Diving into GPUs - Fusing, Threading, and Mixing

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

이승한 (Seunghan Lee)

Overview

공부할 내용: “**GPU 내부 구조**와 **병렬 실행**의 원리”

- 이전) **모델 병렬화**(예: DP, PP, TP 등)처럼 **“모델 수준”**의 분산 구조
- 앞으로) 훨씬 더 하드웨어 가까운, **GPU 내부 수준의 병렬화(fusion, threading, scheduling)**

1. A Primer on GPUs

1. A Primer on GPUs

A) GPU 구조의 기초

GPU 성능의 양쪽 축!

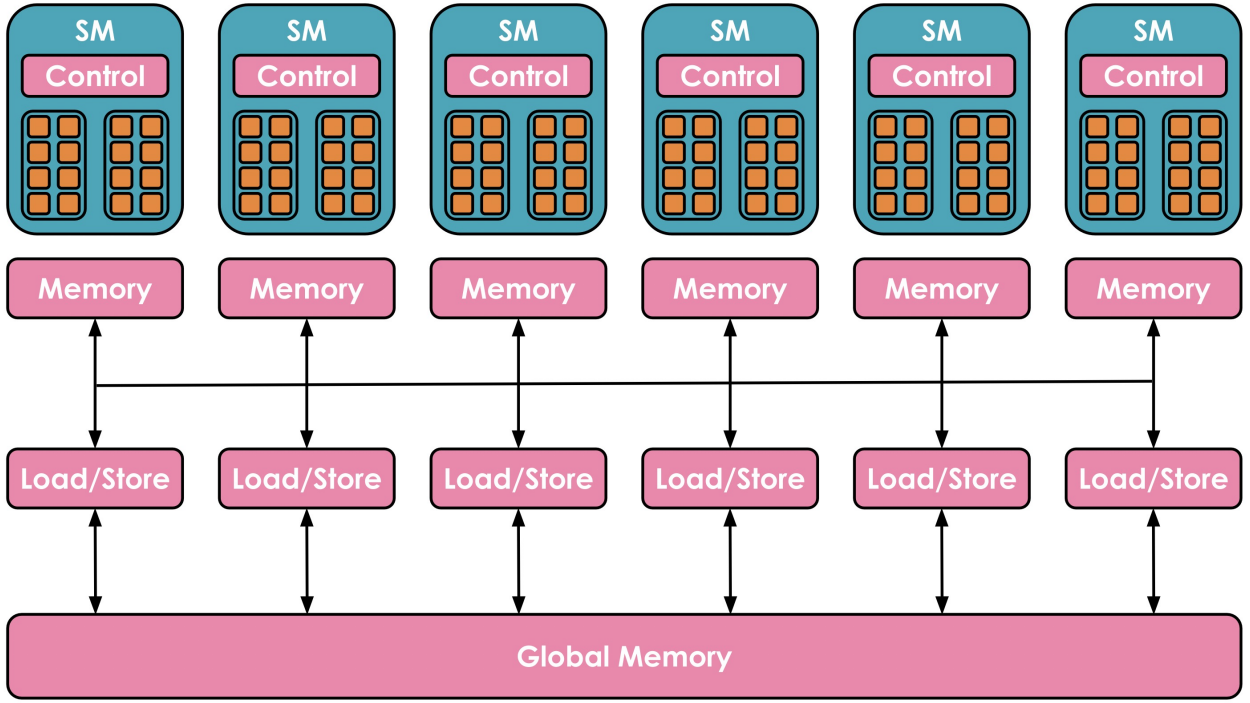
- [1] GPU의 **구성 요소** (Compute hierarchy) = “어디서 **연산**이 일어나는가”
    - 작은 **병렬 연산 유닛**을 수천 개 묶은 구조
        - GPU: 코어 수가 **많고** + **단순**
        - CPU: 코어 수가 **적고 + 똑똑**
- [2] GPU의 **메모리 계층** (Memory hierarchy) = “어디서 **데이터**를 가져오는가”

1. A Primer on GPUs

A) GPU 구조의 기초

- GPU는 “**병렬 처리용** 프로세서”
- 수천 개의 **작은 연산 유닛(코어)**을 가진 칩
- 대규모 **벡터/행렬 연산**을 **동시**에 수행하도록 설계
- 활용) 딥러닝에서 자주 등장하는 행렬곱, 컨볼루션, 벡터 연산을 빠르게 처리

1. A Primer on GPUs



B) 구성 요소: GPU의 계층적 구조

- GPU 내부는 **계층적 (hierarchical)**으로 구성
- 예시: NVIDIA H100

  → **132개의 SM** × **128 코어** = 총 16,896 코어

하드웨어적 구분	
(HW) **SM (Streaming Multiprocessor)**	- GPU의 **'작은 CPU'** - **독립적**으로 명령을 실행하는 단위
(HW) **Core (CUDA core, Tensor core)**	- **실제 연산 수행** 유닛 - **SM 안에 여러 개** 존재
(HW) **Control Logic (Scheduler)**	- SM 내부에서 **스케줄링/명령 제어** 담당

1. A Primer on GPUs

B) 구성 요소: GPU의 계층적 구조

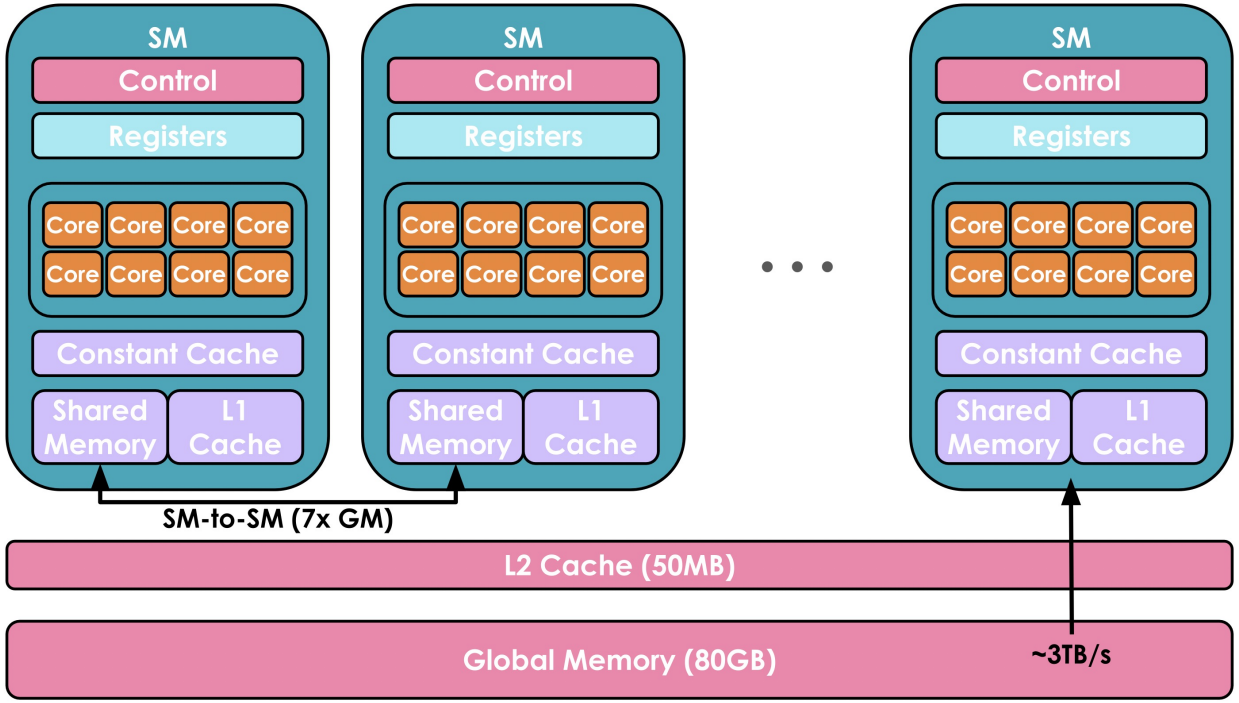
- GPU 내부는 **계층적 (hierarchical)**으로 구성

단위	역할	예시 수치 (H100)	병렬 단위
(HW) **GPU 전체**	**하나의 물리적 칩**	GPU 1개	수천 개의 연산
(HW) **SM (Streaming Multiprocessor)**	독립적인 연산 **클러스터**. 명령을 받아 여러 **코어를 제어**	132개	Block-level
(HW) **Core (CUDA / Tensor Core)**	덧셈, 곱셈 등의 **연산 수행**	SM당 128개	Thread-level
(SW) **Warp**	**32개의 thread** 묶음 (동일 명령 실행)	수천 개 동시	Instruction-level
(SW) **Thread**	**가장 작은 실행 단위** (데이터 한 점 계산)	수십만 개 활성	데이터

1. A Primer on GPUs

C) 메모리 계층: GPU의 메모리 계층 구조

- GPU의 메모리: **크기** & **접근 범위**에 따라 나뉨
- GPU 연산 최적화의 핵심:
  - (1) 느린 **Global** Memory 접근을 **최소화**하기!
  - (2) 가능한 **Shared** Memory를 **많이** 활용하기!



메모리 단계	공유 범위	속도	설명
Registers	개별 thread	**가장 빠름**	각 thread 전용의 임시 변수 저장
Shared Memory / L1 Cache	같은 SM의 thread끼리 공유	**빠름**	블록 내부의 데이터 교환에 사용
L2 Cache	모든 SM 간 공유	**보통**	SM 간 통신 및 캐싱 역할
Global Memory (VRAM)	전체 GPU	**느림**	GPU의 주 메모리 (예: 80GB H100)

1. A Primer on GPUs

D) GPU의 실행 단위: Kernel

- **Kernel** = GPU에서 실행되는 **코드 단위**
  - E.g., CUDA, Triton, cuBLAS 등에서 정의된 함수
- **CPU (Host)**가 **데이터를 준비**한 뒤 → **GPU**로 **"kernel 실행"**을 요청
- GPU에서는 **수천 개의 thread**가 이 kernel을 병렬 실행

1. A Primer on GPUs

E) Thread, Warp, Block

- GPU는 thread를 **단순히 1개씩 돌리지 않음**
- **[TWBG] Thread → Warp → Block → Grid** 의 구조로 묶어 병렬 실행
  - E.g., 한 **kernel** 호출 시 → 1024 thread/block × 132 SM ≈ **135,000 thread**가 동시에 실행

단위	설명
Thread	가장 작은 **실행 단위** (e.g., 하나의 행렬 원소를 계산)
Warp	**32개의 thread**가 묶인 단위 → 항상 **같은 명령**을 동시에 수행
Block	**여러 Warp**로 구성됨 (보통 512~1024 thread)
Grid	전체 GPU에서 실행되는 **모든 Block들의 집합**

1. A Primer on GPUs

F) Scheduling 방식

- SM: **여러 block**을 동시에 실행할 수 있음
- 단, Shared Memory, Register 수, Thread 수에 따라..
  - 실행 가능한 **block 개수가 제한**
  - 나머지 block은 **"대기(queue)"**에 들어갔다가, 자원이 남으면 실행
- Kernel의 성능 = Block/thread 구성을 얼마나 효율적으로 했는지

1. A Primer on GPUs

G) 코드 실행 흐름

- Step 1) **CPU(Host)** → **데이터** 준비 및 **GPU** 메모리로 복사
- Step 2) **CPU** → **GPU**에게 kernel 실행 요청
- Step 3) **GPU** → **SM**별로 Block 배정
- Step 4) 각 **SM** 내부 → **Warp** 단위로 **thread** 실행
  - Warp = Thread x 32개
  - Warp 내의 thread들은 동일 명령을 따름
- Step 5) **Thread**들은 동일 명령(=**SIMD**, Single Instruction Multiple Data) 수행
- Step 6) 연산 결과를 **global memory**로 저장

1. A Primer on GPUs

H) 최적화의 핵심

최적화 기법	설명
Fusing (커널 융합)	**여러 연산**(예: matmul + bias + relu)을 **하나의 kernel**로 합쳐서, memory read/write 감소
Threading (적절한 thread 구성)	**block/warp/thread 크기를 잘 맞춰서** SM 점유율(occupancy) 극대화
Mixing (다양한 정밀도 혼합)	FP32, FP16, BF16, FP8 등 **서로 다른 precision을 혼합**해 throughput 향상

1. A Primer on GPUs

I) Flash Attention과의 연결

- 기존 Attention = **Global Memory 접근**이 너무 잦았음
- FlashAttention = **GPU 메모리 계층 구조를 최대한 활용**한 예시!
    - L1/Shared Memory에 데이터를 캐싱해 Memory I/O를 줄임

1. A Primer on GPUs

J) Summary

- GPU는 **수천 개의 코어(SM 내부의 thread)**가 계층적 메모리 구조를 활용해 수천 개의 연산을 동시에 수행
- (빠름/작음) **Register** → **Shared Memory** → **L2** → **Global Memory** (느림/큼)
- **커널 최적화 (fusion, threading, mixed precision)**: 병렬성 & 계층적 메모리 구조를 얼마나 잘 활용하는지

계층	구성 요소	병렬 개념	주요 역할
GPU 전체	여러 SM	Block-level parallel	여러 block 병렬 실행
SM	여러 Core + Shared Memory	Warp-level parallel	block 내부 연산 담당
Core	여러 Thread	Thread-level parallel	실제 연산 수행
Memory	Register → Shared → L2 → Global	Memory hierarchy	빠른 캐시 우선 접근

2. Improving Performance with Kernels

2. Improving Performance with Kernels

Overview

“왜 빠른지→어떻게 쓰는지→언제 Triton/CUDA로 넘어가는지”

- 1) 왜 @torch.compile만 붙여도 빨라질까?
- 2) Triton은 뭐가 다른가?
- 3) Example: ELU
- 4) 언제 무엇을 쓰면 좋나? (난이도/속도/유연성)
- 5) 성능이 나오는 원리 (핵심 최적화 체크리스트)
- 6) Graph

2. Improving Performance with Kernels

Overview

“왜 빠른지→어떻게 쓰는지→언제 Triton/CUDA로 넘어가는지”

- **1) 왜 @torch.compile만 붙여도 빨라질까?**
- 2) Triton은 뭐가 다른가?
- 3) Example: ELU
- 4) 언제 무엇을 쓰면 좋나? (난이도/속도/유연성)
- 5) 성능이 나오는 원리 (핵심 최적화 체크리스트)
- 6) Graph

2. Improving Performance with Kernels

A) 왜 @torch.compile만 붙여도 빨라질까? - Brief

- PyTorch는 원래 **연산** 하나당 **커널** 하나를 호출
- E.g., **elu(x) = exp → sub → where** … 이 각각 별도 커널이면
  - 매번 Global Memory(느림)에서 읽고/쓰고
  - 커널 호출 오버헤드도 누적
- **@torch.compile(TorchDynamo+Inductor)**
  - **연산 그래프**를 “캡처 → 결합(fuse) → Triton 커널 생성” (여러 연산을 하나의 커널로 합쳐서 실행)
  - 중간 결과를 레지스터/SM의 **L1/Shared**에 머물게 하므로 **메모리 왕복이 급격히 감소**
  - 커널 호출 횟수도 크게 줄어듬!

2. Improving Performance with Kernels

A) 왜 @torch.compile만 붙여도 빨라질까? - Detail

- **(A-1) 전체 Pipeline 개요**
  - **@torch.compile(fn)**을 붙이면 PyTorch가 아래 단계를 수행
    - Step 1) 그래프 캡처 (TorchDynamo)
    - Step 2) Autograd 변환 (AOTAutograd)
    - Step 3) 코드 생성 (TorchInductor)
    - Step 4) 런타임 캐시 (Runtime Cache)

2. Improving Performance with Kernels

A) 왜 @torch.compile만 붙여도 빨라질까? - Detail

- Step 1) 그래프 캡처 (TorchDynamo)
- Step 2) Autograd 변환 (AOTAutograd)
- Step 3) 코드 생성 (TorchInductor)
- Step 4) 런타임 캐시 (Runtime Cache)

- **(A-1) 전체 Pipeline 개요**
  - Step 1) **그래프 캡처 (TorchDynamo)**
    - 한 줄 요약: **"파이썬 함수를 그래프로 바꿔서 컴파일할 준비를 하는 과정"**
    - 파이썬 함수 fn을 "트레이싱/분석"하여 **PyTorch IR(Graph)**로 바꿈
    - 파이썬 레벨의 반복문/분기 등은 최대한 **Tensor 연산 조합**으로 끌어내리려 함
    - 그래프에 포함되지 못한 부분이 있으면 **graph break(성능 저하 원인)** 발생

2. Improving Performance with Kernels

- Step 1) 그래프 캡처 (TorchDynamo)
- Step 2) Autograd 변환 (AOTAutograd)
- Step 3) 코드 생성 (TorchInductor)
- Step 4) 런타임 캐시 (Runtime Cache)

A) 왜 @torch.compile만 붙여도 빨라질까? - Detail

- **(A-1) 전체 Pipeline 개요**
  - Step 1) **그래프 캡처 (TorchDynamo)**
    - 예시: **f(x)= torch.sin(x) + torch.exp(x)**: 이걸 실행하면, Python 인터프리터가..
      - (1) torch.sin(x) 실행 → (2) 결과를 메모리에 저장 → (3) torch.exp(x) 실행 → (4) 둘을 더함
    - 즉, GPU 커널을 **매번 따로** 호출

      → 빠르지만, “코드의 전체 흐름”은 GPU가 모름

      → PyTorch는 “매번” 해석(interpreting) 중

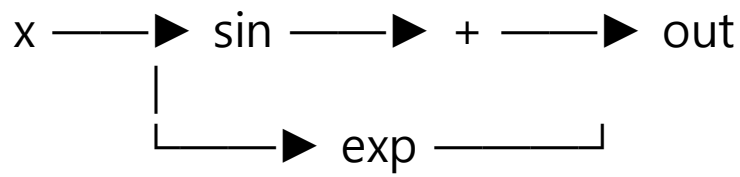
2. Improving Performance with Kernels

- Step 1) 그래프 캡처 (TorchDynamo)
- Step 2) Autograd 변환 (AOTAutograd)
- Step 3) 코드 생성 (TorchInductor)
- Step 4) 런타임 캐시 (Runtime Cache)

A) 왜 @torch.compile만 붙여도 빨라질까? - Detail

- **(A-1) 전체 Pipeline 개요**
  - Step 1) **그래프 캡처 (TorchDynamo)**
    - TorchDynamo의 역할: 이걸 **그래프로 캡처**한다는 것은?
      - 그래프 = **어떤 연산들이 있는지 (sin, add) + 어떤 데이터가 어디로 흐르는지**를 한 눈에 담은 도식
      - "PyTorch **코드의 흐름(연산 순서, 연결 관계)**을 한 번에 읽어서,

        GPU가 그대로 실행할 수 있는 **그래프 구조**로 바꾸는 것"



    - 내부적으로 Python 바이트코드를 가로채서 분석 (trace)
    - 이 그래프를 다음 단계 (= TorchInductor)가 받아서, 커널을 합치고 최적화

2. Improving Performance with Kernels

- Step 1) 그래프 캡처 (TorchDynamo)
- **Step 2) Autograd 변환 (AOTAutograd)**
- Step 3) 코드 생성 (TorchInductor)
- Step 4) 런타임 캐시 (Runtime Cache)

A) 왜 @torch.compile만 붙여도 빨라질까? - Detail

- **(A-1) 전체 Pipeline 개요**
  - Step 2) **Autograd 변환 (AOTAutograd)**
    - AOT(Ahead-Of-Time) Autograd의 핵심 요약
      - Dynamo가 만든 그래프를 **"미분 가능한 형태"**로 재구성
      - Forward/backward를 **"따로"** 컴파일하기 위한 준비
    - 불필요한 중간 텐서 저장을 최소화
    - Backward에서 재계산(recompute)할지 등 메모리/속도를 최적화

2. Improving Performance with Kernels

- Step 1) 그래프 캡처 (TorchDynamo)
- **Step 2) Autograd 변환 (AOTAutograd)**
- Step 3) 코드 생성 (TorchInductor)
- Step 4) 런타임 캐시 (Runtime Cache)

A) 왜 @torch.compile만 붙여도 빨라질까? - Detail

- **(A-1) 전체 Pipeline 개요**
  - Step 2) **Autograd 변환 (AOTAutograd)**
    - 주어진 함수의 **forward** 그래프 & 대응하는 **backward** (VJP) 그래프를 미리 **따로** 만들어, **컴파일 가능**하게 바꿔주는 단계.
      - Step 1) Dynamo가 만든 큰 계산 그래프를 받아서
      - Step 2) 미분에 필요한 부분만 깔끔하게 분리하고
      - Step 3) 저장 vs 재계산(re-materialization) 정책을 정해
      - Step 4) 결과적으로 **"컴파일 가능한 forward·backward 두 그래프"**를 뽑아줌
    - 이걸 Inductor가 받아서 커널로 최적화/생성

2. Improving Performance with Kernels

- Step 1) 그래프 캡처 (TorchDynamo)
- **Step 2) Autograd 변환 (AOTAutograd)**
- Step 3) 코드 생성 (TorchInductor)
- Step 4) 런타임 캐시 (Runtime Cache)

A) 왜 @torch.compile만 붙여도 빨라질까? - Detail

- **(A-1) 전체 Pipeline 개요**
  - Step 2) **Autograd 변환 (AOTAutograd)**
    - 필요한 이유? PyTorch Autograd는 보통 **런타임 중 테이프를 쌓고, backward 때 그걸 거꾸로 재생**
    - Note: 컴파일을 잘 하려면…
      - 정적인 그래프(ops, 텐서 의존 관계)가 필요하고
      - Forward/backward를 독립적으로 최적화할 수 있어야!
    - AOTAutograd는 이를 위해, 아래의 **각각 하나의 그래프**로 뽑아 Inductor에게 전달
      - **Forward**: 순전파 계산 + (backward에 필요한) “저장 텐서들” 산출
      - **Backward**: v = ∂L/∂y가 들어오면 VJP(vector-Jacobian product)로 ∂L/∂x, ∂L/∂θ를 계산

2. Improving Performance with Kernels

A) 왜 @torch.compile만 붙여도 빨라질까? - Detail

- Step 1) 그래프 캡처 (TorchDynamo)
- **Step 2) Autograd 변환 (AOTAutograd)**
- Step 3) 코드 생성 (TorchInductor)
- Step 4) 런타임 캐시 (Runtime Cache)

- **(A-1) 전체 Pipeline 개요**
  - Step 2) **Autograd 변환 (AOTAutograd)**

- 모델 $y = f(x, \theta)$, 손실 $L(y)$
- 역전파는 보통 $v = \frac{\partial L}{\partial y}$가 주어졌을 때

  $\frac{\partial L}{\partial x} = v^\top \frac{\partial f}{\partial x}, \quad \frac{\partial L}{\partial \theta} = v^\top \frac{\partial f}{\partial \theta}$
- **AOTAutograd**는 이 f의 **전방 그래프**와, 위 **VJP**를 수행하는 **후방 그래프**를 **미리 생성**

2. Improving Performance with Kernels

- Step 1) 그래프 캡처 (TorchDynamo)
- Step 2) Autograd 변환 (AOTAutograd)
- **Step 3) 코드 생성 (TorchInductor)**
- Step 4) 런타임 캐시 (Runtime Cache)

A) 왜 @torch.compile만 붙여도 빨라질까? - Detail

- **(A-1) 전체 Pipeline 개요**
  - Step 3) **코드 생성 (TorchInductor)**
    - 한 줄 요약: (앞서 Dynamo → AOTAutograd로 만든) PyTorch 그래프를 **GPU/CPU**에서 실제로 실행될 **저수준 코드(커널)** 로 바꿔주는 compiler 백엔드
    - 연산 그래프를 **연산 융합(fusion)**해서 낮은 수의 커널로 만듦
    - 백엔드로 GPU면 **Triton 커널**(그리고 필요시 cuBLAS 호출)들을 생성
    - 커널마다 타일링/스케줄/메모리 접근(연속 접근) 등을 자동으로 정함

2. Improving Performance with Kernels

A) 왜 @torch.compile만 붙여도 빨라질까? - Detail

- **(A-1) 전체 Pipeline 개요**
  - Summary
    - **TorchDynamo**: “무엇을 계산해야 하는지” 알아내고
    - **AOTAutograd**: “forward/backward를 어떻게 나눌지” 정리하고
    - **Inductor**: 그걸 “어떻게 가장 빠르게 실행할지” 실제 코드로 만듦

```
PyTorch 코드
   ↓
TorchDynamo → 그래프 캡처
   ↓
AOTAutograd → forward/backward 분리 + 저장 정책 결정
   ↓
TorchInductor
   ├── 연산 융합 (fusion)
   ├── 최적화 (타일링, 스케줄링, coalesced memory)
   ├── Triton 커널 생성 (GPU)
   └── cuBLAS/cuDNN 연동
```

2. Improving Performance with Kernels

A) 왜 @torch.compile만 붙여도 빨라질까? - Detail

- **(A-1) 전체 Pipeline 개요**
  - Step 4) **런타임 캐시 (Runtime Cache)**
    - **“이미 만들어 둔 빠른 커널을 다시 만들지 않고 재사용하자”**
    - 한 번 생성한 커널/그래프는 입력 shape/dtype 키로 캐싱.
    - 다음 호출부턴 준비된 커널 재사용 → 오버헤드 없이 빠르게 실행!

- Step 1) 그래프 캡처 (TorchDynamo)
- Step 2) Autograd 변환 (AOTAutograd)
- Step 3) 코드 생성 (TorchInductor)
- **Step 4) 런타임 캐시 (Runtime Cache)**

2. Improving Performance with Kernels

A) 왜 @torch.compile만 붙여도 빨라질까? - Detail

- **(A-1) 전체 Pipeline 개요**
  - Step 4) **런타임 캐시 (Runtime Cache)**
    - 처음 @torch.compile 함수를 실행할 때는,
      - TorchDynamo가 그래프를 만들고
      - AOTAutograd가 forward/backward를 분리하고
      - TorchInductor가 커널을 생성하느라

      시간이 좀 걸림 (= 컴파일 시간)

- Step 1) 그래프 캡처 (TorchDynamo)
- Step 2) Autograd 변환 (AOTAutograd)
- Step 3) 코드 생성 (TorchInductor)
- **Step 4) 런타임 캐시 (Runtime Cache)**

2. Improving Performance with Kernels

- Step 1) 그래프 캡처 (TorchDynamo)
- Step 2) Autograd 변환 (AOTAutograd)
- Step 3) 코드 생성 (TorchInductor)
- **Step 4) 런타임 캐시 (Runtime Cache)**

A) 왜 @torch.compile만 붙여도 빨라질까? - Detail

- **(A-1) 전체 Pipeline 개요**
  - Step 4) **런타임 캐시 (Runtime Cache)**
    - 하지만 딱 **한 번만** 하면 끝!
    - 이후에는 그 함수가 **같은 형태의 입력**(shape, dtype 등)을 받으면,

      이미 만든 커널을 **캐시(cache) 에서 꺼내서** 바로 실행
    - 두 번째 실행부터는 파이썬 해석·컴파일 과정이 모두 생략되어서,

      마치 “이미 빌드된 바이너리”처럼 즉시 빠르게 동작

2. Improving Performance with Kernels

- Step 1) 그래프 캡처 (TorchDynamo)
- Step 2) Autograd 변환 (AOTAutograd)
- Step 3) 코드 생성 (TorchInductor)
- **Step 4) 런타임 캐시 (Runtime Cache)**

A) 왜 @torch.compile만 붙여도 빨라질까? - Detail

- **(A-1) 전체 Pipeline 개요**
  - Step 4) **런타임 캐시 (Runtime Cache)**
    - 캐시 키(저장 기준): PyTorch는 입력의 다음 정보를 기준으로 구분
      - **shape**: 텐서 크기 (예: [32, 128])
      - **dtype**: 데이터 타입 (float16, float32 등)
      - **device**: 실행 장치 (GPU 0, GPU 1 등)
    - 이 세 가지가 동일하면 “같은 커널”을 재사용할 수 있음

→ 그래서 동적 shape이 너무 자주 바뀌면 매번 새 커널을 만들어야 하므로 속도가 떨어짐

2. Improving Performance with Kernels

A) 왜 @torch.compile만 붙여도 빨라질까? - Detail

- **(A-1) 전체 Pipeline 개요**
  - Step 4) **런타임 캐시 (Runtime Cache)**
    - Step 1) 처음 실행 → 그래프 분석 + 커널 생성 (**조금 느림**)
    - Step 2) 결과를 (shape, dtype, device) 키로 **캐시에 저장**
    - Step 3) 다음 실행 → 캐시에서 커널 로드 → 바로 GPU 실행 (**매우 빠름**)

- Step 1) 그래프 캡처 (TorchDynamo)
- Step 2) Autograd 변환 (AOTAutograd)
- Step 3) 코드 생성 (TorchInductor)
- **Step 4) 런타임 캐시 (Runtime Cache)**

2. Improving Performance with Kernels

A) 왜 @torch.compile만 붙여도 빨라질까? - Detail

- **(A-2) 어디에서 시간이 절약되는지?**
  - **a) 커널 호출 수 급감 (Launch Overhead↓)**
    - 기본 PyTorch: 연산마다 커널을 호출
    - **@torch.compile**: 여러 연산을 **한 커널로 묶어** 호출 수를 크게 줄임
    - GPU 커널 런치는 비싸기 때문에, 호출 수 감소만으로도 큰 이득
  - **b) 메모리 왕복 감소 (Global Memory I/O↓)**
    - 기본: exp → sub → where 같은 중간 결과를 **매번 Global Memory에 쓰고 다시 읽음**
    - **Fusion**: 중간값을 레지스터/SM L1/Shared 에만 머물게 함 → **Global 왕복 제거**
    - 메모리 대역폭이 병목인 pointwise/elementwise 연산에서 효과가 특히 큼

2. Improving Performance with Kernels

A) 왜 @torch.compile만 붙여도 빨라질까? - Detail

- **(A-2) 어디에서 시간이 절약되는지?**
  - **c) 연산 재배치 & 공통부분 제거 (CSE, Reordering)**
    - 그래프 최적화 중에 공통 부분식 제거, 브로드캐스트 정리, 레아웃 변환 최소화 등 수행.
    - E.g. 동일한 exp(x)가 여러 곳에서 쓰이면 한 번만 계산해서 공유
  - **d) Autograd 경로 최적화**
    - 불필요한 중간 텐서 저장을 줄임 & 필요 시 backward에서 재계산하여 메모리 절약
    - 메모리 절약은 곧 캐시 적중률↑ / 스왑↓ → 속도 향상으로 이어짐.

2. Improving Performance with Kernels

A) 왜 @torch.compile만 붙여도 빨라질까? - Detail

- **(A-2) 어디에서 시간이 절약되는지?**
  - **c) 벡터화/연속 접근 정렬 (Coalesced Access)**
    - Inductor가 생성하는 Triton 커널은 인접 thread가 연속 주소를 접근하도록 인덱스 배치
    - 덕분에 메모리 병렬 대역폭을 최대한 활용

2. Improving Performance with Kernels

A) 왜 @torch.compile만 붙여도 빨라질까? - Detail

- **(A-3) 언제 효과가 큰지?**
  - **Elementwise 연산이 많은 커스텀 함수**
    - norm, 활성함수 조합, 잦은 reshape/where/broadcast 등
  - **연산 간 데이터 이동이 많은 파이프라인**
    - 중간 텐서가 큰 경우일수록 Fusion 효과↑
  - **반복 호출되는 함수**
    - JIT 준비비용이 1~2회 호출 후 암티즈되어 순수 실행이 매우 빨라짐

2. Improving Performance with Kernels

A) 왜 @torch.compile만 붙여도 빨라질까? - Detail

- **(A-4) 사용 방법**

```
@torch.compile
def my_fn(x, w):
    # 파이토치 연산을 자연스럽게 작성
    return (x @ w).relu().mul_(0.5) + x.sin()
```

```
model = torch.compile(model, mode="max-autotune")  # 속도 중심 탐색
# mode: "default", "reduce-overhead", "max-autotune" 등
```

2. Improving Performance with Kernels

Overview

“왜 빠른지→어떻게 쓰는지→언제 Triton/CUDA로 넘어가는지”

- 1) 왜 @torch.compile만 붙여도 빨라질까?
- **2) Triton은 뭐가 다른가?**
- 3) Example: ELU
- 4) 언제 무엇을 쓰면 좋나? (난이도/속도/유연성)
- 5) 성능이 나오는 원리 (핵심 최적화 체크리스트)
- 6) Graph

2. Improving Performance with Kernels

B) Triton은 뭐가 다른가?

- **(B-1) Triton이란**
  - “CUDA 대신 쓸 수 있는 **파이썬 기반 GPU 커널 언어**”
  - PyTorch 2.x의 **@torch.compile(Inductor)**이 **자동으로 Triton 커널을 생성**
  - 혹은, 우리가 직접 Triton으로 커널을 써서 더 미세하게 튜닝할 수도 있음
    - Inductor가 만들어 준 커널을 출발점으로 가져와 살짝 손보면,
      “마지막 5~20%”까지 성능을 끌어올릴 여지 있음

2. Improving Performance with Kernels

B) Triton은 뭐가 다른가?

- **(B-1) Triton이란**
  - **PyTorch**: 고수준 텐서 API **(쉬움 + 느림)**
  - **@torch.compile**: 자동 최적화 + 자동 Triton 커널 생성 **(쉬움 + 빠름)**
  - **Triton**: 직접 커널 작성(파이썬)으로 블록/메모리 패턴 최적화 **(어렵지 않음 + 더 빠름)**
    - CUDA에서 “grid/block/thread”를 직접 다루던 걸, Triton은 “프로그램(=블록) 단위”로 단순화!

    → Thread 레벨을 세세히 건드리지 않아도 OK
  - **CUDA(C++)**: low-level 전부 제어 **(어려움 + 아주 빠름)**

2. Improving Performance with Kernels

B) Triton은 뭐가 다른가?

- **(B-2) Triton의 프로그래밍 모델**
  - **a) 프로그램 당 블록 1개**: Triton 커널은 “한 프로그램”이 데이터의 일부분(타일/블록) 을 처리
  - **b) 그리드 (grid)**: 이런 프로그램이 여러 개 실행되어 전체 데이터를 커버
  - **c) program_id(dim)**: 몇 번째 블록인지 알려주는 ID (예: tl.program_id(0), tl.program_id(1)).
  - **d) BLOCK_SIZE / num_warps / num_stages**: 하나의 블록이 처리할 요소 수와 실행 리소스를 결정
  - **e) 마스킹(mask)**: 경계(out-of-bounds) 처리를 위해 mask로 안전하게 load/store
  - **f) 메모리 접근**: tl.load, tl.store로 연속(coalesced) 접근을 만들면 대역폭을 잘 사용

2. Improving Performance with Kernels

B) Triton은 뭐가 다른가?

- **(B-2) Triton의 프로그래밍 모델**
  - **a) 프로그램 당 블록 1개**: Triton 커널은 "**한 프로그램**"이 데이터의 **일부분(타일/블록)** 을 처리
    - Triton에서 @triton.jit 커널은 **"프로그램(program)"**이라는 최소 실행 단위로 돌아감
    - 전체를 한 번에 다 하지 않고, **조각내서** 여러 프로그램이 병렬로 처리
    - 비유) 큰 피자를 100조각으로 잘라서, 100명의 사람이 각자 자기 조각만 먹는 느낌.

      → "사람 1명 = 프로그램 1개", "조각 = 타일/블록"

```
pid = tl.program_id(0)  # 이 프로그램의 ID (0번째 조각, 1번째 조각, ...)
# pid가 다르면 서로 다른 조각(인덱스 구간)을 맡습니다.
```

2. Improving Performance with Kernels

B) Triton은 뭐가 다른가?

```
grid = lambda META: ((n_elements + BLOCK_SIZE - 1) // BLOCK_SIZE,)  # 프로그램 개수
kernel[grid](...)  # 이만큼의 프로그램이 뜸
```

- **(B-2) Triton의 프로그래밍 모델**
  - **b) 그리드 (grid)**: 이런 프로그램이 여러 개 실행되어 전체 데이터를 커버
    - "그리드" = **프로그램들의 배열**
    - "그리드" x M개 = **프로그램들의 배열** x M개 = **전체 프로그램**
    - 우리가 런처에서 **"총 몇 개의 프로그램을 띄울지"**를 지정하면,
      Triton이 그 개수만큼 프로그램을 동시에/반복적으로 실행해서 전체 데이터 범위를 덮음
      - 1D 그리드: 길이 N짜리 벡터를 BLOCK_SIZE만큼 쪼개서 ceil_div(N, BLOCK_SIZE)개의 프로그램 실행
      - 2D 그리드: 행렬처럼 (행, 열) 2차원 타일을 처리
    - e.g., 1000개의 프로그램을 10개의 프로그램(block) x 100의 block size로 처리

2. Improving Performance with Kernels

B) Triton은 뭐가 다른가?

- **(B-2) Triton의 프로그래밍 모델**
  - **c) program_id(dim)**: 몇 번째 블록인지 알려주는 ID (예: tl.program_id(0), tl.program_id(1)).
    - 1D 그리드: pid = tl.program_id(0)
    - 2D 그리드: pid_m = tl.program_id(0), pid_n = tl.program_id(1) 처럼 행/열 타일의 좌표가 됨.

예시 (1D)

python

```
pid = tl.program_id(0)
offsets = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)  # 내가 맡은 인덱스들
```

→ pid=0이면 0..BLOCK_SIZE-1, pid=1이면 BLOCK_SIZE..2*BLOCK_SIZE-1 …

예시 (2D; 행렬 타일)

python

```
pid_m = tl.program_id(0)  # 타일의 행 인덱스
pid_n = tl.program_id(1)  # 타일의 열 인덱스
row_offsets = pid_m*BM + tl.arange(0, BM)[:, None]
col_offsets = pid_n*BN + tl.arange(0, BN)[None, :]
# 이 타일이 담당하는 (행,열) 좌표 집합
```

2. Improving Performance with Kernels

B) Triton은 뭐가 다른가?

- **(B-2) Triton의 프로그래밍 모델**
  - **d) BLOCK_SIZE / num_warps / num_stages**: 하나의 블록이 처리할 요소 수와 실행 리소스를 결정
    - BLOCK_SIZE: 한 프로그램이 처리할 요소(타일)의 크기
      → 크면 한 번에 많이 처리하지만, 레지스터/캐시 압박↑ → 동시성↓ 가능.
    - num_warps: 프로그램 내부 병렬성. 보통 2, 4, 8 등을 시도 → 너무 크면 자원 과다 사용.
    - num_stages: Pipelining 수준(데이터 프리페치/중첩). 1~4 정도 시도.

2. Improving Performance with Kernels

B) Triton은 뭐가 다른가?

- (B-2) Triton의 프로그래밍 모델
  - d) BLOCK_SIZE / num_warps / num_stages: 하나의 블록이 처리할 요소 수와 실행 리소스를 결정

```python
@triton.jit
def kernel(x_ptr, y_ptr, n, BLOCK_SIZE: tl.constexpr):
    pid = tl.program_id(0)
    offs = pid*BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)
    ...
# 런처에서:
kernel[grid](x, y, n, BLOCK_SIZE=1024, num_warps=4, num_stages=2)
```

감각

- 작은 BLOCK_SIZE → 런치/오버헤드↑, 메모리 대역폭 활용↓
- 너무 큰 BLOCK_SIZE → 레지스터/Shared 사용↑ → 동시 실행 가능한 프로그램 수(점유율)↓
- **적당한 균형점**이 있음 → 그래서 튜닝이 중요!

2. Improving Performance with Kernels

B) Triton은 뭐가 다른가?

- **(B-2) Triton의 프로그래밍 모델**
  - **e) 마스킹(mask)**: 경계(out-of-bounds) 처리를 위해 mask로 안전하게 load/store
    - 보통 N이 BLOCK_SIZE의 배수가 아님.

      → 따라서, 마지막 프로그램은 범위를 살짝 넘는 인덱스가 생기게 됨(Out-of-Bounds).
    - 해결책: “마스크”를 쓰면 경계 안전하게 처리 가능.
    - CUDA에서 직접 if-guard를 매번 써야 하는 일을 Triton은 마스크 인자로 간단히 해결

```
offs = pid*BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)
mask = offs < n_elements
x = tl.load(x_ptr + offs, mask=mask)    # 범위 밖은 읽지 않음
...
tl.store(y_ptr + offs, y, mask=mask)    # 범위 밖은 쓰지 않음
```

2. Improving Performance with Kernels

B) Triton은 뭐가 다른가?

- **(B-2) Triton의 프로그래밍 모델**
  - **f) 메모리 접근**: tl.load, tl.store로 연속(coalesced) 접근을 만들면 대역폭을 잘 사용
    - GPU 메모리는 **연속 주소**를 여러 쓰레드가 한 번에 읽을 때 **대역폭을 최대로 뽑음**

    → 이를 **coalesced access(연속 접근)** 라고 함

좋은 패턴(연속)

python 복사

```
offs = pid*BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)  # 0,1,2,3,... 연속
x = tl.load(x_ptr + offs, mask=mask)              # 인접 쓰레드가 인접 주소를 읽음
```

나쁜 패턴(비연속/stride가 크거나 꼬임)

python 복사

```
offs = pid*BLOCK_SIZE + stride * tl.arange(0, BLOCK_SIZE)  # stride가 크면 멀리 떨어짐
x = tl.load(x_ptr + offs, mask=mask)  # 대역폭 활용 저하
```

2D에서의 연속 접근:

행렬에서 “행 우선” 메모리 레이아웃(C-Contiguous)라면,

- 열 방향 스캔은 stride가 큼 → 비연속
- 행 방향 스캔은 stride=1 → 연속

2. Improving Performance with Kernels

B) Triton은 뭐가 다른가?

- **(B-3) Example**

- (1)(2): 전체 데이터를 BLOCK_SIZE 단위로 쪼개, 그리드로 커버
- (3): program_id(0)로 “내 조각”의 시작 오프셋 계산
- (4): BLOCK_SIZE/num_warps/num_stages로 성능 튜닝
- (5): 마스크로 OOB 방지, 분기 최소화
- (6): 연속 접근(coalesced)로 대역폭 극대화

```
@triton.jit
def elu_kernel(x_ptr, y_ptr, n_elements,
               BLOCK_SIZE: tl.constexpr, alpha: tl.constexpr):
    pid = tl.program_id(0)                                  # (3) 이 프로그램의 ID
    offs = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)      # (1)(3)(4) 내가 맡을 인덱스들
    mask = offs < n_elements                                # (5) 경계 안전

    x = tl.load(x_ptr + offs, mask=mask)                    # (6) coalesced 읽기
    y = tl.where(x < 0, alpha * (tl.exp(x) - 1), x)         # 연산
    tl.store(y_ptr + offs, y, mask=mask)                    # (6) coalesced 쓰기
```

```
grid = lambda META: ((n + BLOCK_SIZE - 1)//BLOCK_SIZE,)
elu_kernel[grid](x, y, n, BLOCK_SIZE=1024, alpha=1.0,
                 num_warps=4, num_stages=2)
```

2. Improving Performance with Kernels

B) Triton은 뭐가 다른가?

- (B-4) Summary
  - **프로그램** = 타일(조각) 처리자
  - **그리드** = 프로그램들의 배열 (전체 범위 커버)
  - **program_id(dim)** = 내가 담당할 타일 좌표
  - **BLOCK_SIZE/num_warps/num_stages** = 성능 하이퍼파라미터 (튜닝 중요)
  - **마스크** = 경계 안전 + 분기 최소화
  - **coalesced 접근** = 인접 쓰레드가 인접 주소를 읽도록 인덱싱 → 대역폭 최대 활용

2. Improving Performance with Kernels

Overview

“왜 빠른지→어떻게 쓰는지→언제 Triton/CUDA로 넘어가는지”

- 1) 왜 @torch.compile만 붙여도 빨라질까?
- 2) Triton은 뭐가 다른가?
- **3) Example: ELU**
- 4) 언제 무엇을 쓰면 좋나? (난이도/속도/유연성)
- 5) 성능이 나오는 원리 (핵심 최적화 체크리스트)
- 6) Graph

2. Improving Performance with Kernels

C) Example: ELU

- **Overview**
  - C-1) Pytorch
  - C-2) @torch.compile
  - C-3) Triton (수동 커널)

2. Improving Performance with Kernels

C) Example: ELU

- (C-1) Pytorch
  - 쉽지만, 연산이 쪼개져 여러 커널이 실행될 가능성↑

```python
def elu(x, alpha=1.0):
    return torch.where(x < 0, alpha * (torch.exp(x) - 1), x)
```

2. Improving Performance with Kernels

C) Example: ELU

- **(C-2) @torch.compile**
  - 같은 코드인데 그래프 캡처 → 연산 결합 → Triton 커널 자동 생성
  - 환경변수 TORCH_LOGS=output_code 로 생성된 Triton 커널을 확인 가능

```python
@torch.compile
def elu(x, alpha=1.0):
    return torch.where(x < 0, alpha * (torch.exp(x) - 1), x)
```

2. Improving Performance with Kernels

C) Example: ELU

- **(C-3) Triton (수동 커널)**
  - 블록 크기(BLOCK_SIZE), 그리드 크기(grid = ceil_div(N, BLOCK_SIZE))를 우리가 정함
  - 메모리 연속 인덱스로 읽고/쓰면 coalesced access가 되어 대역폭 효율↑

python

```python
@triton.jit
def elu_kernel(input_ptr, output_ptr, num_elements, BLOCK_SIZE: tl.constexpr):
    pid = tl.program_id(0)                          # 블록 ID
    offs = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)
    mask = offs < num_elements
    x = tl.load(input_ptr + offs, mask=mask)
    y = tl.where(x < 0, tl.exp(x) - 1.0, x)         # alpha=1 버전
    tl.store(output_ptr + offs, y, mask=mask)
```

2. Improving Performance with Kernels

Overview

“왜 빠른지→어떻게 쓰는지→언제 Triton/CUDA로 넘어가는지”

- 1) 왜 @torch.compile만 붙여도 빨라질까?
- 2) Triton은 뭐가 다른가?
- 3) Example: ELU
- **4) 언제 무엇을 쓰면 좋나? (난이도/속도/유연성)**
- 5) 성능이 나오는 원리 (핵심 최적화 체크리스트)
- 6) Graph

2. Improving Performance with Kernels

D) 언제 무엇을 쓰면 좋나

- 순서 1) **PyTorch**로 정확한 참조 구현
- 순서 2) **@torch.compile**로 쉽게 1차 가속
- 순서 3) 성능이 목표 대비 부족하면, 자동 생성된 **Triton 커널**을 바탕으로 수동 튜닝
- 순서 4) 그래도 병목이면 **CUDA**로 low-level 최적화(shared mem tile, bank conflict, occupancy 등)

도구	난이도	속도	유연성	코멘트
PyTorch	가장 쉬움	느림	낮음	프로토타입/레퍼런스
@torch.compile	쉬움	빠름	중간	첫 번째 시도로 강력 추천
Triton	중간	더 빠름	높음	블록/타일/메모리 패턴을 직접 최적화
CUDA(C++)	어려움	최고	최고	Shared/레지스터/스케줄 세밀 제어, 유지보수 비용 큼

2. Improving Performance with Kernels

Overview

“왜 빠른지→어떻게 쓰는지→언제 Triton/CUDA로 넘어가는지”

- 1) 왜 @torch.compile만 붙여도 빨라질까?
- 2) Triton은 뭐가 다른가?
- 3) Example: ELU
- 4) 언제 무엇을 쓰면 좋나? (난이도/속도/유연성)
- **5) 성능이 나오는 원리 (핵심 최적화 체크리스트)**
- 6) Graph

2. Improving Performance with Kernels

E) 성능이 나오는 원리

- Overview
  - E-1) 메모리 접근
  - E-2) 병렬/스케줄
  - E-3) 수치/정밀도
  - E-4) 벤치마킹 습관

2. Improving Performance with Kernels

E) 성능이 나오는 원리

- **(E-1) 메모리 접근**
  - **Coalesced access**: 인접 thread가 **연속 주소**를 읽도록 인덱싱
  - **Fusion**: 중간 텐서를 Global Memory에 쓰지 말고 **Register/Shared memory에 유지**
  - **Shared Memory 활용**: 반복 접근 데이터를 tile로 올려서 **재사용**
  - **Vectorized I/O**: float2/float4 같은 **벡터 로드/스토어**로 대역폭 활용
- **(E-2) 병렬/스케줄**
  - **Block/Grid 크기 튜닝**: 장치별 sweet spot (예: 128, 256, 512, 1024) 실험
  - **Occupancy (점유율)**: Register/Shared 사용량이 너무 커지면 동시 실행 Block 수↓ → 오히려 느려질 수!
  - **분기(divergence) 최소화**: 같은 warp(32 threads)에서 if-else 갈라지면 직렬화됨 → tl.where/마스크 사용

2. Improving Performance with Kernels

Overview

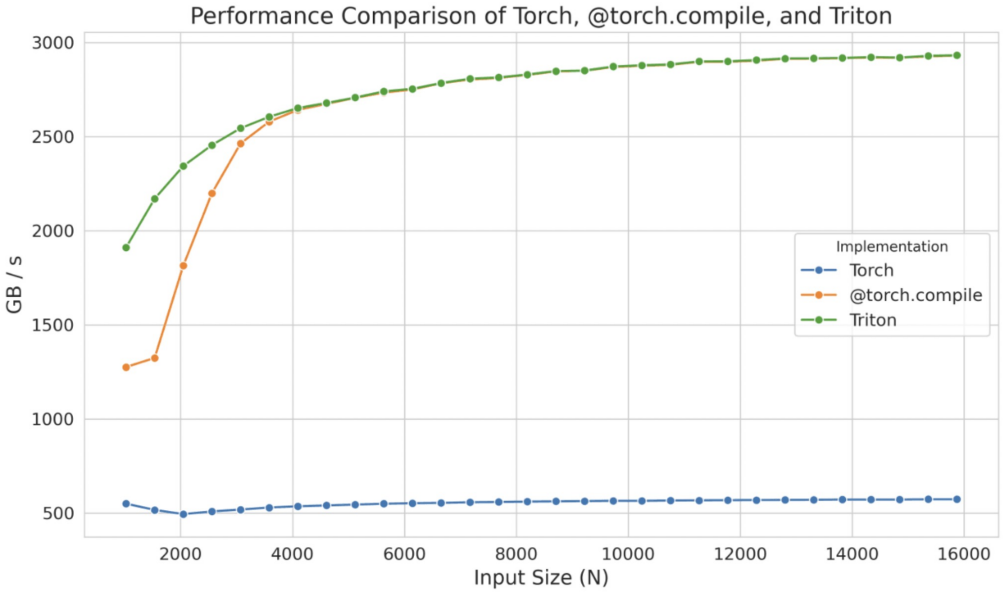
“왜 빠른지→어떻게 쓰는지→언제 Triton/CUDA로 넘어가는지”

- 1) 왜 @torch.compile만 붙여도 빨라질까?
- 2) Triton은 뭐가 다른가?
- 3) Example: ELU
- 4) 언제 무엇을 쓰면 좋나? (난이도/속도/유연성)
- 5) 성능이 나오는 원리 (핵심 최적화 체크리스트)
- **6) Graph**

2. Improving Performance with Kernels

F) Graph

- 그래프 1: **파란선(torch)** vs **주황선(@torch.compile)**
  - 동일 ELU를 @torch.compile만 붙였는데 GB/s 5배 이상↑
  - N이 커질수록 주황선이 장비 최대 대역폭에 근접 (성능 포화)
- 그래프 2: 여기에 **초록선(Triton)** 추가
  - 주황선과 거의 같거나, 작은 N에서 더 좋음
  - 큰 N에서는 둘 다 거의 최대치 도달 (병목=메모리 대역폭)



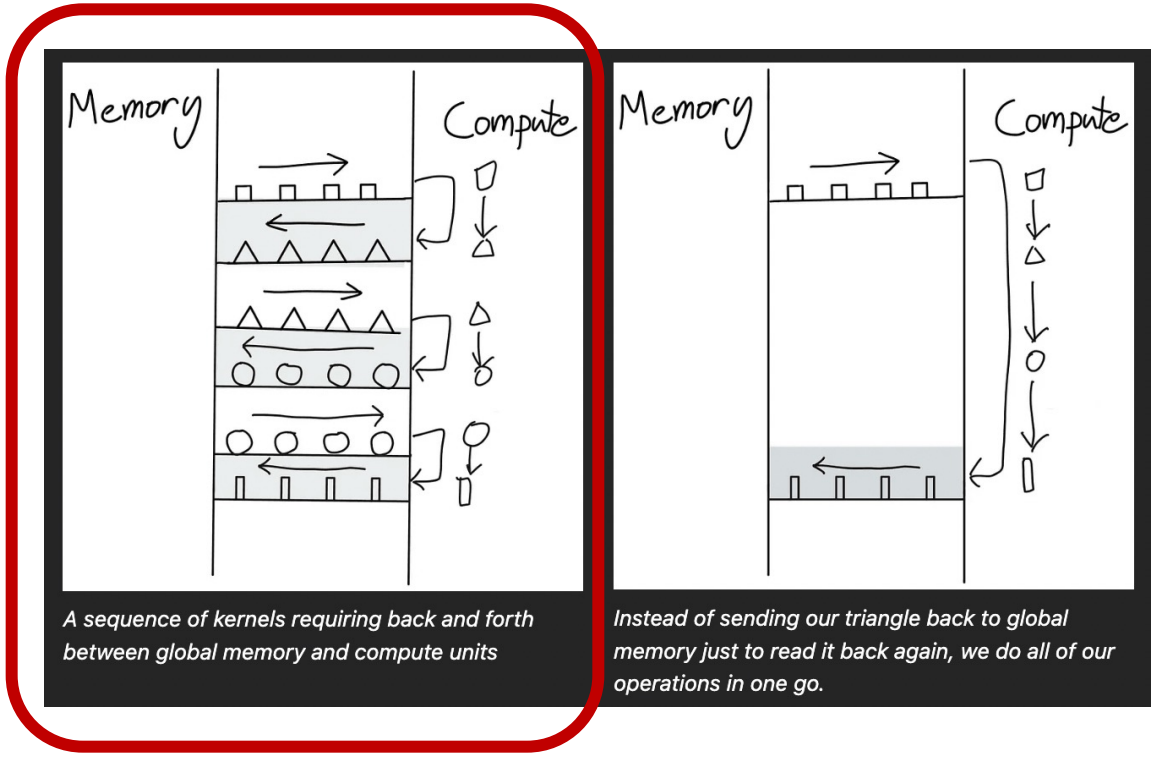
3. Fused Kernels

3. Fused Kernel

- **“커널 융합(Fused Kernels)”**을 시각적으로 설명한 예시
- GPU가 왜 ‘**여러 연산을 한 번에 묶는 것**’이 빠른지

[비효율적인 방식] (커널이 따로따로)

- Memory: GPU의 **Global Memory (가장 느린 영역)**
- Compute: 실제 연산이 일어나는 SM

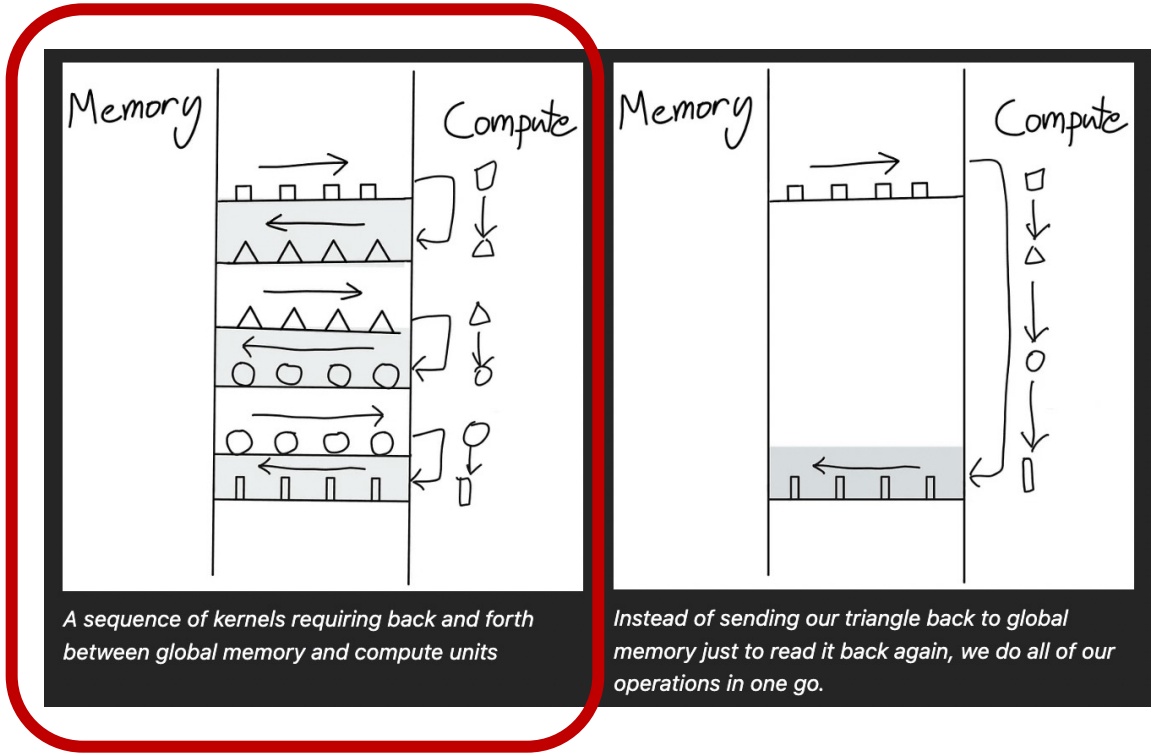


3. Fused Kernel

[비효율적인 방식] (커널이 따로따로)

- 각 연산(네모/세모/동그라미)이 서로 다른 커널로 실행
- Procedure
  - 한 연산이 끝나면 결과를 Global Memory에 저장 (→)
  - 다음 연산에서 다시 그걸 읽어옴 (←)
- 즉, **연산-메모리-연산-메모리 왕복**이 계속 일어남
- CPU와 GPU가 계속 **"명령 주고받기"** 때문에

→ 메모리 왕복 (latency), 런치 오버헤드가 엄청 커짐



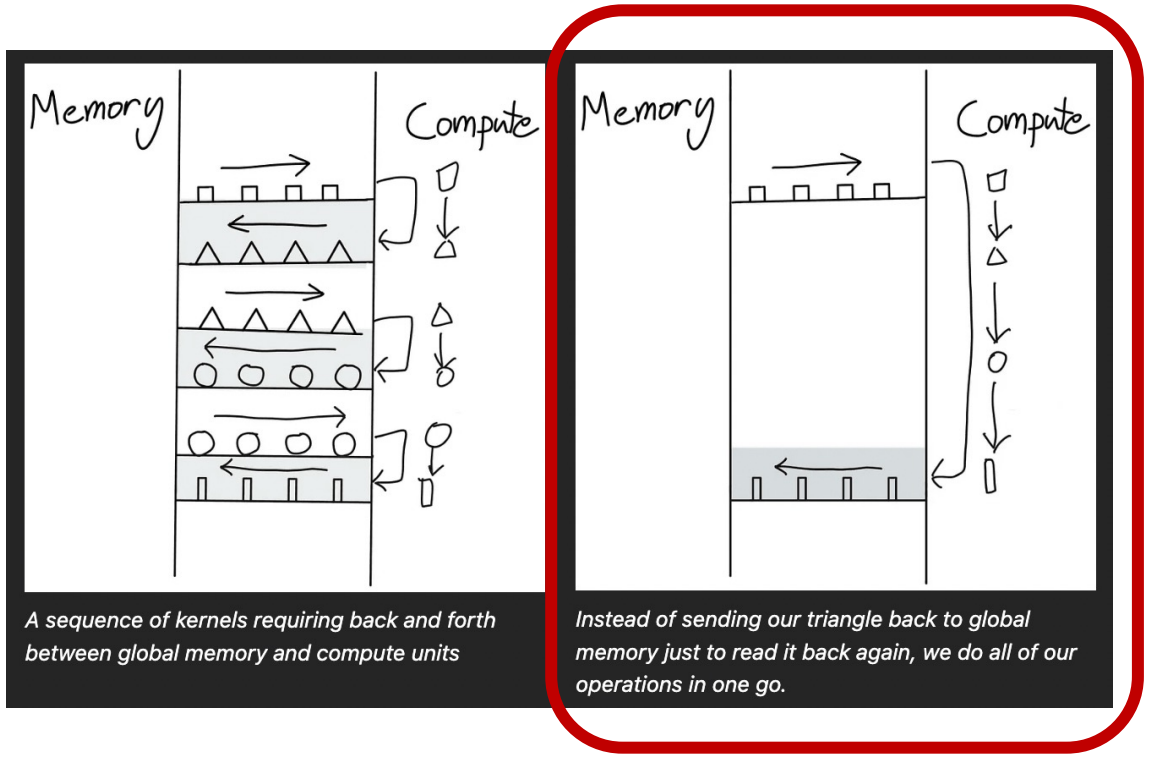
3. Fused Kernel

[효율적인 방식] (Fused Kernel)

- 여러 연산을 **하나의 커널로 묶어서** 한 번에 실행
- 한 번 Global Memory에서 데이터를 가져오면

  → 연산 A → 연산 B → 연산 C …
- 전부 **SM 내부의 빠른 메모리 (L1/Shared/Register)** 안에서 처리하고, 마지막에만 Global Memory로 결과를 저장함

“데이터는 한 번만 들여오고, 필요한 계산을 다 끝내고 나서 내보낸다.”



3. Fused Kernel

Summary

구분	Fusion X	Fusion O
커널 개수	여러 개 (각 연산마다)	**1개** (모든 연산을 **한 번에**)
메모리 왕복	많음 (매번 읽고 씀)	**1회** (맨 처음/맨 끝만)
속도	느림 (I/O 병목)	**빠름** (계산 집중형)
GPU 활용	낮음	**높음**
코드 예시	torch.exp(x); torch.sin(x); torch.add(...);	**@torch.compile** or **Triton fused kernel**

3. Fused Kernel

Example: Transformer에서의 예시

```
y = LayerNorm(x)
y = y * scale + bias
y = GELU(y)
```

- 각각 커널로 돌리면 매번 메모리 왕복 발생
- 하지만 Fused Kernel에서는:
  - **한 번에** LayerNorm + Scale + Bias + GELU 를 계산
  - **중간 결과를 메모리에 저장하지 않음**

    → GPU 내부 캐시에서 바로 다음 연산으로 연결
- 이게 LayerNorm/GELU 등에서 PyTorch가. **@torch.compile** 로 엄청 빨라지는 이유

4. Flash Attention

4. Flash Attention

Overview

- FlashAttention이 왜 필요한가
- 핵심 아이디어: Tiling + Online Softmax + Kernel Fusion
- 알고리즘 (Pseudocode)
- 이점/제약
- v1/v2/v3

4. Flash Attention

Overview

- **FlashAttention이 왜 필요한가**
- 핵심 아이디어: Tiling + Online Softmax + Kernel Fusion
- 알고리즘 (Pseudocode)
- 이점/제약
- v1/v2/v3

4. Flash Attention

A) FlashAttention이 왜 필요한가

- 병목: **HBM 왕복**

$$S = QK^\top (\text{크기} N \times N) \text{와} P = \text{softmax}(S)$$

- Naïve attention: S를 **HBM(글로벌 메모리) 에 "완전히" 만들어 두고** 다음 연산에 사용
- 한계점: GPU의 속도는 "연산"보다 **"메모리 왕복"**(HBM ↔ SM/레지스터)이 병목!
  - **SRAM/레지스터 (온칩)**: 매우 빠르지만 작음(수 MB~수십 MB).
  - **HBM (오프칩)**: 크지만 상대적으로 느림.
- 길이 N이 커지면 S가 N^2이라 엄청 큼

  → 계속 HBM에 쓰고 다시 읽는 비용이 압도적.
- 핵심 문제: **"큰 S와 P를 HBM에 만들었다가/읽었다가 하는 왕복이 병목"**

4. Flash Attention

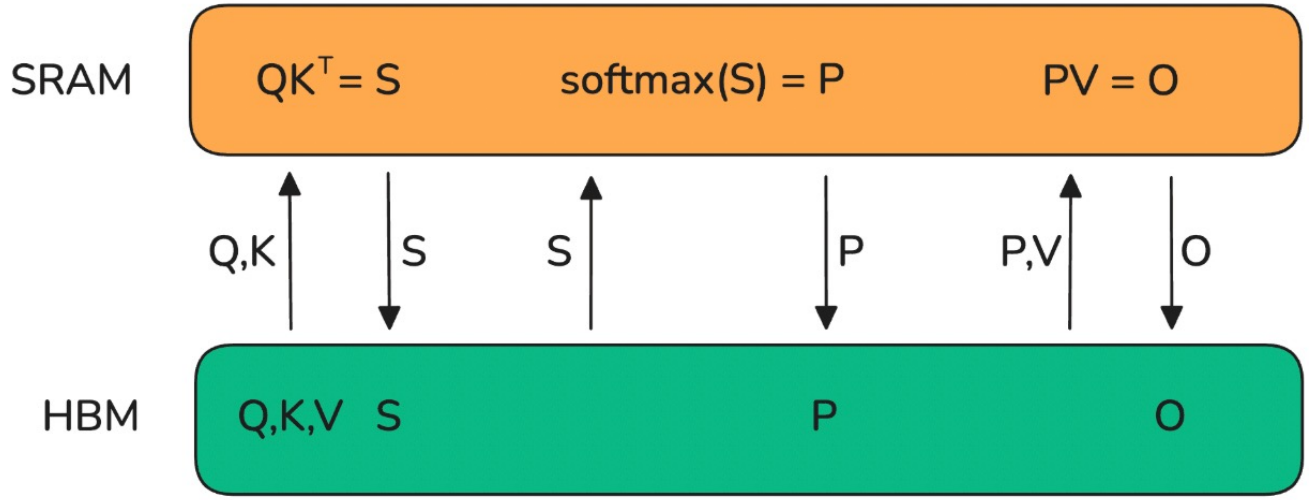
Overview

- FlashAttention이 왜 필요한가
- **핵심 아이디어: Tiling + Online Softmax + Kernel Fusion**
- 알고리즘 (Pseudocode)
- 이점/제약
- v1/v2/v3

4. Flash Attention

B) 핵심 아이디어: Tiling + Online Softmax + Kernel Fusion

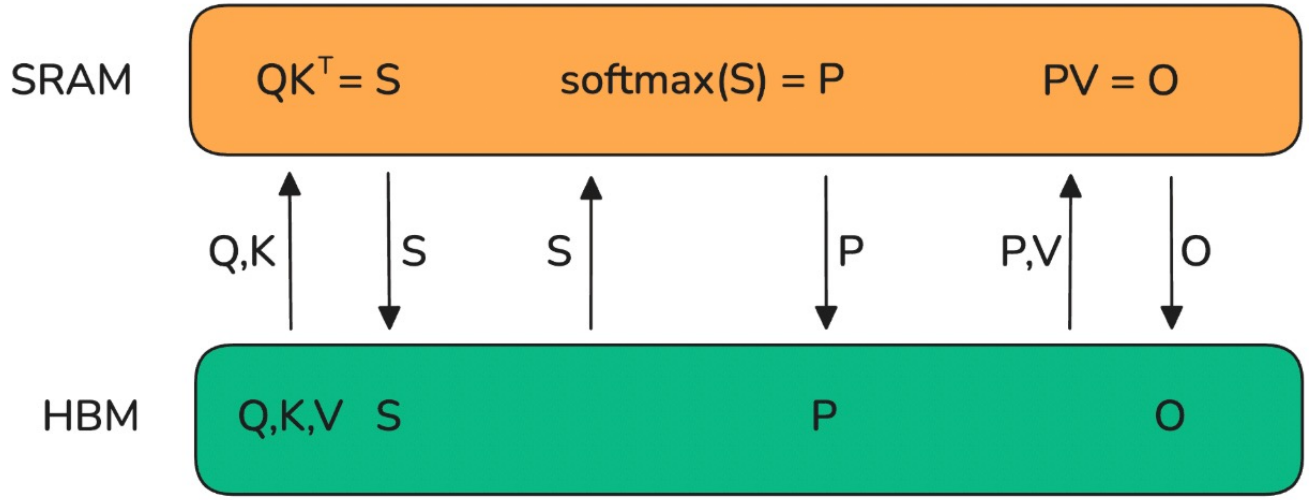
- 1. 타일링(Tiling)
- 2. S를 “절대” 물질화(materialize)하지 않기
- 3. 온라인 소프트맥스 (Online Softmax)
- 4. 연산 융합 (Kernel Fusion)



4. Flash Attention

B) 핵심 아이디어: Tiling + Online Softmax + Kernel Fusion

- **1. 타일링(Tiling)**
- 2. S를 “절대” 물질화(materialize)하지 않기
- 3. 온라인 소프트맥스 (Online Softmax)
- 4. 연산 융합 (Kernel Fusion)



4. Flash Attention

B) 핵심 아이디어: Tiling + Online Softmax + Kernel Fusion

- **1. 타일링(Tiling)**
  - “타일링(Tiling)” = GPU의 “메모리 크기”와 “속도” 차이 때문에 나온 **데이터 쪼개기 전략**
  - FlashAttention뿐 아니라 CNN, matmul, transformer 등 모든 GPU 연산의 핵심 아이디어
  - Flash Attention에서의 적용:
    - Q,K,V를 작은 **블록(타일) 단위**로 잘라 **온칩(SRAM)으로 가져와** 계산 → 바로 buffer에 누적
    - 이렇게 하면 온칩에서 대부분의 계산을 끝내고, **HBM 왕복을 최소화**

4. Flash Attention

B) 핵심 아이디어: Tiling + Online Softmax + Kernel Fusion

- **1. 타일링(Tiling)**
  - (Before) **기본 Attention**대로 한다면?
    - 너무 커서 SRAM에 절대 못 들어감 → HBM 왕복이 많아짐 → 느려짐

$$O = \text{softmax}(QK^T)V$$

- Q, K, V 는 각각 $[N \times d]$ 크기의 행렬
- 이걸 그대로 계산하면 $S = QK^T$ 는 **[N × N]** 크기!

(예: N=4096이면, 4096×4096 = 16M개 값!)

4. Flash Attention

B) 핵심 아이디어: Tiling + Online Softmax + Kernel Fusion

- **1. 타일링(Tiling)**
  - (After) **Flash Attention**을 사용한다면?

> - Q, K, V를 작은 블록(타일) 로 나누어서
>
> 온칩 SRAM에 불러오고, 그 안에서 QK^T 와 softmax 를 한 번에 계산해요.
> - 계산한 결과(일부 O)만 HBM에 저장하고, 다시 다음 타일로 넘어감.

4. Flash Attention

B) 핵심 아이디어: Tiling + Online Softmax + Kernel Fusion

- **1. 타일링(Tiling)**
  - Procedure

단계	설명
①	Q, K, V를 **HBM(큰 메모리)**에서 **작은 조각(타일)**로 나눔
②	**각 타일**을 빠른 **SRAM(Shared Memory)**에 잠깐 올림
③	SRAM 위에서 **연산 수행** (QK^T → softmax → PV)
④	계산 결과 **일부(O 블록)만** 다시 HBM에 저장
⑤	다음 타일로 넘어감

4. Flash Attention

B) 핵심 아이디어: Tiling + Online Softmax + Kernel Fusion

- **1. 타일링(Tiling)**
  - 한 번에 다 계산하면 4096×4096 matrix = 너무 큼!
  - FlashAttention에서는:
    - Q의 128행씩 (작은 타일) & K,V도 128행씩 잘라!

  → **"작은 128×128 블록"**을 SRAM에서 계산

  → 이 블록을 순차적으로 누적해서 전체 결과 O 완성!

- $Q : 4096 \times 64$
- $K : 4096 \times 64$
- $V : 4096 \times 64$

```
큰 Q,K,V
 └─> Q_tile[0:128], K_tile[0:128], V_tile[0:128]  -> SRAM에 로드
 └─> Q_tile[128:256], K_tile[128:256], V_tile[128:256]  -> 다음 블록 계산
 ...
```

4. Flash Attention

B) 핵심 아이디어: Tiling + Online Softmax + Kernel Fusion

- **1. 타일링(Tiling)**
  - Question) Softmax를 취하려면 **(부분인) 128개**가 아니라 **(전체인) 4096개**가 다 필요하지 않나?
  - Answer) **Online Softmax**

4. Flash Attention

B) 핵심 아이디어: Tiling + Online Softmax + Kernel Fusion

- **2. S를 “절대” 물질화(materialize)하지 않기**
  - (앞선 내용과 사실상 동일)
  - 두 줄 요약
    - (1) 큰 S 행렬(N×N)을 아예 만들지도, HBM에 쓰지도 (write) 않는다는 뜻
    - (2) 대신 작은 블록씩 계산하면서 softmax에 필요한 최소 정보(정규화 통계)만 들고 다니다가, 최종 출력 O만 HBM에 저장

4. Flash Attention

B) 핵심 아이디어: Tiling + Online Softmax + Kernel Fusion

- **2. S를 “절대” 물질화(materialize)하지 않기**
  - Q1) 왜 S를 만들지 (write) 않는가?
  - A1)
    - S는 크기 N×N이라 엄청 큼 (e.g., N=4096 → 1,677만 element).
    - 기존 방식) 큰 S를 HBM에 써놓은 뒤, softmax → P=softmax(S) → O=PV
      - 이 과정에서 HBM 왕복(쓰기/읽기)이 병목. 메모리도 터짐
    - 해결책) S 전체를 만들지 말고, 타일(블록) 단위로 S의 일부분만 잠깐 계산해서 곧바로 softmax에 반영하고 버린다!

4. Flash Attention

B) 핵심 아이디어: Tiling + Online Softmax + Kernel Fusion

- **2. S를 “절대” 물질화(materialize)하지 않기**
  - Q2) “정규화 통계”만 유지한다는 게 뭔데?
  - A2)
    - 즉, softmax 전체를 위해 S의 모든 값을 보관할 필요가 없고, 각 행의 (m, l)만 있으면 충분
    - FlashAttention은 블록씩 S의 조각을 계산할 때마다, 각 행의 (m, l) 을 온라인으로 갱신

softmax 한 행(row)에 대해

$$p_i = \frac{e^{s_i}}{\sum_j e^{s_j}}$$

여기서 필요한 건 사실 **두 가지**:

- 행의 **최댓값** $m = \max_j s_j$ (수치 안정화용)
- 행의 **지수합** $l = \sum_j e^{s_j - m}$ (분모)

4. Flash Attention

B) 핵심 아이디어: Tiling + Online Softmax + Kernel Fusion

- **3. 온라인 소프트맥스 (Online Softmax)**
  - 앞서 언급했듯, Softmax는 각 행에 대해 $p_i = \frac{e^{s_i}}{\sum_j e^{s_j}}$ 이므로,

    행마다 **"현재까지의 최대값/지수합"** 만 유지하면 블록을 순회하며 정확히 계산 가능!
  - 한 행 r에 대해, 현재까지 본 블록들에서
    - 누적 최댓값 m
    - 누적 지수합 l
    - 누적 출력 합 o (나중에 O = o/l 로 정규화)
  - 을 들고 있다고 하자.

4. Flash Attention

B) 핵심 아이디어: Tiling + Online Softmax + Kernel Fusion

- **3. 온라인 소프트맥스 (Online Softmax)**
  - 다음 블록에서 S_block = Q_block · K_block^T 를 계산하면:

---

1. 새 블록의 행별 최대 m_{blk} 를 구함
2. 새 최대와 기존 최대를 합쳐

   $m' = \max(m, m_{\text{blk}})$
3. 스케일 조정(정규화 기준이 m′로 바뀌었으니):

   $\text{scale} = e^{m - m'}$ (행별 스칼라)
4. 새 블록의 지수합:

   $l_{\text{blk}} = \sum_{j \in \text{block}} e^{s_j - m'}$
5. 누적 지수합 갱신:

   $l' = l \cdot \text{scale} + l_{\text{blk}}$
6. 출력 누적(여기서 V도 같이 사용):

   $o' = o \cdot \text{scale} + \sum_{j \in \text{block}} e^{s_j - m'} V_j$

4. Flash Attention

B) 핵심 아이디어: Tiling + Online Softmax + Kernel Fusion

- **3. 온라인 소프트맥스 (Online Softmax)**
  - 다음 블록에서 S_block = Q_block · K_block^T 를 계산하면:

그다음 $m \leftarrow m', l \leftarrow l', o \leftarrow o'$ 로 업데이트하고,

다음 블록으로 넘어갑니다. **S_block은 즉시 버림**(물질화 X).

모든 블록을 다 돌았으면,

최종 **출력**은 $O = \frac{o}{l}$

(행별로 나눔)만 HBM에 저장하면 끝!

요약:

- S 전체를 만들 필요 없이, **각 블록의 S만** 잠깐 계산
- **(m,l,o)** 만 업데이트
- **S는 버림**

4. Flash Attention

Overview

- FlashAttention이 왜 필요한가
- 핵심 아이디어: Tiling + Online Softmax + Kernel Fusion
- **알고리즘 (Pseudocode)**
- 이점/제약
- v1/v2/v3

4. Flash Attention

C) 알고리즘 (Pseudocode)

- (1) 정확성: Online softmax는 일반 softmax와 수치적으로 동일
- (2) 메모리: 거대한 S나 P를 만들지 않고, 행별 (m,l,o) 상태만 유지하므로 메모리 절감

```
# 입력: Q[N, d], K[N, d], V[N, d]
# 출력: O[N, d]  where  O = softmax(QK^T) V
```

```
for q_tile in tiles_of_rows(Q):                    # 바깥 루프: Q의 행 타일
    m = [-inf]*q_tile_rows                         # running max per row
    l = [0.0]*q_tile_rows                          # running sum(exp) per row
    o = zeros(q_tile_rows, d)                      # running output per row

    for k_tile, v_tile in zip(tiles_of_rows(K), tiles_of_rows(V)):  # 안쪽 루프
        # 1) 온칩(SRAM)으로 타일 로드
        Qb = load_to_SRAM(q_tile)                  # [Bq, d]
        Kb = load_to_SRAM(k_tile)                  # [Bk, d]
        Vb = load_to_SRAM(v_tile)                  # [Bk, d]

        # 2) 블록 점수 S_block = Qb @ Kb^T
        S_block = Qb @ Kb.T                        # [Bq, Bk]

        # 3) 온라인 소프트맥스 갱신 (수치안정: 행별 max 사용)
        m_new = max(m, rowmax(S_block))            # 행별 새로운 최대
        # scale = exp(m - m_new)       (각 행별 스칼라)
        # p_block = exp(S_block - m_new[:,None])
        # l_new = l*scale + rowsum(p_block)
        # o_new = o*scale[:,None] + p_block @ Vb

        scale  = exp(m - m_new)                    # [Bq]
        p_blk  = exp(S_block - m_new[:,None])      # [Bq, Bk]
        l_new  = l*scale + rowsum(p_blk)           # [Bq]
        o_new  = o*scale[:,None] + p_blk @ Vb      # [Bq, d]

        m, l, o = m_new, l_new, o_new              # 상태 갱신

    # 4) 타일의 최종 출력 정규화: o / l[:,None]
    O[q_tile_rows, :] = o / l[:,None]              # HBM에 최종 결과만 저장
```

4. Flash Attention

Overview

- FlashAttention이 왜 필요한가
- 핵심 아이디어: Tiling + Online Softmax + Kernel Fusion
- 알고리즘 (Pseudocode)
- **이점/제약**
- v1/v2/v3

4. Flash Attention

D) 이점/제약

이점

- **1. HBM 왕복 제거**: S, P를 HBM에 쓰고 **다시 읽는 과정을 없앰**
- **2. 온칩 재사용 극대화**: Q,K,V 타일을 **Register/Shared memory**에 올려 곧바로 다음 연산에 활용
- **3. O(N^2) 알고리즘 유지**: 근사(linear attention)가 아니라 **정확한 attention**
- **4. 메모리 피크가 큰 폭으로 감소**: 길이 **N이 커질수록** 차이가 큼 → **더 긴 context 가능**

4. Flash Attention

D) 이점/제약

제약 (신경써야할 점)

- **1. 패턴 제약**
  - (O) Causal/Masking/Dropout 등 자주 쓰는 패턴은 지원되지만,
  - (X) “아주 복잡한 custom 마스크”은 기본 FlashAttention에서 제약이 있을 수 있음(→ FlexAttention 참고)
- **2. 타일 크기 튜닝**
  - 하드웨어/정밀도(BF16/FP16/FP8)에 따라 최적 타일/스케줄이 달라짐
  - But 라이브러리에서 자동 튜닝 지원해줌!
- **3. 숫자 안정성**: Online softmax는 행별 max를 적절히 갱신해야 under/overfloew 회피

4. Flash Attention

Overview

- FlashAttention이 왜 필요한가
- 핵심 아이디어: Tiling + Online Softmax + Kernel Fusion
- 알고리즘 (Pseudocode)
- 이점/제약
- **v1/v2/v3**

4. Flash Attention

E) Flash Attention v1,v2,v3

- **FlashAttention-1**: 핵심 아이디어(Tiling + Online softmax + Kernel fusion)
- **FlashAttention-2**
  - 비 matmul 연산 최소화
  - warp/블록 간 작업 분할을 더 정교화
  - 스케줄링 최적화 → 추가 속도 상승
- **FlashAttention-3**:
  - Hopper(H100) 의 Tensor Core/FP8 활용을 극대화
  - 세대별 하드웨어 특성 맞춤 최적화 → 더 높은 처리량

v2/v3의 향상은 “알고리즘 변화”보다
저수준 커널·스케줄링·정밀도 활용의
공학적 미세 최적화에 가까움

5. Mixed Precision Training

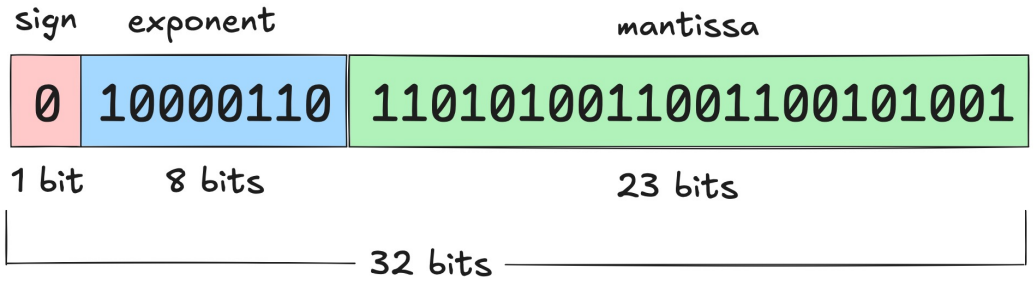
5. Mixed Precision Training

A) Floating-point의 형식

- 비트 수(32, 16, 8비트 등)가 달라질 때 **1) 표현 범위, 2) 정밀도, 3)속도/메모리**가 어떻게 바뀌는지?

구분 (FP32 기준)	Bit 수	역할
Sign	1	0이면 +, 1이면 –
Exponent	8	수의 크기(**스케일**)를 조정
Mantissa (= Fraction)	23	유효 숫자(**정밀도**)를 표현

- E.g., 예: (-1)^sign × (1.mantissa$_2$) × 2^(exponent–bias)
- 부호 × 유효숫자 × 10^n (X)
- **부호 × 맨티사 × 2^e** (O)



5. Mixed Precision Training

A) Floating-point의 형식

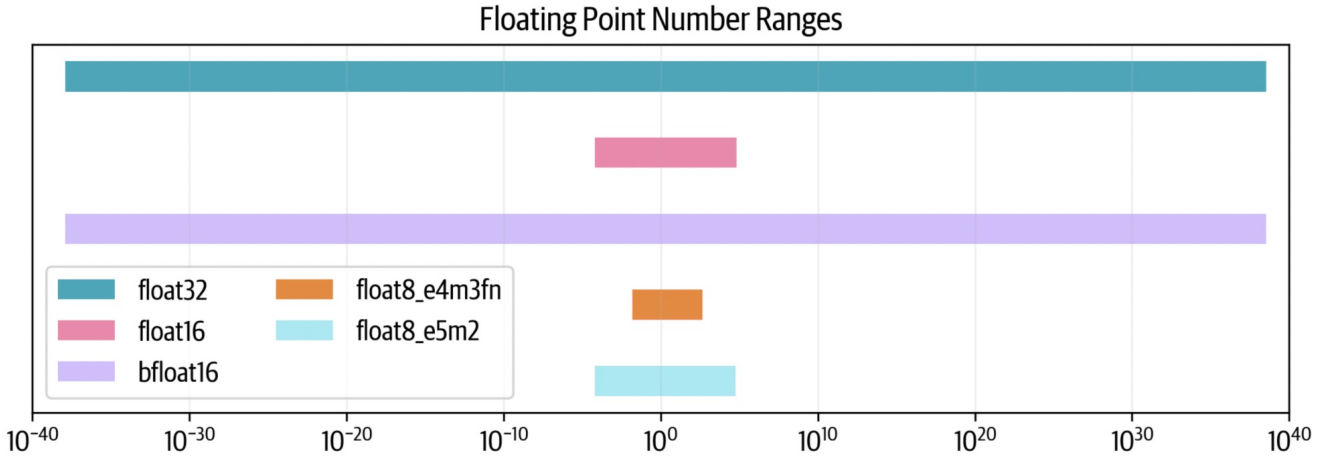
- 비트 수를 줄이면?
  - **Mantissa** 비트를 줄이면 → **정밀도**(소수점 아래 정확도) ↓
  - **Exponent** 비트를 줄이면 → **표현 범위**(아주 큰/작은 값) ↓
  - 전체 비트 수가 작아지면 → **메모리 절약 ↑ / 속도 ↑**

형식	전체비트	부호	지수	유효숫자	비고
float32	32	1	8	23	기준
float16	16	1	5	10	반정밀
bfloat16	16	1	8	7	지수 범위는 FP32와 같지만 정밀도 낮음
float8 e4m3	8	1	4	3	범위 작음 / 정밀도 조금 높음
float8 e5m2	8	1	5	2	범위 넓음 / 정밀도 더 낮음

5. Mixed Precision Training

B) 표현값의 범위

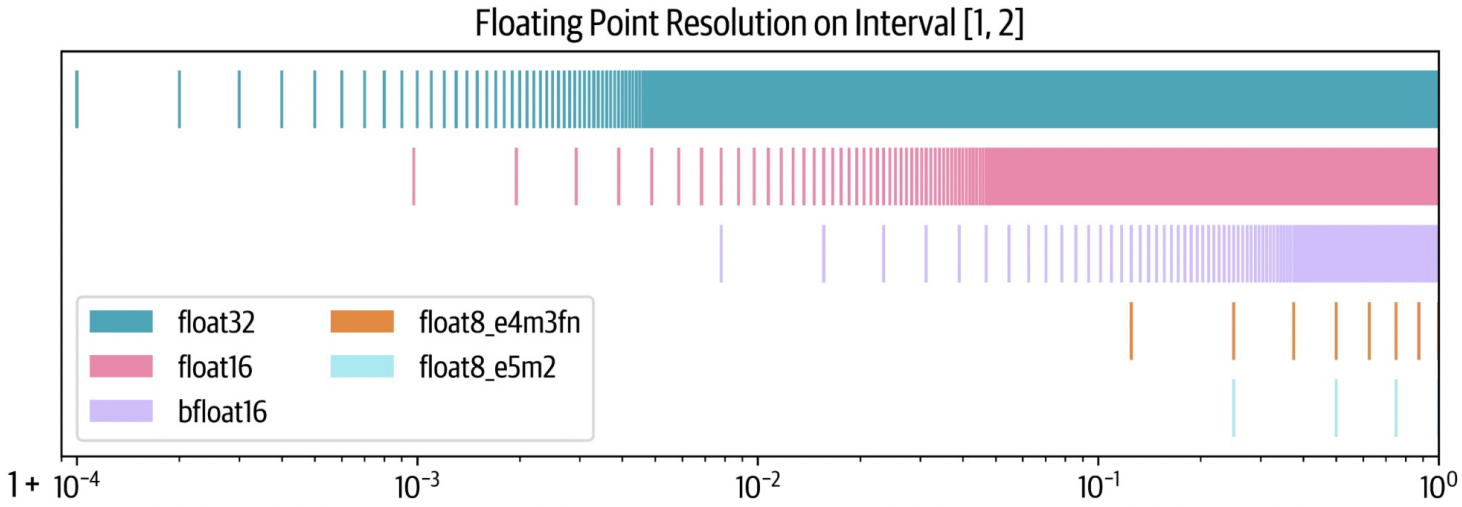
- **float32** : 약 10^{-38} ~ 10^{38} (80 자릿수 범위)
- **float16** : 지수 5개 → 범위 약 10^{-5} ~ 10^{5} 정도로 좁음
- **bfloat16** : 지수 8개 → 범위는 float32와 거의 같음 (정밀도만 낮음)
- **float8 (e5m2/e4m3)** : 범위가 매우 좁음 → 큰 값/작은 값을 다루면 빠르게 overflow·underflow 발생



5. Mixed Precision Training

C) 정밀도 비교

- **float32**는 1과 2 사이에서도 매우 촘촘하게 수 표현 (10^{-7} 정도 간격)
- **float16**은 간격이 10^{-3} 정도
- **bfloat16**은 간격이 더 크고(10^{-2} 수준)
- **FP8**은 단 몇 개의 값만 표현 가능



즉, 비트 수가 적을수록 "근처 값 끼리 구별 능력(정밀도)"이 떨어짐!

5. Mixed Precision Training

D) Mixed Precision Training이란?

- **연산의 종류**에 따라 **정밀도를 달리 써서** 속도·메모리를 절약하는 훈련 방법
  - FP32: 가중치 업데이트, 손실 누적 등 민감한 부분
  - FP16 / bfloat16: 행렬곱·컨볼루션 등 대부분의 대규모 연산
  - FP8: 최신 GPU(Hopper 등)에서는 더 과감히 적용 (통신·저장에 유리)
- 이렇게 하면 메모리 사용량을 절반 이하로 줄이고

  & Tensor Core를 풀로 활용해서 속도는 2~4배 빨라짐

5. Mixed Precision Training

E) Mixed Precision Training 상세

- **FP16·BF16 혼합정밀 훈련** & **FP8(더 낮은 정밀도)** 훈련의 핵심 아이디어
- 실제 대형 모델(예: GPT, Llama, DeepSeek) 훈련에서 GPU 효율을 극대화하기 위한 핵심 기술
- **"왜 문제가 생기고, 그걸 어떻게 해결하는지"**

5. Mixed Precision Training

E) Mixed Precision Training 상세

- (E-1) FP16 (half precision)만 사용하면 안되는 이유?
  - FP32(32비트)보다 **표현 가능한 수의 범위가 훨씬 좁고** 소수점 이하 **정밀도도 낮음**

구분	문제 설명
(a) **Underflow**	너무 작은 값(예: 1e−9)이 0으로 반올림되어 사라짐
(b) **Overflow**	너무 큰 값(예: 1e+6)이 무한대(Inf) 로 날아감
(c) **Rounding error**	계산을 반복하다 보면 작은 반올림 오차가 쌓임

- 훈련 중에는 **gradient가 아주 작음** → 이 값들이 FP16에서는 **0으로 소실**되어 학습 멈춤 (= loss 발산)

5. Mixed Precision Training

E) Mixed Precision Training 상세

- (E-2) FP16 only의 세 가지 해결책
  - (1) FP32 copy of weights (마스터 웨이트 유지)
  - (2) Loss scaling (손실 스케일링)
  - (3) FP32 accumulation (누적 시에는 32비트)

5. Mixed Precision Training

E) Mixed Precision Training 상세

- (E-2) FP16 only의 세 가지 해결책
  - **(1) FP32 copy of weights (마스터 웨이트 유지)**

```
FP32 master weights  ← optimizer update
     ↓ cast
FP16 model weights   ← forward/backward 계산용
```

    - 한 줄 요약: "**모델의 가중치**는 **FP16으로 계산**하지만, **진짜 원본**은 **FP32로 보관**한다"
    - 이유?
      - **FP16**: 작은 weight update가 0으로 반올림되어, **더 이상 갱신되지 않음**
      - 해결책:
        - **Optimizer(Adam, SGD)**는 **FP32 master weight를 업데이트**
        - **계산**할 때만 **FP16으로 캐스팅**해서 GPU에서 빠르게 연산

5. Mixed Precision Training

torch.cuda.amp.GradScaler()

E) Mixed Precision Training 상세

```
# 개념 예시
scaled_loss = loss * scale
scaled_loss.backward()
for param in model.parameters():
    param.grad /= scale  # unscale
```

- (E-2) FP16 only의 세 가지 해결책
  - **(2) Loss scaling (손실 스케일링)**
    - 한 줄 요약: “기울기가 **너무 작아서 0으로** 사라지는 걸 막기 위해, **loss를 미리 크게 키워준다**”
    - 방법
      - a) Forward에서 **loss를 scale factor(예: 1024) 로 곱함**

        → Gradient도 같이 커짐 (그래서 FP16에서도 0 안 됨!)
      - b) Backward 후에 **다시 gradient를 1/scale 로 나눠서** 원래 값 복원

        → 최종 결과는 동일, 하지만 계산 과정에서는 underflow 없음.

5. Mixed Precision Training

E) Mixed Precision Training 상세

- (E-2) FP16 only의 세 가지 해결책
  - **(3) FP32 accumulation (누적 시에는 32비트)**
    - 한 줄 요약: “16비트 계산 중간에 합/평균낼 때는 **FP32로 누적**하고, **마지막에만 FP16으로 변환**”
    - 예시
      - Batchnorm, layernorm, reduction(sum, mean) 같은 연산은 많은 값을 더해야!
      - FP16로 합치면 중간합이 금방 overflow·underflow
      - 그래서 **누적용 버퍼만 FP32**로 두고, **마지막 결과만 FP16으로** 캐스팅
    - 효과: 안정성 & 속도를 모두 확보

5. Mixed Precision Training

E) Mixed Precision Training 상세

- (E-2) FP16 only의 세 가지 해결책

트릭	원리	문제 해결
(1) **FP32 master weights**	FP16으로 계산하되, FP32 가중치로 업데이트	작은 업데이트가 사라지는 문제
(2) **Loss scaling**	Loss를 미리 크게 곱해 gradient 소실 방지	Underflow 방지
(3) **FP32 accumulation**	덧셈/평균 중간 결과는 32비트로 저장	Overflow/Underflow 방지

5. Mixed Precision Training

E) Mixed Precision Training 상세

- (E-3) **BF16**이 더 안정적인 이유?
  - bfloat16 (Brain Float 16) 은 **FP16과 비트 수는 같지만, bit 배분이 다름**
    - FP16: 0.00006 ~ 65504까지만 표현 가능
    - BF16: FP32와 같은 범위 (~10^{-38} ~ 10 ^{38})
  - 그래서 BF16은 **underflow/overflow가 거의 없음**

    → Loss scaling이 필요 없음

    → **대부분의 대형 모델**은 FP16 대신 BF16으로 학습

    (특히 TPU, H100 GPU에서는 BF16이 기본)

형식	exp	mantissa	특징
FP16	5	10	정밀도 높지만 표현 범위 좁음
BF16	8	7	정밀도 낮지만 범위가 넓음

5. Mixed Precision Training

E) Mixed Precision Training 상세

- (E-4) **FP8 훈련**으로 나아가면?
  - 계산 속도는 **FP16보다 2배 빠름** (H100에서 FP8 GEMM은 BF16 대비 2× FLOPS)
  - 하지만 정밀도와 범위 둘 다 낮아서 **불안정** → 학습이 쉽게 폭주하거나 **loss가 NaN으로** 터짐!
  - FP8 훈련의 기본 아이디어
    - **[FP8] 주요 행렬 연산 (QKV, GEMM 등)**
    - **[FP16/32] 중요한 값(마스터 웨이트, 옵티마 상태)**
    - 입력/출력은 정규화(normalization) 해서 **FP8 범위** 안에 맞춰줌

      → 예: 타일 단위(1×128 등)로 절댓값 최대치(amax) 기준 스케일링
  - FP8은 작은 범위를 가지므로, **입력 데이터를 압축(quantization) 시켜야**

5. Mixed Precision Training

E) Mixed Precision Training 상세

- (E-4) **FP8 훈련**으로 나아가면?
  - DeepSeek-V3의 FP8 전략

효과:

- 메모리 25~50% 절약
- 속도 2배 이상 향상
- 품질은 BF16 수준에 근접

항목	정밀도 조합	설명
Weight (GEMM)	FP8	모든 행렬곱은 FP8에서 수행
Master Weight	FP32	업데이트 안정성 위해 FP32 유지
Gradient Accum	FP32	덧셈은 FP32로
Optimizer State (m,v)	BF16	역전파 안정성 향상
Scaling	타일 단위(1×128, 128×128)로 스케일 정규화	outlier 영향을 줄임

5. Mixed Precision Training

E) Mixed Precision Training 상세

- (E-4) **FP8 훈련**으로 나아가면?
  - Memory 비교

방식	주요 정밀도 조합	총 메모리	절감율
BF16 + FP32 baseline	BF16 + FP32 누적	20 bytes	기준
FP8-LM	FP8 + FP16	9 bytes	−55%
DeepSeek-V3	FP8 + FP32/BF16	15 bytes	−25%
Nanotron FP8	FP8 + BF16	10 bytes	−50%

5. Mixed Precision Tr

항목	정밀도	바이트
Weight (BF16)	2	현재 가중치
Master weight (FP32)	4	안정적 업데이트용
Gradient (BF16)	2	역전파 결과
Gradient accum (FP32)	4	여러 스텝 평균
Optimizer state m (FP32)	4	1차 모멘트
Optimizer state v (FP32)	4	2차 모멘트

E) Mixed Precision Training 상세

- (E-4) **FP8 훈련**으로 나아가면?
  - Memory 비교

방식	주요 정밀도 조합	총 메모리	절감율
BF16 + FP32 baseline	BF16 + FP32 누적	**20 bytes**	기준
FP8-LM	FP8 + FP16	9 bytes	−55%
DeepSeek-V3	FP8 + FP32/BF16	15 bytes	−25%
Nanotron FP8	FP8 + BF16	10 bytes	−50%

5. Mixed Precision Tr

항목	정밀도	바이트
Weight (FP8)	1	현재 가중치
Master weight (FP16)	2	안정적 업데이트용
Gradient (FP8)	1	역전파 결과
Gradient accum (FP16)	2	여러 스텝 평균
Optimizer state m (FP8)	1	1차 모멘트
Optimizer state v (FP16)	2	2차 모멘트

E) Mixed Precision Training 상세

- (E-4) **FP8 훈련**으로 나아가면?
  - Memory 비교

방식	주요 정밀도 조합	총 메모리	절감율
BF16 + FP32 baseline	BF16 + FP32 누적	20 bytes	기준
FP8-LM	FP8 + FP16	**9 bytes**	−55%
DeepSeek-V3	FP8 + FP32/BF16	15 bytes	−25%
Nanotron FP8	FP8 + BF16	10 bytes	−50%

5. Mixed Precision Tr

항목	정밀도	바이트
Weight (FP8)	1	현재 가중치
Master weight (FP32)	4	안정적 업데이트용
Gradient (BF16)	2	역전파 결과
Gradient accum (FP32)	4	여러 스텝 평균
Optimizer state m (BF16)	2	1차 모멘트
Optimizer state v (BF16)	2	2차 모멘트

E) Mixed Precision Training 상세

- (E-4) **FP8 훈련**으로 나아가면?
  - Memory 비교

방식	주요 정밀도 조합	총 메모리	절감율
BF16 + FP32 baseline	BF16 + FP32 누적	20 bytes	기준
FP8-LM	FP8 + FP16	9 bytes	−55%
DeepSeek-V3	FP8 + FP32/BF16	**15 bytes**	−25%
Nanotron FP8	FP8 + BF16	10 bytes	−50%

5. Mixed Precision Tr

E) Mixed Precision Training 상세

- (E-4) **FP8 훈련**으로 나아가면?
  - Memory 비교

항목	정밀도	바이트
Weight (FP8)	1	현재 가중치
Master weight (BF16)	2	안정적 업데이트용
Gradient (FP8)	1	역전파 결과
Gradient accum (FP8)	1	여러 스텝 평균
Optimizer state m (FP8)	1	1차 모멘트
Optimizer state v (FP8)	1	2차 모멘트
이론상으로는 7byte나, 논문에서는 보수적으로 10byte 표기		

방식	주요 정밀도 조합	총 메모리	절감율
BF16 + FP32 baseline	BF16 + FP32 누적	20 bytes	기준
FP8-LM	FP8 + FP16	9 bytes	−55%
DeepSeek-V3	FP8 + FP32/BF16	15 bytes	−25%
Nanotron FP8	FP8 + BF16	**10 bytes**	−50%

5. Mixed Precision Training

F) Summary

- **FP16**: **빠르지만 수** 범위가 좁아서 **loss scaling + FP32 backup**이 필요
- **BF16**: **범위가 넓어** 더 안정적 → **요즘 대형 모델 기본**
- **FP8**: **더 빠르고** 가볍지만 아직 실험적 (**DeepSeek-V3**, H100에서 실제로 실험 성공)

포맷	속도	안정성	품질	스케일링 필요	메모리 절약
FP32	느림	최고	기준	❌	-
FP16	빠름	불안정	약간 낮음	✅ Loss scaling 필요	약 2× 절약
BF16	빠름	안정	거의 동일	❌	약 2× 절약
FP8	매우 빠름	매우 불안정	연구 중	✅ 정규화 필수	4× 절약