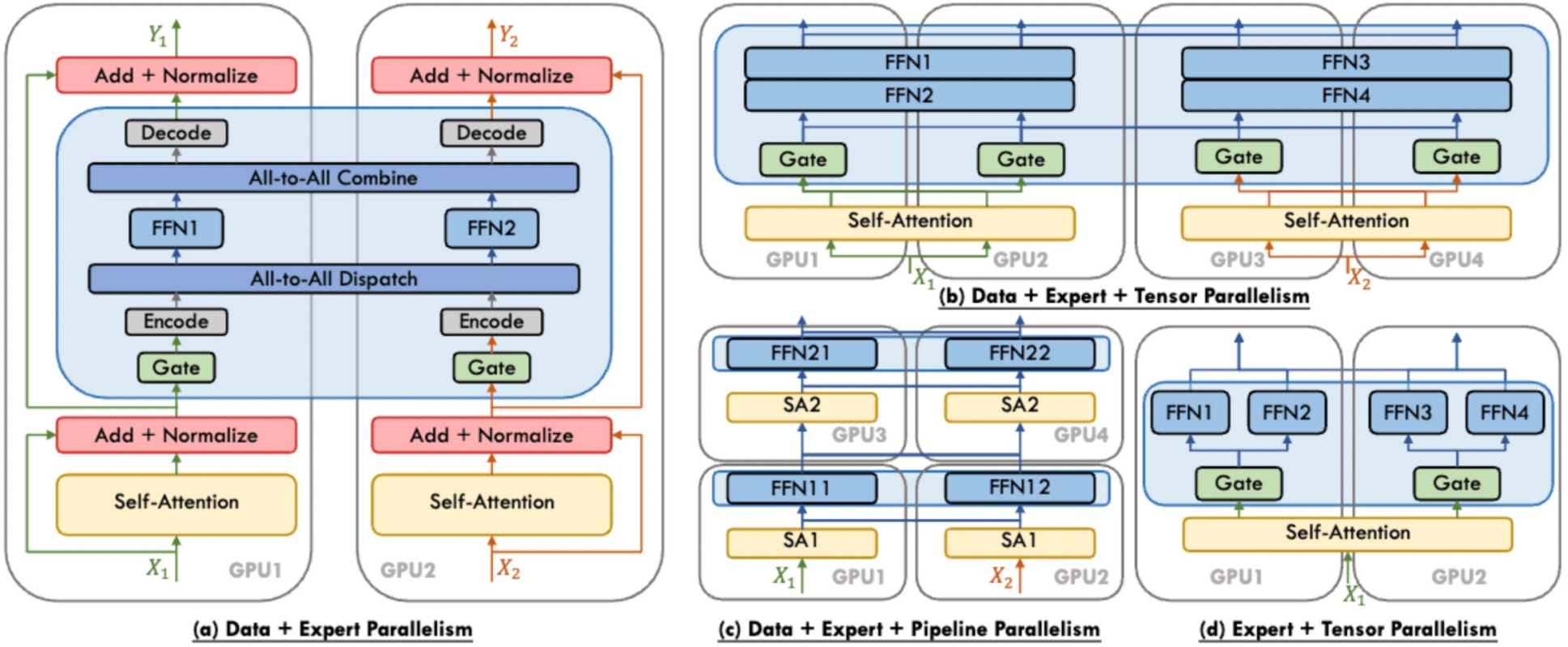


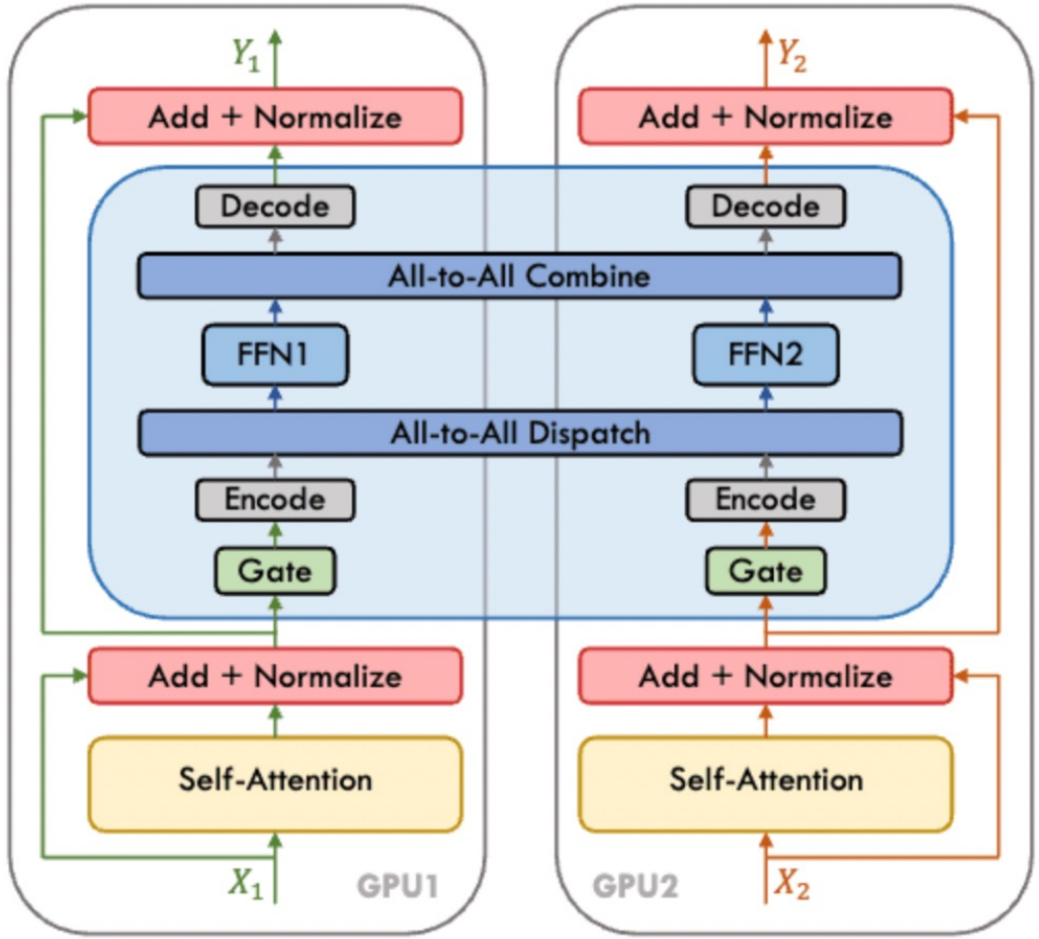
Fig. 8. Schematic depiction of diverse parallel strategies for MoE. For clarity and conciseness, this illustration omits some All-to-All, All-Reduce, Point-to-Point communication within parallelism, and Normalization, Encode, Decode, Gate in subfigures (b), (c), and (d).

7. Expert Parallelism

D) EP의 흐름 요약

Fig.8 (a)의 “All-to-All Dispatch / Combine”

- Step 1) 입력 token → Router로 들어감
- Step 2) Router가 “어떤 Expert로 갈지” 결정
- Step 3) All-to-All 통신으로, **각 token을 해당 Expert가 있는 GPU로** 전송 (**Dispatch**)
- Step 4) 각 GPU의 Expert가 FFN 계산 수행
- Step 5)다시 **All-to-All로 결과를 합쳐서** 원래 순서대로 복원 (**Combine**)



(a) Data + Expert Parallelism

7. Expert Parallelism

E) 실제 구현에서의 최적화

- 문제점: “모든 토큰을 All-to-All로 GPU 사이에 보내는 것”이 통신 부하가 크다!
- DeepSeek-V3의 최적화 (해결책) :
  - 각 토큰이 최대 M개의 Expert에만 라우팅되게 제한 (예: M=4) → 통신량 제한
  - Router가 가능하면 같은 노드 안의 Expert로만 토큰을 보내도록 제약 → 노드 간 통신 최소화

→ 이렇게 하면 Expert Parallelism의 효율이 훨씬 좋아짐

8. 5D Parallelism in a Nutshell

8. 5D Parallelism in a Nutshell

A) 5D Parallelism

- 이 **5가지**는 모델의 **서로 다른 차원을 분할**해서 계산 부하를 분산

병렬화 방법	분할 기준 (Sharding dim)	핵심 아이디어
Data (DP)	Batch	**데이터**를 나눠 **여러 GPU에 복제된 모델**을 동시에 학습
Tensor (TP)	Hidden	한 layer의 **행렬 곱 연산**을 여러 GPU가 분할해서 수행
Sequence / **Context** (SP / CP)	Sequence length	**입력 시퀀스**를 여러 GPU에 나눠서 **긴 문장**도 처리 가능
Pipeline (PP)	Layers	**모델의 layer들**을 여러 GPU에 **순서대로** 분배 (pipeline)
Expert (EP)	Experts (MoE)	여러 **FFN(Expert)**을 GPU마다 나눠서 병렬 실행

8. 5D Parallelism in a Nutshell

B) ZeRO의 3단계

- **ZeRO**는 Data Parallelism (**DP**) 기반에서 메모리 절감을 위한 최적화 전략
- ZeRO-1/2/3은 **"무엇을 shard할지"**에 따라 **단계적**으로 나뉨
- **ZeRO-3**은 가장 강력하지만, 통신량이 많아 구현이 복잡해짐

단계	Sharding 범위	설명
ZeRO-1	**Optim states**	Optimizer(예: Adam)의 1차, 2차 모멘트를 GPU 간 분할
ZeRO-2	Optim states + **Gradients**	Gradient 자체도 분산 저장 (통신으로 집계)
ZeRO-3	Optim states + Gradients + **Parameters**	모델 param까지 GPU 간 분할 저장 → 전체 메모리 최적

8. 5D Parallelism in a Nutshell

C) PP vs. ZeRO-3

- (공통점) "**모델 parameter**를 GPU에 분할 저장"
- (차이점) **GPU 간 무엇을 주고받는지 (통신 단위)** 가 다름

비교 항목	ZeRO-3	Pipeline Parallelism (PP)
GPU가 저장하는 내용	**한 layer의 일부** weight shard	**전체 layer** 단위
통신 시점	layer 계산 시, 다음 **layer weight** prefetch	forward/backward 중 **activation** 전달
통신 대상	Weights	Activations
구현 난이도	**Parameter shard** 관리 복잡	**Pipeline schedule** 설계 복잡
확장 성능	큰 mini-batch, 긴 seq_len에 유리 (통신 숨기기 쉬움)	큰 gradient_accumulation에 유리 (bubble 숨기기)
공통점	모델 깊이 축을 따라 분할, 통신-계산 겹치기 가능	

8. 5D Parallelism in a Nutshell

C) PP vs. ZeRO-3

- Summary
  - **ZeRO-3**: “파라미터 중심 (**weight** transfer)” 방식
  - **PP**: “활성값 중심 (**activation** transfer)” 방식
- 둘 다 layer 단위 분할이지만 **통신 방향이 다름**
- **함께 쓸 수는 있지만 잘 안 씀** (통신비용이 커지기 때문)
- 결합하려면, ZeRO-3이 pipeline의 여러 micro-batch 동안 weight를 메모리에 유지해야 함

→ 불필요한 weight 통신을 줄이기 위해서

8. 5D Parallelism in a Nutshell

D) PP & ZeRO-1,2

- 이들은 쉽게 결합 가능!
- ZeRO-1,2는 optimizer state/gradient만 shard하므로, model layer 분할(=PP) 과 독립적
- E.g., DeepSeek-V3는 PP + ZeRO-1 조합을 사용

8. 5D Parallelism in a Nutshell

E) TP + SP

- TP/SP는 모델 내부 **행렬곱 연산**을 쪼개는 방식 → **PP, ZeRO-3**과 자연스럽게 호환!
  - 이유: 각 layer 내부 연산이 분산될 뿐, **layer 단위 흐름은 동일**하기 때문
- TP의 장단점
  - 장점: 통신과 연산을 잘 겹치면 고속 학습 가능
  - 단점:
    - 통신이 연산의 critical path에 직접 포함되므로 **너무 많이 scale-up 시 성능 급락**
    - 구현이 모델 의존적 (hidden dimension vs sequence dimension에서 **shard 패턴 관리 필요**)

8. 5D Parallelism in a Nutshell

E) TP + SP

- 결합 시 고려사항
  - **TP 그룹**(통신하는 GPU들)은 **node 내부**(high-bandwidth NVLink) 에 묶어야 함.
  - **inter-node 통신**엔 **ZeRO-3 또는 PP**를 쓰는 게 효율적
    - 이들은 통신 패턴이 단순하고, 계산과 통신을 더 쉽게 겹칠 수 있음

8. 5D Parallelism in a Nutshell

F) CP

- **SP의 확장판**: 아주 **긴 sequence** (예: 128k 토큰) 학습을 위해 설계됨.
- **Sequence 길이 방향**으로 activations을 shard.
- MLP, LayerNorm 등은 독립적으로 처리 O, but **Attention**은 **모든 토큰의 K/V**를 봐야 하므로 통신 필요
- 이를 위해 **Ring-Attention** 패턴 사용 (통신과 계산을 overlap)
- 장/단점
  - 장점: 극단적으로 긴 sequence도 GPU 메모리에 맞춰 분산 처리 가능
  - 단점: Attention에서 **통신 오버헤드** 존재

8. 5D Parallelism in a Nutshell

G) EP

- **MoE (Mixture of Experts)** layer에서만 적용
- 각 **Expert의 FFN**을 서로 다른 GPU에 분산
- Router가 각 token을 적절한 Expert GPU로 All-to-All 통신으로 라우팅하고, 계산 후 결과를 다시 모음
- 토큰마다 선택된 Expert만 활성화되므로, 통신량은 있지만 모델 용량을 극대화할 수 있음
- EP & DP의 공통점/차이점
  - (공통점) EP도 **입력 데이터를 여러 GPU에 분산** 처리하므로 **DP의 특수 형태**로 볼 수 있음
  - (차이점) **DP**는 모든 GPU가 **동일** 모델을 돌림 / **EP**는 GPU마다 Expert가 **다름**

8. 5D Parallelism in a Nutshell

H) 각 병렬화가 적용되는 영역

전략	주로 영향을 주는 부분	통신의 초점
TP + SP	모델 전체 (**행렬곱**)	Matrix multiply (column/row)
CP	**Attention** block	Keys/Values 교환
EP	**MoE FFN** block	Token routing All-to-All
PP	**전체 layer 흐름**	Activation 전송
DP	**Batch** 단위 복제	Gradient 평균화
ZeRO-1/2/3	**Optimizer states / Grad / Params**	파라미터 shard 간 통신

8. 5D Parallelism in a Nutshell

I) 통신의 특성 & 병렬화 적합 영역

병렬화	통신 패턴	권장 환경	주요 한계
TP+SP	높은 대역폭 All-Reduce	intra-node NVLink	cross-node 확장 시 통신 병목
CP	Attention의 Key/Value 교환	긴 시퀀스 학습	통신비 증가
EP	All-to-All 라우팅	MoE 구조	Routing 통신 오버헤드
PP	Activation 전달	Layer 깊은 모델	Pipeline bubble 관리 필요
DP	Gradient All-Reduce	모든 환경	batch 한계 존재
ZeRO-1/2/3	파라미터 shard 간 통신	DP 조합용	shard 관리 복잡성

8. 5D Parallelism in a Nutshell

I) 통신의 특성 & 병렬화 적합 영역

결합 방식	특징	주의점
PP + ZeRO-1/2	가장 흔한 조합	별도 충돌 없음
PP + ZeRO-3	가능하나 드뭄	통신량 커서 global batch↑ 필요
TP + SP + PP/ZeRO-3	자연스러운 결합	TP 그룹은 node 내부로 제한
CP + TP	긴 sequence 처리용	Attention 통신 관리 필요
EP + DP/TP	MoE 학습용	All-to-All 비용 관리 필요

8. 5D Parallelism in a Nutshell

J) 각 전략의 메모리 절감 및 병목

방법	메모리 절감 대상	분할 차원	단점 / 병목
DP	Activations	Batch	Batch 크기 한계
PP	Model parameters	Layers	Idle bubble, 복잡한 스케줄
TP+SP	Params + Activations	Hidden / Seq	통신 대역폭 필요, 모델 종속
CP	Activations	Sequence	Attention 통신 오버헤드
EP	Experts params	Experts	Routing 통신 오버헤드
ZeRO-1	Optimizer states	DP shard	Param 통신비 증가
ZeRO-2	Optimizer + Gradients		
ZeRO-3	Optimizer + Grad + Params		

8. 5D Parallelism in a Nutshell

K) Summary

- **DP**: 단순하지만 Batch 한계 존재
- **PP**: 모델을 **layer별**로 나누어 **깊은 네트워크** 처리
- **TP/SP**: 내부 **행렬 연산**을 쪼개지만, **통신이 critical path라 scale 제한**
- **CP**: 긴 Sequence 처리용, **attention 통신** 필요
- **EP**: **MoE** 기반 모델에서 전문가 병렬화, **All-to-All 통신** 필수
- **ZeRO-1/2/3**: **DP** 기반 메모리 최적화, shard 단계별로 통신 복잡도 증가

8. 5D Parallelism in a Nutshell

K) Summary

- **5D** Parallelism = **DP + TP + (SP/CP) + PP + EP**
- **ZeRO** = **DP** 기반의 메모리 절감 전략 (**1~3단계**)
- **TP/SP/EP** = **Layer "내부"** 연산/구조 병렬화
- **PP** = **Layer "간"** 단위 파이프라인
- **CP** = 긴 sequence 처리용 (Attention 나누기)
- **ZeRO-3 ↔ PP** 차이점: "**weight vs activation 통신**"의 대조
- **DeepSeek-V3**: PP + ZeRO-1 + EP 조합으로 초거대 모델을 효율적으로 학습