FlashAttention-3

Fast and Accurate Attention with Asynchrony and Low-precision

2024.08.01

HPC Lab

홍성준, 박지연, 김유나

Contents Table

1 Summary

✓ **FlashAttention1, 2는 A100 GPU에서 메모리 Read/Write를 최소화**하여 Attention 속도를 높이는 접근 방식을 소개했다. 그러나 FlashAttention-2는 H100 GPU에서 35%의 활용률만 달성했다.

✓ 따라서 **Hopper GPU**에서 Attention 속도를 높이기 위해 세 가지 주요 기술을 개발했다.

1. Warps-Specialization을 통한 전체 **계산과 데이터 이동을 Overlap** 한다.
2. Block 단위의 **Matmul 및 Softmax 연산을 교차적**으로 수행한다.
3. **FP8 Low-Precision**에 대한 하드웨어 지원을 활용하는 Block Quantization 및 Incoherent Processing.

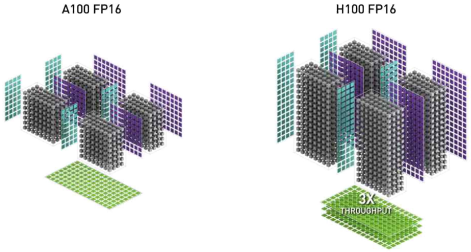
✓ H100 GPU에서 FP16으로 **740 TFLOPs/s (75% 활용률)**를 달성하고, FP8에서는 **1,200 TFLOPs/s**를 달성했다.

Features on Hopper GPU Hardware

- ✓ **WGMMA**: WarpGroup level에서 **Matrix Multiply 및 Accumulate** operation을 수행하는 명령어이다.

  Ampere 아키텍처의 mma.sync 명령어 보다 **높은 처리량을 제공**한다.

- ✓ **TMA**(Tensor Memory Accelerator): GMEM와 SMEM 간의 **데이터 전송을 가속화**하는 하드웨어 unit이다.
- ✓ WGMMA 명령어는 Hopper에서 FP8 Tensor Core를 대상으로 하여 FP16과 비교할 때 SM당 처리량을 2배로 제공한다.

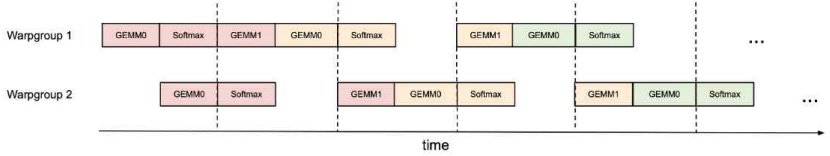


Warps-Specialization

- ✓ CTA(Cooperative Thread Array) 내의 **Warps를 특정 역할**(Producer / Consumer)**로 분할**하여 각각의 Warp가 특정 작업**(데이터 로드 / 계산)만 수행**하도록 하는 방식이다.
- ✓ **Producer Warps**: TMA(Tensor Memory Accelerator)를 이용하여 **데이터 로드를 비동기적으로** 처리할 수 있어 하나의 데이터 전송이 완료되기를 기다리지 않고 다른 작업을 진행할 수 있다.
- ✓ **Consumer Warps**: WGMMA는 전체 워프 그룹 내에서 GEMM을 수행할 수 있어, 병렬 처리를 극대화한다.
- ✓ 이를 통해 **데이터 로드와 계산을 비동기적으로 처리**하여 병렬성이 향상되고, 대기 시간을 최소화한다.

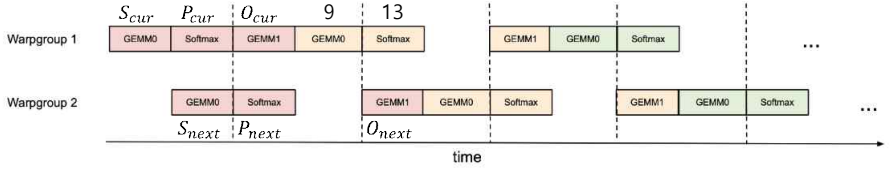
Ping-Pong Scheduling

- ✓ Matmul 연산은 **Tensor Core**에서 처리되고, Non-matmul 연산은 **Multi-function unit**에서 수행된다.
- ✓ Non-matmul 연산의 처리량은 Matmul 연산보다 훨씬 낮다.
- ✓ 따라서 **Tensor Core에서 Matmul 연산을 수행할 때 Multi-function unit에서 지수 계산이 스케줄링** 되는 것이 이상적이다.
- ✓ Warpgroup1의 Softmax는 Warpgroup2가 GEMM을 수행하는 동안 수행된다.
- ✓ 그런 다음, Warpgroup2가 Softmax를 수행하는 동안 Warpgroup1이 GEMM을 수행한다.



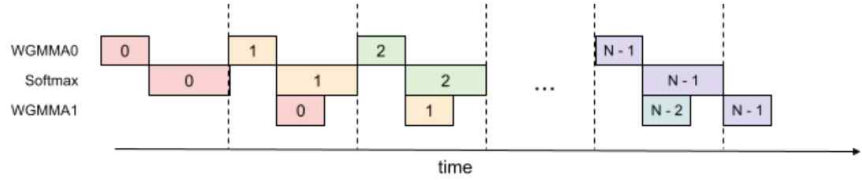
Ping-Pong Scheduling

4: Compute $\mathbf{S}_{\text{cur}} = \mathbf{Q}_i\mathbf{K}_0^T$ using WGMMA. Commit and wait.
5: Release the 0th stage of the buffer for $\mathbf{K}$.
6: Compute m_i, $\tilde{\mathbf{P}}_{\text{cur}}$ and ℓ_i based on $\mathbf{S}_{\text{cur}}$, and rescale $\mathbf{O}_i$.
7: **for** $1 \le j < T_c - 1$ **do**
8: Wait for $\mathbf{K}_j$ to be loaded in shared memory.
9: Compute $\mathbf{S}_{\text{next}} = \mathbf{Q}_i\mathbf{K}_j^T$ using WGMMA. Commit but do not wait.
10: Wait for $\mathbf{V}_{j-1}$ to be loaded in shared memory.
11: Compute $\mathbf{O}_i = \mathbf{O}_i + \tilde{\mathbf{P}}_{\text{cur}}\mathbf{V}_{j-1}$ using WGMMA. Commit but do not wait.
12: Wait for the WGMMA $\mathbf{Q}_i\mathbf{K}_j^T$.
13: Compute m_i, $\tilde{\mathbf{P}}_{\text{next}}$ and ℓ_i based on $\mathbf{S}_{\text{next}}$.
14: Wait for the WGMMA $\mathbf{P}_{\text{cur}}\mathbf{V}_{j-1}$ and then rescale $\mathbf{O}_i$
15: Release the $(j\,\%\,s)$th, resp. $(j-1\,\%\,s)$th stage of the buffer for $\mathbf{K}$, resp. $\mathbf{V}$.
16: Copy $\mathbf{S}_{\text{next}}$ to $\mathbf{S}_{\text{cur}}$.
17: **end for**



Intra-warpgroup overlapping GEMMs and softmax

- ✓ 한 **Warpgroup 내에서도** Softmax 명령어와 GEMM의 일부 명령어를 겹칠 수 있다.
- ✓ Loop 안에서 반복 j의 두 번째 WGMMA 연산(O = PV)은 반복 j+1의 Softmax 연산과 겹친다.
- ✓ 따라서 아래 그림과 같이 한 Warpgroup 내에서도 병렬화가 가능하다.



Intra-warpgroup overlapping GEMMs and softmax

✓ 이 파이프라인은 이론적인 성능 향상을 제공하지만, 실질적인 측면에서 고려해야 할 점이 있다.

1. **Compiler Reordering**: Compiler(NVCC)는 최적화를 위해 종종 **명령어를 재배치**하여 설계된 WGMMA 및 non-WGMMA 연산 파이프라인 순서를 방해할 수 있다.

2. **Register Pressure**: **GEMM의 Accumulate와 Softmax의 Input / Output을 보관**하기 위해 Threadblock 당 $Br \times Bc \times$ float* 크기의 추가 레지스터 사용을 초래한다. 이러한 레지스터 수요 증가는 **더 큰 블록 크기 사용(Another Common Optimization)과 충돌**할 수 있으며, 이는 레지스터를 많이 요구한다. 따라서 실제로는 프로파일링 결과에 따라 절충해야 한다.

* B_r, B_c: Block size

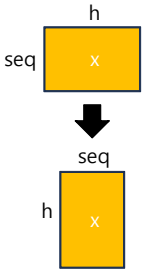
Low-Precision with FP8

✓ **Efficiency**: **FP8 precision**에서 Layout 일치 문제가 발생한다.

1. 입력 Tensor Q, K, V는 일반적으로 **Head dimension에서 연속적**이지만,

   FP8 WGMMA의 GEMM을 위해서 V가 **Sequence dimension에서 연속적**이어야 한다.

   -> SMEM에 로드한 후 V의 타일을 Kernel 내에서 **전치**한다.



2. FP8 WGMMA의 FP32 Accumulator의 메모리 Layout은 레지스터에 저장된 피연산자 A의 **Layout과 다르다**.

   -> Byte permute 명령어를 사용하여 이전의 WGMMA Accumulator를 다음 WGMMA에 적합한 형식으로 변환할 수 있다.

{d0 d1 d4 d5 d2 d3 d6 d7}

The order in sequence

Low-Precision with FP8

✓ **Accuracy**: FP8 형식은 FP16/BF32에 비해 더 높은 **수치적 오류**(Outlier)를 초래한다.

텐서별 스케일링(M)을 사용하여 각 텐서(Q, K, V)에 대해 하나의 스칼라 값을 유지한다.

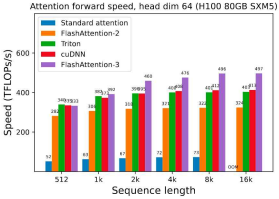
1. **Block Quantization**: 블록당 하나의 스칼라 값을 유지하여, Q, K, V 각각에 대해 텐서를 블록으로 나누고 **별도로 양자화**한다.
2. **Incoherent Processing**: Outlier를 고르게 하기 위해 Q, K에 **랜덤 직교 행렬 M**을 곱한 후 양자화한다.

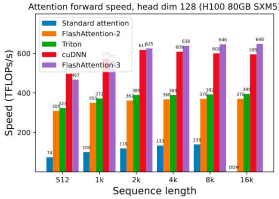
$$\mathbf{M}\mathbf{M}^\top = I \text{ and so } (\mathbf{Q}\mathbf{M})(\mathbf{K}\mathbf{M})^\top = \mathbf{Q}\mathbf{K}^\top$$

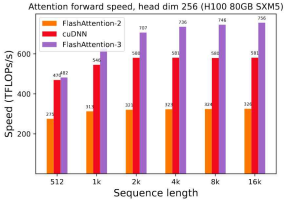
Empirical Validation

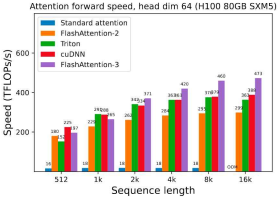
Without Causal Mask

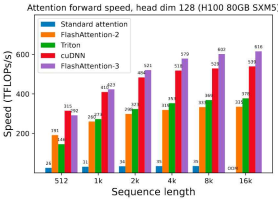
With Causal Mask











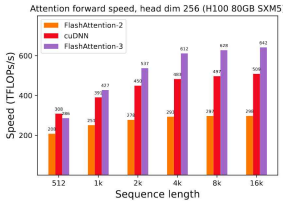
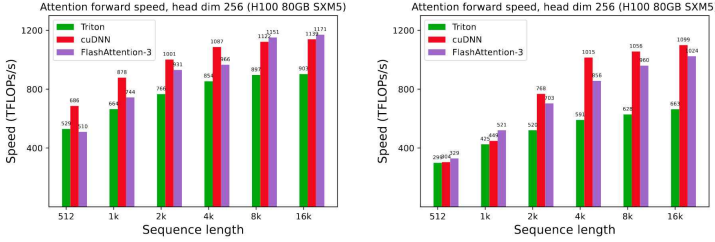


Figure 5: Attention forward speed (FP16/BF16) on H100 GPU

Empirical Validation



(a) Forward, without causal mask, head dim 256

(b) Forward, with causal mask, head dim 256

Figure 7: Attention forward speed (FP8) on H100 GPU

Table 3: Numerical error comparisons in FP16 and FP8 (e4m3).

Method	Baseline FP16	FlashAttention-2 FP16	FlashAttention-3 FP16
RMSE	3.2e-4	**1.9e-4**	**1.9e-4**

Method	Baseline FP8	FlashAttention-3 FP8	No block quant	No incoherent processing
RMSE	2.4e-2	**9.1e-3**	9.3e-3	2.4e-2


Thank You