석사학위논문

DNN 모델 추론 및 학습 시 분배 메커니즘을 통한 효율성 향상

2023년

한 성 대 학 교 대 학 원

I T 융 합 공 학 과

I T 융 합 전 공

김 인 모

석 사 학 위 논 문
지도교수 김명선

DNN 모델 추론과 학습 시 분배 메커니즘을 통한 효율성 향상

Improving the Efficiency of DNN Model Inference and Training through Distribution Mechanisms

2022년 12월 일

한 성 대 학 교 대 학 원

I T 융 합 공 학 과

I T 융 합 공 학 전 공

김 인 모

석 사 학 위 논 문
지도교수 김명선

DNN 모델 추론 및 학습 시 분배 메커니즘을 통한 효율성 향상

Improving the Efficiency of DNN Model Inference and Training through Distribution Mechanisms

위 논문을 공학 석사학위 논문으로 제출함

2022년 12월 일

한 성 대 학 교 대 학 원

I T 융 합 공 학 과

I T 융 합 공 학 전 공

김 인 모

김인모의 공학 석사학위 논문을 인준함

2022년 12월 일

심사위원장 오 희 석 (인)

심 사 위 원 이 웅 희 (인)

심 사 위 원 김 명 선 (인)

국 문 초 록

DNN 모델 추론 및 학습 시
분배 메커니즘을 통한 효율성 향상

한 성 대 학 교 대 학 원
I T 융 합 공 학 과
I T 융 합 공 학 전 공
김 인 모

딥러닝 기술의 발전으로 다양한 응용분야에서 DNN 모델들이 활용되고 있고 이러한 분야에 사용되는 DNN 모델들은 추론 시 높은 정확도와 빠른 추론시간을 요구하고 높은 정확도에 따른 높은 연산량을 가지는 DNN모델은 학습 시 많은 GPU 자원을 필요로 한다.

DNN을 실행하는 환경은 크게 두 가지로 나눌 수 있다. 하나는 단일 DNN을 실행하는 환경이고, 다른 하나는 다수의 DNN을 실행하는 환경이다. 본 논문에서는 분배 메커니즘을 적용하여 단일 DNN 실행환경과 다수의 DNN 실행환경에서 효율성을 향상시키는 방안에 대하여 제시한다. 단일 DNN 실행 시에는 DNN 모델 내부에 분배 메커니즘을 적용하여 모델 추론 과정을 대상으로 하여 추론 시간 감소를 통한 효율성 향상을 확인한다. 다수의 DNN 실행 시에는 DNN 모델 학습 과정을 대상으로 학습 시 할당되는 GPU 자원을 분배 메커니즘을 적용하여 효율적으로 나눠주어 전체 학습 응용이 공정하게 GPU 자원을 배분받을 수 있는 알고리즘에 대하여 제시한다.

단일 DNN 모델 추론과정에서는 객체추적 모델인 CSWinTT를 대상으로 하여 CSWinTT 모델 내부 트랜스포머 인코더의 MHA(Multi-Head Attention) 각 head 연산을 병렬실행으로 변경하여 추론시간을 감소시키는 방법을 제시한다. 병렬실행을 위하여 하나의 모듈로 이루어진 MHA 모듈을 head연산 단위로 분리하고 분리된 각각의 모듈을 멀티쓰레드 환경에서 실행한다. 이때 완전한 멀티쓰레드 동작을 위하여 Python 환경이 아닌 C++ 환경에서 모델을 실행한다. 병렬실행의 효과를 확인한 결과 기존방식 대비 인코더의 평균 실행시간은 56.8% 감소하였고, 평균 FPS는 63.3% 증가하였다.

다수의 DNN 모델 학습 과정에서는 GPU 자원의 총량이 모든 학습 응용들이 필요로 하는 자원의 총량보다 적어 GPU 자원을 차지하기 위한 경쟁이 DNN 학습 응용 간에 발생하여 상대적으로 긴 학습시간을 가지는 응용이 나타나는 문제를 해결하기위한 방안을 제시하였다. 현재 각 응용들의 GPU 자원 활용 상태를 기반으로 예측된 학습 완료 시간과 오프라인에서 각 응용들이 GPU를 독점해서 사용했을 때의 학습 완료 시간의 비율을 구하여 응용의 Slow-Down으로 정의한다. 학습이 진행되는 응용들의 Slow-Down 값이 비슷한 값으로 유지되도록 각 학습 응용들의 GPU당 미니배치 데이터 비율을 주기적으로 조절하여 GPU 자원의 공정한 배분이 이루어지도록 한다. 실험을 통해 확인한 결과 최대 Slow-Down은 53%이상 감소하였고, 전체적인 GPU 활용률은 25% 증가하였다.

【주요어】 멀티 헤드 어텐션, 멀티쓰레딩, 다중 GPU 학습, Slow-Down

목 차

제 1 장 서 론 ····· 1

제 2 장 연구 배경 및 문제점 ····· 4

제 1 절 단일 DNN 추론 시 연구배경 ····· 4
1) 타겟모델(CSWinTT) 구조 및 동작방식 ····· 4
2) 타겟모델 추론 시 실행시간 비율 분석 ····· 6
제 2 절 다수의 DNN 학습 시 문제 상황 및 자원배분 방법 ····· 7
1) 다수의 DNN 모델 학습 시 불공정한 자원배분 ····· 7
2) DNN 학습 응용들의 GPU 자원배분 방법 ····· 8

제 3 장 문제 해결 및 개선을 위한 기법 제안 ····· 9

제 1 절 병렬 MHA 연산 기반 트랜스포머 인코더 ····· 9
1) 모듈 분리 ····· 9
2) MHA 연산과정 ····· 11
3) 병렬 MHA 연산적용 ····· 12
제 2 절 적응형 SD 기반 미니배치 재분배 기법 ····· 15
1) 제안 기법 ····· 15
2) 학습 응용에서 각 GPU에 분배하는 미니배치 데이터의 비율 ····· 16
3) 전체 시스템의 동작방식 ····· 17
4) GPU에 할당할 미니배치 비율 설정 알고리즘 ····· 19

제 4 장 실험 및 분석 ····· 23

제 1 절 병렬 MHA 연산적용에 따른 추론시간 감소 실험 ····· 23
1) 실험 환경 ····· 23
2) 실험 결과 및 분석 ····· 24
가) 각 Case의 인코더 실행시간 비교 ····· 24
나) 각 Case의 평균 FPS 비교 ····· 25

다) 각 Case의 이미지 전처리와 인코더 실행시간 비율 비교 ····· 26
제 2 절 다수의 DNN 학습 시 GPU 자원배분 실험 ····· 27
1) 실험 환경 ····· 27
2) 실험 결과 및 분석 ····· 28
가) 시간에 따른 $SD_{\max}^{diff}$ 변화 ····· 29
나) 시간에 따른 평균 SD값의 변화 ····· 29
다) 시간에 따른 전체 GPU의 평균 활용률 변화 ····· 30

제 5 장 결 론 ····· 32

참 고 문 헌 ····· 33

ABSTRACT ····· 36

표 목 차

[표 2-1] 객체추적에 소요되는 각 기능 블록의 실행시간 비율 ····· 6
[표 2-2] [그림 2-2]에 나타낸 응용들의 완료시간 ····· 7
[표 3-1] 4개의 GPU에서 2개의 DNN 학습 응용이 실행되는 예시 ····· 16
[표 4-1] 단일 DNN 추론 실험환경 ····· 23
[표 4-2] 다수의 DNN 학습 실험환경 ····· 27

그 림 목 차

[그림 2-1] CSWinTT 모델의 전체 실행과정 ····· 4
[그림 2-2] 3개의 학습 응용이 2개의 GPU에서 실행되는 예시 ····· 7
[그림 3-1] 파이썬 환경, C++환경에서의 PyTorch 모델 실행 ····· 9
[그림 3-2] TorchScript를 통한 인코더 레이어 모듈 분리 ····· 10
[그림 3-3] 인코더 내 한 레이어 실행 과정 ····· 11
[그림 3-4] 파이썬 환경에서 각 쓰레드들이 커널을 전달하는 과정 ····· 13
[그림 3-5] C++ 환경에서 각 쓰레드들이 커널을 전달하는 과정 ····· 14
[그림 3-6] MHA병렬연산 적용 후 인코더 내 한 레이어 실행 과정 ····· 14
[그림 3-7] 적응형 SD기반 미니배치 재분배 시스템의 전체 동작 과정 ····· 18
[그림 3-8] DR Management 알고리즘 수도코드 ····· 20
[그림 3-9] DR 업데이트 알고리즘 수도코드 ····· 21
[그림 4-1] 인코더 평균 실행시간 ····· 25
[그림 4-2] 평균 FPS 비교 ····· 25
[그림 4-3] 전체 실행과정 중 이미지 전처리와 인코더 실행시간의 점유율 비교 ····· 26
[그림 4-4] 학습시간에 따른 $SD_{\max}^{diff}$ 변화 ····· 29
[그림 4-5] 학습시간에 따른 전체 학습 응용 SD 평균값의 변화 ····· 30
[그림 4-6] 학습시간에 따른 GPU 활용률 평균값의 변화 ····· 30

제 1 장 서론

최근 딥러닝의 기술적 발전에 따라 DNN 모델의 정확도가 높아지고 있다. 높은 정확도의 DNN 모델들을 활용하여 좋은 성과를 거두는 분야 또한 늘어나고 있으며 대표적으로 자율주행, 이미지 세그멘테이션(Image Segmentation), 객체추적(Visual Object Tracking) 등이 있다.

이 중 객체추적은 이전 프레임에서의 객체 위치 정보를 기반으로 현재 프레임에서의 객체 위치를 도출하는 기술을 일컫는다. 객체추적 추론 시 주어지는 유일한 정보는 첫 번째 프레임에서의 찾고자하는 객체의 위치 정보이다. 추론이 진행됨에 따라 이전 프레임에서의 객체 위치정보를 이용하여 현재 프레임에서의 객체 위치를 찾아낸다. 객체추적 분야에서는 VIT(Dosovitskiy, et al., 2020)의 발표이후 본래 자연어 처리 분야에서 많이 쓰이던 트랜스포머(Vaswani, et al., 2017)를 접목시켜 높은 성능을 보이는 객체추적 모델들이 발표되고 있다. 2022년 CVPR(Computer Vision and Pattern Recognition)학회에서 발표된 CSWinTT(Song, et al., 2022) 모델 또한 트랜스포머를 적용한 객체추적 모델이다. CSWinTT는 VOT2020 데이터셋(Kristan, et al., 2020)을 이용한 성능 측정에서 다른 9개의 객체추적 모델과 비교하였을 때 가장 높은 예상 평균 중첩도(EAO, Expected Average Overlap)값을 가져 최고성능을 내는 모델로 평가받는다. 또한 CSWinTT는 객체 추적의 성능을 높이기 위하여 모델 내부 트랜스포머 인코더(Encoder)에 자연어처리(Nadkarni, et al., 2011)에서 사용한 멀티 헤드 어텐션(Multi-Head Attention, 이하 MHA) (MultiheadAttention - PyTorch 1.13 documentation, online)이 아닌 Multi-Scale Window Partition과 Cyclic Shift(Liu, et al., 2021)를 적용한 독특한 방식의 MHA를 사용하였다. 이러한 방식은 Attention 연산(Vaswani, et al., 2017)을 하는 최소단위가 패치(픽셀의 집합)이기 때문에 픽셀간의 Attention 연산에 비하여 추적하려는 객체를 더 적게 손상시킨다는 장점이 있다. 또한 Cyclic Shift를 통해 객체를 찾아야하는 후보 이미지 영역에 대한 더 많은 위치정보를 가지게 된다.

객체추적 기술에서 추론속도는 매우 중요한 요소이다. 하나의 프레임에 대

한 추론속도는 이미지가 로드되고부터 bounding box 결과가 도출될 때 까지 걸리는 시간을 의미한다. 객체추적의 특성상 임의의 i번째 프레임에 대한 추론이 진행되기 위하여 $i-1$번째 프레임에서의 객체위치정보가 필요하다. $i-1$번째 프레임의 추론속도 감소는 i번째 프레임의 추론시작 시간을 앞당겨 i번째 프레임까지의 추론완료 시간을 감소시킨다. 이는 마지막 프레임까지 연쇄적으로 작용하여 전체 프레임에 대한 추론속도 감소로 이어진다.

본 논문에서는 객체추적을 수행하는 CSWinTT 모델을 대상으로 추론시간을 감소시켜 초당 처리하는 프레임의 수를 늘리는 방법을 제안한다. 먼저 추론 과정에서 각 기능별로 소요되는 시간을 분석하여 실행시간을 단축시킬 수 있는 방안을 제안한다.

DNN 모델의 정확도가 높아질수록 모델 자체의 연산 복잡도는 늘어가고 이러한 DNN 모델을 학습할 때 DNN 연산에 매우 효과적으로 사용될 수 있는 SIMT(Single Instruction Multi Thread)기능을 제공하는 GPU가 많이 사용되고 있다. 보통의 GPU 서버는 복수개의 GPU가 하나의 클러스터 형태로 제공되며 큰 모델을 학습할 때는 여러 개의 클러스터가 한꺼번에 동원되기도 한다.

클러스터 내부에서 다수의 DNN 모델 학습 응용 실행 시 사용가능한 GPU자원이 학습이 진행 중인 모든 응용들이 필요로 하는 GPU 자원의 총량보다 많은 경우 각각의 학습 응용은 필요한 만큼의 자원을 충분하게 할당받을 수 있다. 이 경우 응용 간에 GPU 자원 경쟁이 발생하지 않아 GPU 자원 배분에 대한 문제가 발생하지 않는다. 하지만 클러스터 내부의 사용가능한 GPU자원이 학습이 진행 중인 모든 응용들이 필요로 하는 GPU 자원의 총량보다 적은 경우에는 응용들 간 GPU 자원에 대한 경쟁이 발생한다.

어떤 응용은 GPU 자원을 충분히 사용하여 모델 학습이 마치 하나의 GPU를 독점하여 사용했을 때와 동일하게 종료될 수 있고, 어떤 응용은 GPU 자원에 대한 심한 경쟁으로 인하여 GPU 하나를 독점하여 학습을 종료할 경우에 비하여 훨씬 더 긴 시간동안 학습이 진행 될 수도 있다. 따라서 클러스터 내부의 복수개의 GPU를 다수의 DNN 모델 학습 응용에 할당하는 경우 공정한 자원분배가 이루어져야 한다.

DNN 모델 학습 응용들의 현재 학습 진행 정도를 x라 하고, GPU 하나를 독점해서 사용할 때의 학습 진행정도를 y라고 할 때, 어떤 응용들은 x/y의 비율 값이 매우 크고 어떤 응용들은 상대적으로 매우 작을 수 있다. 각 응용들이 x/y값을 비슷하게 유지할 수 있다면 모든 응용들의 GPU 자원 사용정도를 공정하게 유지할 수 있다.

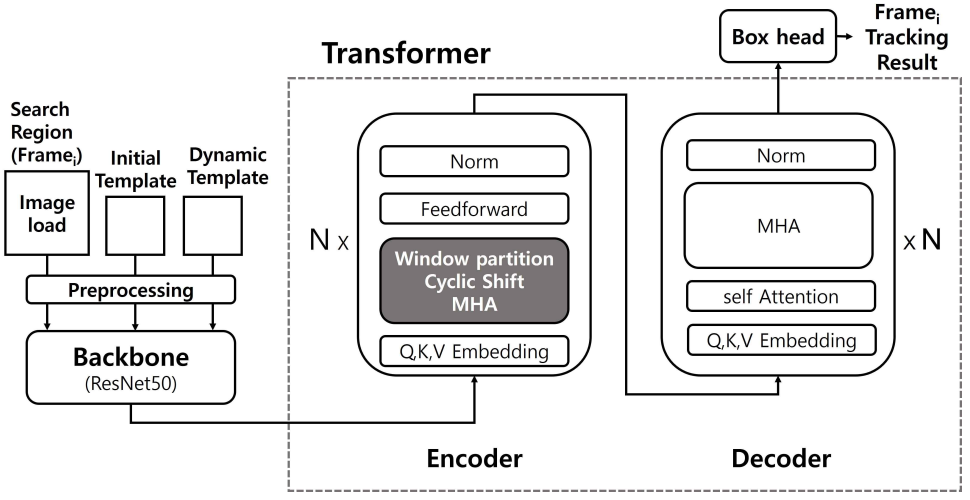
본 논문에서는 다수의 DNN 학습 응용들이 비슷한 x/y 값을 가지고 그 값이 유지되는 것을 목표로 하는 알고리즘을 제시한다. 학습이 진행되는 동안 주기적으로 x/y 값이 작은 응용들이 사용하는 GPU 자원을 x/y 값이 큰 응용들에게 분배함으로써 전체 응용의 공정한 GPU 자원 활용을 보장한다.

논문의 2장에서는 단일 DNN 추론 환경에 대한 연구배경과 다수의 DNN 학습시의 문제점을 설명하고, 3장에서는 각각의 환경에 대하여 효율성 향상을 위한 방법을 제시한다. 4장에서는 효율성 향상을 확인하기 위한 실험과정을 설명하고 결과를 분석한다. 마지막으로 5장에서 결론을 정리한다.

제 2 장 연구배경 및 문제점

제 1 절 단일 DNN 모델 추론 시 연구 배경

1) 타겟모델(CSWinTT) 구조 및 동작방식



[그림 2-1] CSWinTT 모델의 전체 실행과정

[그림 2-1]은 CSWinTT 모델 추론 시 임의의 i번째 프레임에서 객체추적의 결과인 bounding box 좌표값을 도출하는 전체 과정을 나타낸다. 프레임은 하나의 이미지를 의미하며, 하나의 데이터는 연속된 여러 프레임의 집합으로 이루어져 있다. 입력값은 총 3가지로, 크게 Template와 Search Region으로 구분된다. Template는 찾으려는 대상 객체이고 Search Region은 Template를 찾을 하나의 프레임을 의미한다. Template는 다시 Initial Template와 Dynamic Template로 구분된다. Initial Template는 데이터의 첫 번째 프레임에서 사용자가 찾을 객체의 위치정보이다. 객체의 위치정보는 좌

상단의 x, y 좌표값과 x, y 좌표값을 기준으로 한 너비, 높이 값 총 4개의 값으로 구성되어 있다. Dynamic Template는 최초에는 Initial Template와 동일하지만, 추론이 진행되면서 일정한 수의 프레임마다 특정 조건을 만족하면 현재 프레임에서 찾은 bounding box가 Dynamic Template가 된다. CSWinTT에서는 25번째 프레임마다 Confidence Score를 검사하여 Score값이 0.5 이상인 조건을 Dynamic Template를 bonding box 좌표값으로 교체하는 조건으로 여긴다. Confidence Score는 하나의 프레임에 대한 추론의 결과인 bounding box안에 찾으려는 대상 객체 존재 유무에 대한 확률을 0~1값으로 나타낸 것이다. Score값이 1에 가까울수록 bounding box안에 객체가 있을 확률이 높음을 의미한다.

i번째 프레임에 대한 추론이 시작되면, 프레임(Search Region)에 대한 전처리가 먼저 진행된다. Initial Template는 첫 프레임 추론 시에, Dynamic Template는 조건을 만족하여 bounding box 좌표값으로 교체될 때 전처리가 이뤄진다. 전처리 과정이 끝난 3개의 입력값은 Backbone으로 전달된다. Backbone은 ResNet50(He, et al., 2016)모델의 첫 번째 레이어부터 세 번째 레이어까지로 구성된 모델이며 Backbone의 결과값은 3개의 입력값에 대한 특성맵(Feature Map)이다. 특성맵은 트랜스포머 모델로 전달되어 인코더의 첫 번째 레이어를 통과하게 된다. 트랜스포머의 인코더와 디코더는 총 N개의 레이어로 구성되어 있으며, CSWinTT에서 $N=6$ 으로 설정되어 인코더와 디코더 각각 6개의 레이어를 가진다. 인코더의 i번째 레이어 출력값은 $i+1$번째 레이어의 입력값이 된다. 인코더의 마지막 6번째 레이어 출력값은 디코더의 첫 번째 레이어 입력값이 된다. 디코더 또한 i번째 레이어 출력값이 $i+1$번째 레이어 입력값이 된다. 디코더의 마지막 레이어를 통과한 결과값은 Box head(Yan, et al., 2021)로 전달된다. Box head는 여러 개의 Convolution 레이어로 이루어져 있으며, 추론의 최종 결과인 bounding box 의 좌표 정보를 도출한다.

2) 타겟모델(CSWinTT) 실행 시간비율 분석

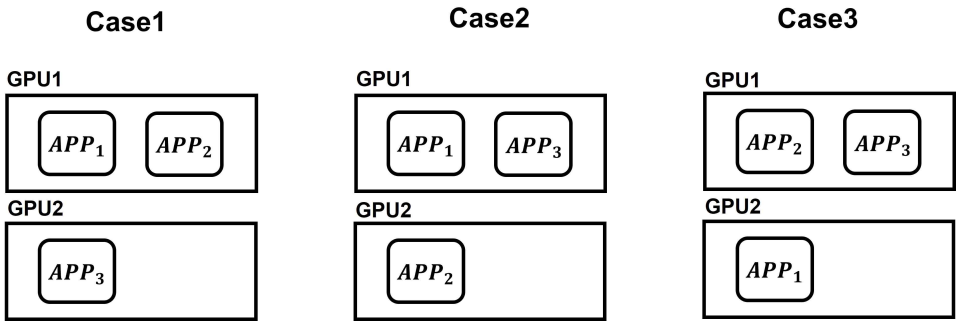
Stage		Ratio
Image load and Preprocessing		17%
Backbone		13%
Transformer	Encoder	58%
	Decoder	7%
Box head		5%

[표 2-1] 객체추적에 소요되는 각 기능 블록의 실행시간 비율

[표 2-1]은 CSWinTT 모델에서 하나의 프레임에 대한 bounding box 결과 도출 시 진행되는 각각의 과정에 대한 소요시간비율을 분석한 결과이다. 트랜스포머 모델 추론 과정이 하나의 프레임에 대한 bounding box 좌표값을 도출하는 전체 과정 중 차지하는 시간 비율이 65%로 가장 높고, 그 중 인코더의 실행시간이 전체 실행시간 대비 58%, 트랜스포머 과정 중 89%로 트랜스포머 과정의 소요 시간 중 대부분을 차지한다. 따라서 인코더 부분의 성능 개선은 전체 트랜스포머의 성능을 향상시킬 수 있는 기회를 제공한다고 볼 수 있다.

CSWinTT 내부 6개의 인코더 레이어 각각은 MHA를 수행하고 MHA연산 과정은 n개의 head연산들로 이루어져있다. 하지만 각각의 head 연산들은 데이터 의존성이 없음에도 직렬 실행된다. MHA 및 각 head 연산과정은 다음 장에서 자세히 설명한다. 본 연구에서는 이 연산들을 최대한 병렬적으로 수행하도록 하여 인코더의 실행시간을 단축시켜, 한 프레임을 처리하는 시간을 단축시키고자 한다.

제 2 절 다수의 DNN 모델 학습 시 문제 상황 및 자원배분 방법



[그림 2-2] 3개의 학습 응용이 2개의 GPU에서 실행되는 예시

1) 다수의 DNN 학습 응용 실행 시 불공정한 GPU 자원배분

[그림 2-2]는 DNN 모델을 학습하는 응용 3개가 2개의 GPU에서 실행될 때 각 GPU에 응용을 배치하는 경우를 예시한다. [표 2-2]의 Solo-Run/GPU는 하나의 GPU에서 어떤 DNN 학습 응용 하나가 종료되기까지의 시간을 나타낸다. 예를 들어 응용 A는 하나의 GPU에서 학습 시, 학습 완료시간은 t이다. 응용별로 공정한 GPU자원 분배가 이루어졌는지 판단하기 위하여 Slow-Down(이하 SD)이라는 개념을 사용한다. SD는 Solo-Run/GPU 대비 실제 학습에 소요되는시간의 비율이다.

	Solo-Run/GPU	Case 1	Case 2	Case 3
APP_1	t	$2t$ / 2	$2t$ / 2	t / 1
APP_2	$2t$	$4t$ / 2	$2t$ / 1	$4t$ / 2
APP_3	$3t$	$3t$ / 1	$6t$ / 2	$6t$ / 2

[표 2-2] [그림 2-2]에 나타낸 응용들의 완료시간

[표 2-2]에서 Case 1의 APP_2는 $\frac{4t}{2t}=2$의 SD값을 가진다. 각 Case의 SD 값을 [표 2-2]에서는 학습 소요시간 옆에 표기하였다.

2) DNN 학습 응용들의 GPU 자원배분 방법

일반적으로 많이 사용되는 딥러닝 프레임워크인 PyTorch(Paszke, et al., 2019)에서 복수개의 GPU를 DNN 모델 학습에 사용하는 방법을 예를 들어 설명한다. PyTorch에서는 DataParallel이라는 API를 사용하여 DNN 모델 학습이 하나 이상의 GPU를 동시에 사용할 수 있게 한다. 입력 파라미터는 대상이 되는 DNN 모델과 수행하고자 하는 GPU의 인덱스이다.

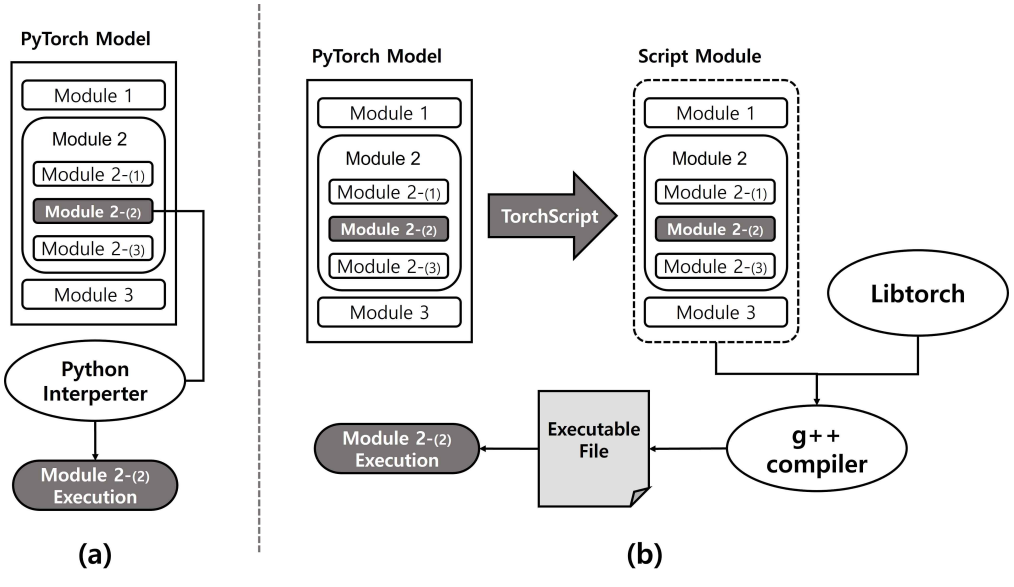
Data Parallel은 학습의 각 이터레이션 마다 다음 과정을 따른다. DNN 모델을 병렬처리에 사용할 모든 GPU에 복사한다. 미니배치 데이터를 모델이 복사된 각각의 GPU에 동등한 비율로 나누어 할당한다. 한 번의 이터레이션이 완료되면 각 GPU의 출력값을 하나의 GPU 메모리에 모은다. 이때 출력값이 모이는 GPU를 Output Device라고 한다. 모인 출력값을 이용하여 Output Device에서 손실함수의 기울기를 계산하고 병렬실행 중인 모든 GPU에 기울기 값을 복사한다. 이후 각각의 GPU에서 기울기 값을 이용한 역전파가 진행된다. 마지막 과정으로 역전파의 결과로 나온 기울기 값을 Output Device에 모아서 모델을 업데이트한다. 매개변수가 업데이트된 모델은 Output Device에만 존재한다. 따라서 다음 이터레이션에서는 Output Device의 메모리에 존재하는 모델을 병렬실행 중인 모든 GPU에 복사해야한다.

제 3 장 문제 해결 및 개선을 위한 기법 제안

제 1 절 병렬 MHA 연산 기반 트랜스포머 인코더

1) 모듈 분리

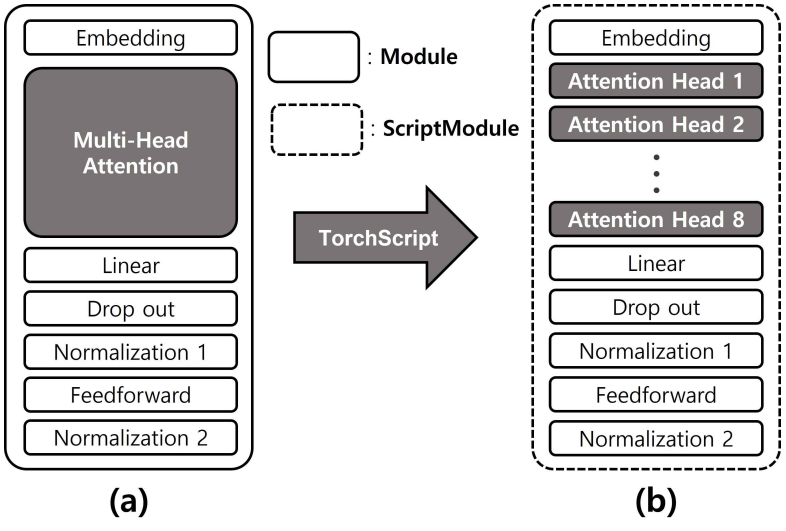
CSWinTT의 트랜스포머 인코더를 구성하는 6개 레이어는 각각 PyTorch 모델 관점에서 보면 하나의 모듈(Modules-Pytorch 1.13 documentation, online)이다. 또한 각 레이어는 복수 개의 모듈들의 집합으로 이루어져 있다. 본 연구에서는 궁극적으로 MHA 내부 연산 과정을 병렬처리 하는데 그 목적을 둔다. 이를 위하여 PyTorch 기반 MHA 내부 모듈들을 쓰레드에서 실행되는 최소단위로 설정하고 이를 멀티쓰레딩 처리한다.



[그림 3-1] 파이썬 환경, C++환경에서의 PyTorch 모델 실행

더불어 우리는 보다 빠른 실행을 보장하기 위해서 순수 Python 모델로 구성된 PyTorch 모델을 C++기반 모델로 우선 변경한다. 이를 위하여 트랜스포머 인코더의 각 레이어를 TorchScript를 사용하여 ScriptModule형태로 변환하였다(Torchscript-Pytorch master documentation, online).

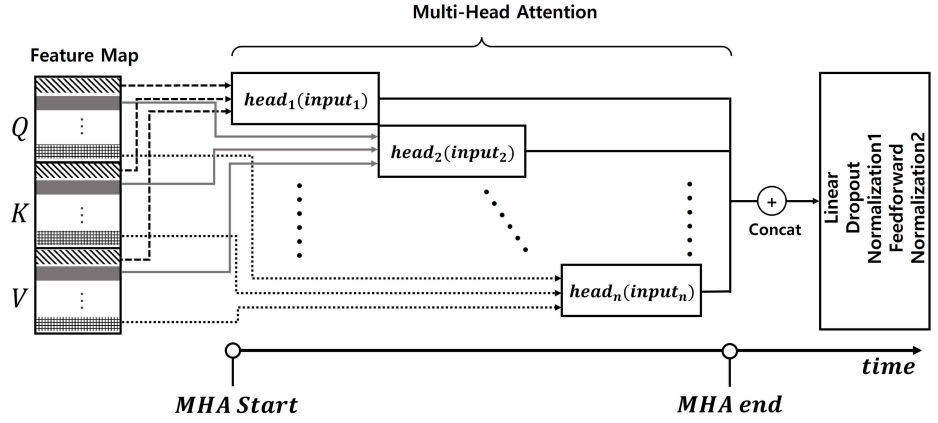
[그림 3-1]은 여러 모듈의 집합인 PyTorch 모델이 실행되는 과정을 나타낸다. 모듈 2의 경우 하나의 모듈 또한 여러 모듈의 집합인 경우를 예시한다. (a)는 Python 환경에서의 실행 과정이고, (b)는 C++ 환경에서의 실행 과정이다. Python 환경에서는 컴파일 과정 없이 Python 인터프리터가 각각의 모듈을 순차적으로 실행시킨다. C++ 환경에서는 우선 PyTorch 모델 전체가 TorchScript를 사용하여 ScriptModule로 변환된다. 이때 안에 담고 있는 모듈의 형태는 유지된다. ScriptModule을 사용하기 위해 Libtorch 라이브러리를 사용하여 컴파일을 진행한 후 Python의 인터프리터 방식이 아닌 실행파일을 생성 후 실행한다.



[그림 3-2] TorchScript를 통한 인코더 레이어 모듈 분리

[그림 3-2]의 (a)는 인코더 내 6개의 레이어중 하나의 레이어를 구성하는 모듈들을 나타낸 것이고, (b)는 (a)를 본 연구의 사용 목적에 맞춰 TorchScript를 사용하여 ScriptModule형태로 변환한 후 ScriptModule을 구성하는 모듈들을 나타낸 것이다. 기존의 CSWinTT 트랜스포머 인코더의 한 레이어 속 MHA과정은 하나의 모듈로 이루어져 있어 하나의 쓰레드 안에서 실행되었다. 복수개의 쓰레드를 활용하여 MHA속 각각의 head 연산들을 병렬 실행하기 위해서 여러 개의 모듈로 분리했으며, 하나의 Attention Head를 기준으로 하여 MHA를 8개의 모듈로 분리한 ScriptModule을 생성했다. 또한 C++ 환경에서 전체 CSWinTT 모델을 구동시키기 위해 Backbone, 트랜스포머 디코더, Box head 모듈 또한 ScriptModule 형태로 변환했다.

2) MHA 연산과정



[그림 3-3] 인코더 내 한 레이어 실행 과정

[그림 3-3]은 인코더의 한 레이어가 실행되는 과정을 시간 축과 함께 나타낸 것이다. MHA의 각 head연산에 필요한 Query, Key, Value 값은 특성맵을 특정 차원을 기준으로 3분할한 Q, K, V 각각을 MHA의 head연산 개수인 n개로 나눈 것 중 하나이다. 식(1)은 이를 수식으로 나타낸 것이다. $head_i(1 \leq i \leq n)$는 MHA의 총 n개의 head연산 중 하나를 의미하며 $head_i$의 입력값인 $input_i$는 식(2)처럼 나타낼 수 있다.

$$Q = Concat(Q_1, Q_2, \ldots Q_n)$$
$$K = Concat(K_1, K_2, \ldots K_n)$$
$$V = Concat(V_1, V_2, \ldots V_n) \quad (1)$$

$$input_i = \{Q_i, K_i, V_i\} \; (1 \le i \le n) \quad (2)$$

MHA의 각 head연산들은 각각 독립된 입력값을 가지게 된다. Python 환경에서 구현된 CSWinTT의 MHA는 head연산들이 직렬실행으로 이루어져 있기에 $head_i$ 가 시작되는 시점은 $head_{i-1}$ 이 종료된 이후이다. 식 (3)은 i번째 head연산을 나타낸 수식이며 d_k는 Key의 차원이다.

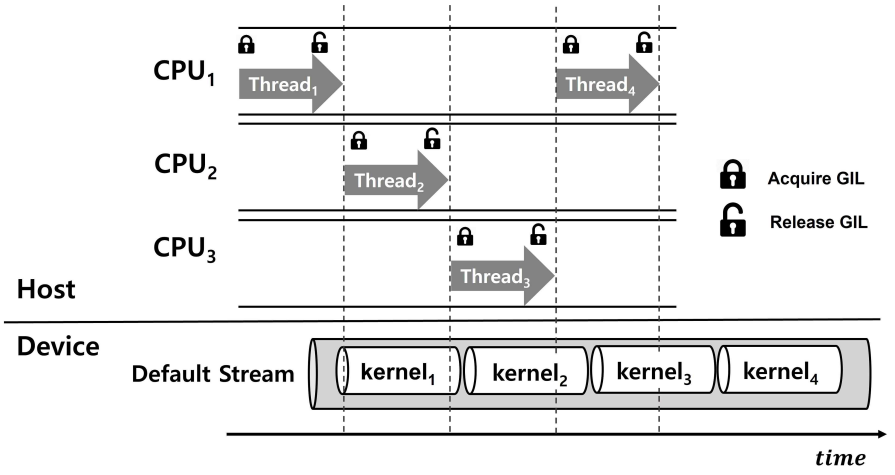
$$head_i(input_i) = Attention(Q_i, K_i, V_i)$$
$$= softmax(\frac{Q_i K_i^T}{\sqrt{d_k}}) V_i \quad (3)$$

MHA 종료 후 각 head의 결과값을 모두 합쳐 하나의 텐서로 만들고 레이어의 남은 과정(Linear～Normalization2)을 순차적으로 수행한다.

3) 병렬 MHA 연산 적용

[그림 3-4], [그림 3-5]는 3개의 코어로 구성된 CPU에서 4개의 쓰레드로 구성된 하나의 프로세스가 동작하는 예시이다. 각각의 쓰레드가 실행되면서 GPU에 모듈실행을 담고 있는 커널을 전달한다.

[그림 3-4]는 Python 환경에서 별도의 개별적인 스트림 기술(Tensor CUDA Stream API-PyTorch master documentation, online)을 적용하지 않고 디폴트 스트림 하나를 사용하여 각각의 쓰레드들이 커널을 전달하는 과정을 보여준다. Python 환경에서는 GIL(Global Interpreter Lock)정책에 의하여

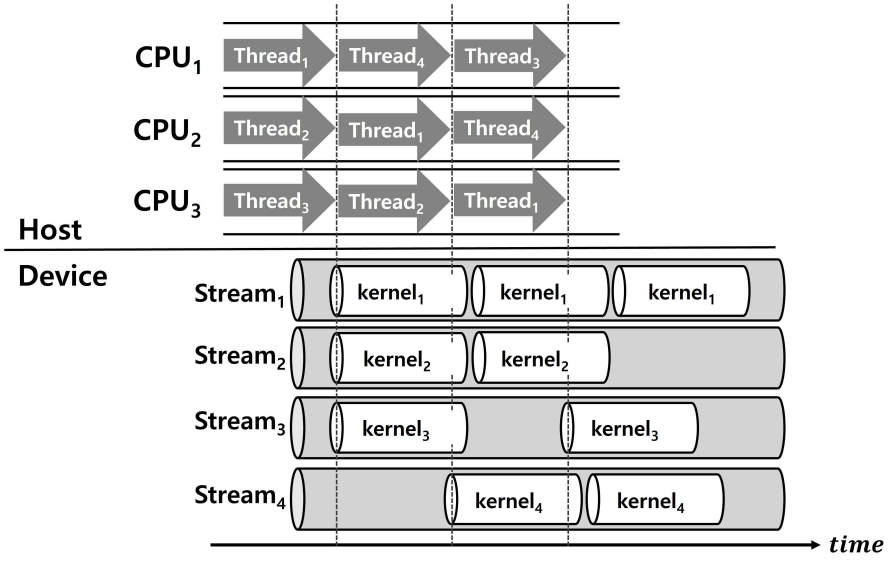


[그림 3-4] 파이썬 환경에서 각 쓰레드들이 커널을 전달하는 과정

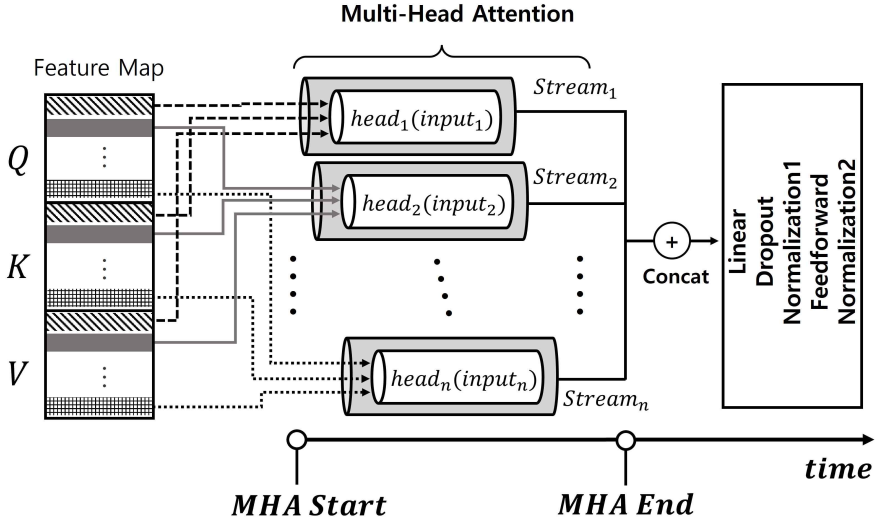
특정 시점에서 하나의 쓰레드만이 Python 인터프리터를 실행시킬 수 있다 (What Is the Python Global Interpreter Lock(GIL)?, online). 하나의 쓰레드가 GIL을 획득하면 다른 쓰레드들은 해당 쓰레드가 획득한 GIL을 해제 할 때 까지 Python 인터프리터를 실행할 수 없다. 따라서 Python 환경에서는 멀티쓰레드로 구성된 프로세스라도 멀티쓰레드의 병렬실행이 실질적으로 불가능하다. 또한 모든 커널이 디폴트 스트림에서 실행되기 때문에, 앞선 커널이 종료되어야만 그 다음 커널의 실행이 시작된다. [그림 3-4]에서 각 커널들이 GPU에 전달되는 과정을 살펴보면, 1) 호스트(CPU)쪽에서 GIL에 따른 병렬적인 실행이 보장되지 못하고 2) 디폴트 스트림 사용에 따른 직렬화된 커널 전달이 이루어짐을 알 수 있다.

[그림 3-5]는 이러한 문제를 해결한 예시를 보여준다. C++ 실행환경을 사용함으로써 멀티쓰레딩이 가능하며 각각의 쓰레드들이 병렬적으로 수행됨을 알 수 있다. 또한 커널 전달 역시 개별적인 스트림을 활용하여 GPU에 전달되는 과정을 보여준다. C++ 환경에서는 Python 환경에서 발생하는 GIL 제약사항의 영향을 받지 않기 때문에 3개의 CPU에서 각각 동시에 쓰레드가 실행되는 것이 가능하다. 각각의 스트림 안의 커널은 순차적으로 실행되고, 서로 다른 스트림에 할당된 커널은 GPU에서 함께 동작하는 것이 가능하다. 따라서 스트림의 개수가 증가할수록, 같은 시간동안 더 많은 커널을 실행시킬

수 있어 병렬성이 향상된다.



[그림 3-5] C++ 환경에서 각 쓰레드들이 커널을 전달하는 과정



[그림 3-6] MHA병렬연산 적용 후 인코더 내 한 레이어 실행 과정

[그림 3-6]은 쓰레드별 스트림을 지정한 멀티쓰레드 환경에서 MHA을 병렬실행 했을 때 시간에 따른 인코더의 한 레이어 실행과정을 나타낸다. 특성

맵에서 각각의 head연산에 입력값을 할당하는 과정은 그림 4와 동일한 과정을 따른다. 각각의 head연산 시작지점이 조금씩 차이가 나는 이유는 head연산의 입력값이 순차적으로 전달되어, 입력값의 전달시점이 각 head마다 다르기 때문이다. $head_i$가 $input_i$를 전달받아 연산이 시작되면, $head_{i+1}$은 $head_i$가 연산중인 것과 관계없이 $input_{i+1}$을 전달받는다. $head_i$ 는 $Stream_i (1 \le i \le n)$에 커널을 보내고 $head_n$이 종료되면 각 head연산의 결과값을 하나로 합치고 [그림 3-3]과 동일하게 레이어의 남은 과정을 순차적으로 수행한다.

제 2 절 적응형 SD 기반 미니배치 재분배 기법

1) 제안 기법

본 논문에서는 다수의 DNN 모델 학습 응용을 실행할 때, 응용들의 GPU 자원의 공정한 사용 정도를 SD로 나타낸다. 어떠한 응용 App_i의 SD값은 SD_i로 나타낸다. 동시에 학습이 진행되는 N개의 응용이 있을 때, 모든 응용의 SD값 집합은 $SD_{set} = \{SD_{1,} SD_{2,} \cdot \cdot \cdot, SD_N\}$ 이다. $1 \le i, j \le N$ 을 만족하는 임의의 SD_i, SD_j에 대하여 그 차이 값이 가장 큰 것을 $SD_{\max}^{diff}$라고 하면, 식 (1)처럼 나타낼 수 있다.

$$SD_{\max}^{diff} = \max(\mid SD_i - SD_j \mid) \quad (1)$$

본 논문에서 제안하는 기법은 다수의 DNN 모델 학습 응용을 실행할 때, 모든 응용들이 공정하게 GPU자원을 사용할 수 있도록 $SD_i \in SD_{set}$ 을 만족하는 SD_i를 일정한 간격으로 관찰한다. 이 때 가장 큰 SD값을 가지는 응용을 가장 GPU 자원을 공정하게 할당받지 못하는 상태로 볼 수 있다. 이를 해결하기 위하여 일정한 주기마다 시스템의 가용 GPU 자원 상태를 파악한 후,

가장 큰 SD값을 가지는 응용의 미니배치 데이터를 가장 최적으로 나누어 각각의 GPU에 할당하도록 한다. 이러한 방법을 통하여 가장 큰 SD값을 기록하는 응용의 SD값을 줄일 수 있고, DNN 모델 학습이 진행됨에 따라 더 공정하게 GPU자원이 모든 응용에게 배분되는 결과를 기대할 수 있다.

2) 학습 응용에서 각 GPU에 분배하는 미니배치 데이터의 비율

매 이터레이션 마다 각 GPU에 할당된 응용의 미니배치 데이터 비율을 Data-Ratio(이하 DR)로 정의한다. 이는 미니배치 데이터의 비율을 리스트 형태로 표현한 것이다. DR의 각 요소는 0이상 10이하의 정수 값으로 구성되고, 각 요소는 GPU에 할당할 미니배치 데이터의 비율을 나타낸다. DR의 모든 요소의 합은 언제나 10이 된다. 예를 들어, 4개의 GPU (GPU0~GPU3)에 미니배치 데이터가 10%, 20%, 30%, 40%로 할당되는 경우를 DR로 표현하면 [1,2,3,4]로 나타낼 수 있다. [표 3-1]은 4개의 GPU에서 2개의 DNN 모델 학습 응용이 실행될 때 각 학습 응용의 DR에 따라 미니 배치 데이터 비율이 GPU에 분배되는 비율을 나타낸다.

	APP_1	APP_2
DR	[10,0,0,0]	[0,7,3,0]
Used GPU	GPU0	GPU1, GPU2
Mini Batch Data Ratio	GPU0 - 100%	GPU1 - 70% GPU2 - 30%

[표 3-1] 4개의 GPU에서 2개의 DNN 학습 응용이 실행되는 예시

[표 3-1]에서 App_1은 하나의 GPU(GPU0)만을 사용하며 미니배치 데이터가 나눠지지 않은 상태이다. App_2는 GPU1에 미니배치 데이터의 70%, GPU2에 미니배치 데이터의 30% 그리고 GPU0과 GPU3은 사용하지 않는 상태를 나타낸다. SD_i는 응용의 DR_i에 따라 달라진다. 이를 정량화하면 식 (2)와 같

다.

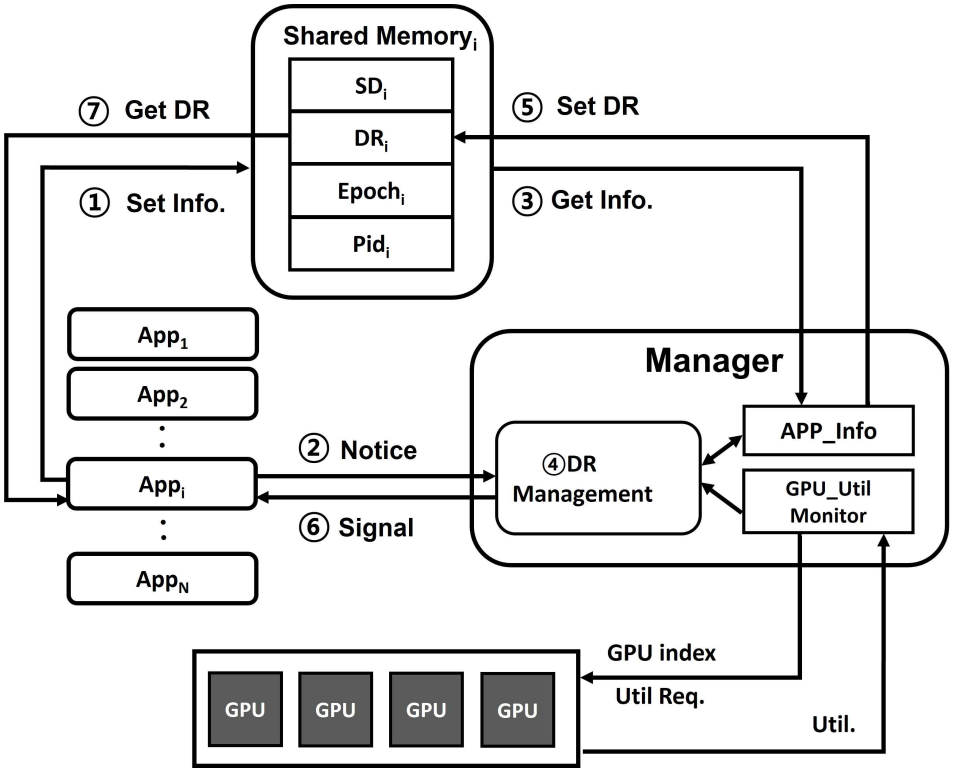
$$SD_i(DR_i) = \frac{T_{current} - T_{start} + iter_left * iter_time(DR_i)}{solorun_time} \quad (2)$$

식 (2)에서 $solorun_time$은 단일 GPU에서 APP_i가 단독으로 실행되었을 때 학습이 완료되기까지 걸린 시간을 의미한다. $solorun_time$은 오프라인 환경에서 사전에 측정된 값을 이용한다. $T_{current}$는 현재의 시각, T_{start}는 응용의 시작 시점을 나타내므로 $T_{current} - T_{start}$는 측정시점까지 학습 응용의 실행시간을 의미한다. $iter_time(DR_i)$는 App_i에 할당된 미니배치 데이터 비율이 DR_i일 때, 현재 에포크에서 측정한 1이터레이션 수행 시 걸리는 평균 시간이다. $iter_left$는 $T_{current}$부터 전체 학습이 완료되는 시간까지 남은 이터레이션의 수를 의미한다.

3) 전체 시스템 동작 방식

[그림 3-7]은 APP_i의 DR_i가 갱신되는 과정을 예시한다. 모든 응용은 자신만의 공유메모리를 가지고 있고, 공유메모리를 통해 각 응용과 Manager가 데이터를 서로 주고받는다. 일정 횟수의 이터레이션 마다 공유메모리에 현재 상태에 대한 SD_i값을 기록한다. 본 논문에서는 5번의 이터레이션 마다 SD_i값을 기록하였다.

[그림 3-7]의 Manager는 $SD_{\max}$가 설정한 임계값을 넘는 경우 SD값이 가장 높은 APP_i의 DR_i를 재설정하여 SD_i를 낮추는 역할을 한다. DR_i 재설정 방식은 상황에 따라 GPU의 사용률 또는 응용의 SD값을 이용한다. 이때 각 GPU의 사용률은 Manager가 GPU 디바이스 드라이버에게 요청하여 일정한 주기로 업데이트된 GPU 사용률 값을 기록한다. [그림 3-7]에서는 GPU_Util Monitor로 표현되었다. APP_i의 DR_i가 갱신되는 과정을 그림[그림 3-7]의



[그림 3-7] 적응형 SD기반 미니배치 재분배 시스템의 전체 동작 과정

①~⑦을 통하여 설명하면 다음과 같다.

APP_i는 현재 수행중인 에포크가 완료되는 시점 즉, 에포크의 마지막 이터레이션에서 SD_i를 계산한 후, ①공유 메모리에 SD_i, DR_i, 현재 에포크 횟수 그리고 프로세스 번호(Pid)등 정보를 기록한다. ②APP_i는 자신이 공유메모리에 정보 기록을 끝냈음을 알리기 위하여 Notice 신호를 Manager에게 보낸다. ③Manager가 모든 응용의 공유메모리에 접근하여 기록된 정보들을 가져와 APP_Info의 $APP_1 \sim APP_N$에 대한 정보를 갱신한다. APP_i뿐만 아니라 모든 응용의 공유메모리에 접근하여 정보를 가져오는 이유는 APP_i가 Notice 신호를 보낸 시점을 기준으로 모든 응용의 정보를 최신화하기 위해서 이다. 학습과정의 마지막 에포크가 끝난 응용이 Notice 신호를 Manager에게 보내면 이 응용은 학습과정을 완료하여 종료될 응용이기 때문에 해당응용에 대한

정보는 APP_Info에서 삭제된다. ④DR Management는 [그림 3-8]의 알고리즘에 의해서 현재 Manager에 신호를 보낸 APP_i에게 할당할 DR값 즉, DR_i를 산출한다. DR_i를 산출하는 알고리즘에 대한 자세한 내용은 이어지는 4)에서 설명한다. DR Management 알고리즘에 따라 APP_i에 주어질 새로운 DR_i가 도출되면 APP_Info의 APP_i정보에서 DR_i를 갱신한다. Manager는 ⑤ APP_i의 공유메모리에 DR_i를 기록하고 ⑥APP_i에게 Signal을 보내서 공유메모리의 값이 갱신되었음을 알린다. ⑦APP_i는 공유메모리에 접근하여 갱신된 DR_i로 미니배치 데이터 비율을 수정하고 다음 에포크를 진행하여 학습을 이어간다.

4) GPU에 할당할 미니배치 비율 설정 알고리즘

SD_{set}에서 가장 큰 SD값을 $SD_{\max}$라고 한다. APP_i의 SD값인 SD_i가 $SD_{\max}$가 아니거나, $SD_{\max}$값이 임계값보다 작은 경우에는 DR_i는 기존과 똑같이 유지된다.([그림 3-8]의 라인 11~13)

APP_i의 학습이 진행될 때 수행속도의 느림과 빠름은 APP_i의 미니배치 데이터 비율뿐만 아니라, 같이 수행되는 다른 응용들의 미니배치 데이터 비율 또한 영향을 준다. 따라서 현재 APP_i에게 할당된 GPU 각각의 사용률과 할당되지 않은 GPU의 사용률 역시 새로운 DR_i를 생성할 때 반드시 고려되어야 한다. 이를 세분화하면 다음의 두 가지 경우이다. ⅰ)SD값이 큰 응용이 사용률이 높은 GPU에 할당되어 있고, 그 GPU의 사용률과 가장 사용률이 낮은 GPU 사이의 차이가 임계값 $Util_{th}$보다 클 경우. 다른 하나는 ⅱ)그 차이가 임계값 $Util_{th}$보다 작을 경우이다.

ⅰ)의 경우는 [그림 3-9]의 DR_Update_Util 함수에 나타내었다. 먼저 사용률이 가장 낮은 GPU를 $GPU_{\min}$이라 한다. APP_i가 사용중인 GPU의 집합을

Algorithm 1 DR Management

GPU^i_{used}: A set of GPUs utilized by App_i
GPU_{set} : $\{GPU_0, GPU_1, \ldots GPU_m\}$, m GPUs in the system
n : Predefined period

1: **Function** DR_management **(**SD_i, DR_i, SD_{set}**)**
2: **if** # of Apps ≤ m
3: GPU_{min} ← A GPU with the smallest Util. in GPU_{set}
4: **for** g in GPU_{set}
5: **if** g == GPU_{min}
6: new_DR_i[a] ← 10
7: **else**
8: new_DR_i[a] ← 0
9: GPU_{set} ← GPU_{set} - $\{GPU_{temp}\}$
10: **return** new_DR_i
11: **if** SD_i != max(SD_{set}) **or** $SD^{diff}_{max} < SD_{th}$
12: new_DR_i ← $current_DR_i$
13: **return** new_DR_i
14: **else**
15: GPU_{max} ← A GPU with the biggest Util. in GPU^i_{used}
16: GPU_{min} ← A GPU with the smallest Util. in GPU_{set}
17: **if** (Util. of GPU_{max}- Util. of GPU_{min})> $Util_{th}$
18: new_DR_i ← DR_Update_Util(GPU^i_{used}, GPU_{set}, GPU_{min})
19: **else**
20: GPU_{dest} ← A GPU which has the minimum Avg. SD value in GPU_{set}
21: GPU_{src} ← A GPU indexed by the biggest value in DR_i
22: new_DR_i ←DR_Update_SD(SD_i, DR_i, SD_{set}, GPU_{src}, GPU_{dest})
23: **return** new_DR_i

[그림 3-8] DR Management 알고리즘 수도코드

GPU^i_{used}라고 할 때, $GPU_{\min}$이 GPU^i_{used}에 포함되지 않는다면 $GPU_{\min}$을 GPU^i_{used}에 포함시킨 후 각 GPU에 분배할 미니배치 데이터 비율을 계산한다. $GPU_{\min}$부터 GPU^i_{used}중 가장 사용률이 높은 GPU까지 APP_i의 미니배치 데이터 비율이 감소하는 형태로 할당한다([그림 3-9]의 라인 2~11).

예를 들어 App_i의 DR_i=[5,5,0,0]인 상태라면, GPU^i_{used}는 GPU0와 GPU1이고, 각각 미니배치 데이터가 50%씩 나눠져 있는 상태이다. GPU0~GPU3의 남은 활용률을 순서대로 20%, 30%, 80%, 50%라고 가정한다. 각 GPU의 사

Algorithm 2 DR Update by Util or SD

GPU^i_{used}: A set of GPUs utilized by App_i
GPU_{set} : $\{GPU_0, GPU_1, \ldots GPU_m\}$, m GPUs in the system
GPU_{min} : A GPU with the smallest Util. in GPU_{set}
GPU_{dest} : A GPU which has the minimum Avg. SD value in GPU_{set}
GPU_{src} : A GPU indexed by the biggest value in DR_i

1: **Function** DR_Update_Util(GPU^i_{used}, GPU_{set}, GPU_{min})
2: **if** $GPU_{min} \notin GPU^i_{used}$
3: $GPU^i_{used} \leftarrow GPU^i_{used} \cup \{GPU_{min}\}$
4: Util. remained ← 0
5: **for** j in GPU^i_{used}
6: Util. remained ← Util. remained + 100 - Util. of GPU_j
7: **for** k in GPU_{set}
8: **if** k in GPU^i_{used}
9: $new_DR_i[k] \leftarrow \left\lfloor \frac{100 - Util.of\ GPU_k}{Util.remained} * 10 + 0.5 \right\rfloor$
10: **else**
11: new_DR_i[k] ← 0
12: **return** new_DR_i

13: **Function** DR_Update_SD(SD_i, DR_i, SD_{set}, GPU_{src}, GPU_{dest})
14: $E_{mbr} \leftarrow \frac{SD_i - 1}{DR_i[GPU_{src}]}$
15: $r \leftarrow \left\lfloor \frac{SD_i - (\max(SD_{set}) - \min(SD_{set}))/2}{E_{mbr}} \right\rfloor$
16: $new_DR_i \leftarrow current_DR_i$
17: $new_DR_i[GPU_{src}] \leftarrow new_DR_i[GPU_{src}]$ - r
18: $new_DR_i[GPU_{dest}] \leftarrow new_DR_i[GPU_{dest}] + r$
19: **return** new_DR_i

[그림 3-9] DR 업데이트 알고리즘 수도코드

용률은 GPU0부터 80%, 70%, 20%, 50% 이기 때문에 사용률이 가장 낮은 GPU2가 $GPU_{\min}$이 된다. 알고리즘에 따라 GPU2가 App_i의 미니배치 데이터를 할당받을 새로운 GPU가 되고, 이 때 GPU^i_{used}는 {GPU0, GPU1, GPU2}이다. GPU^i_{used}의 각 GPU의 남은 활용률을 활용률의 총 합으로 나눈 값은 순서대로 0.09, 0.18, 0.72가 되고, 이 값들을 합이 100%가 되는 비율로 변환하면 10%, 20%, 70%이다. 따라서 App_i에게 새로 할당되는 DR_i는 [1,2,7,0]이다.

ii)의 경우는 APP_i가 미니배치 분배를 위해 사용 중인 GPU중 가장 사용률이 높은 GPU를 $GPU_{\max}$로 한다. $GPU_{\max}$와 $GPU_{\min}$의 사용률 차이가 $Util_{th}$보다 작으면서 $SD_{\max}^{diff} > SD_{th}$ 인 경우에 해당한다. 이 경우, 각 GPU간 사용률 차이가 적기 때문에 i)에서 사용한 사용률을 기준으로 DR을 새로 설정하는 알고리즘을 적용하기 적절하지 않다. 따라서 GPU를 사용 중인 응용들의 SD값들을 사용하여 새로운 DR_i를 찾아내는 알고리즘을 적용한다. APP_i가 사용하는 GPU중 APP_i의 미니배치 데이터 비율이 가장 큰 것을 GPU_{src}라고 한다. GPU_{src}로부터 일정 비율(r, $(0 \le r \le 10)$)의 미니배치 데이터를 가져갈 GPU를 GPU_{dest}로 설정한다. 이후 DR_Update_SD 함수에 의하여 새로운 DR_i를 계산한다. 먼저 GPU_{src}의 미니배치 비율 당 APP_i의 성능저하에 미치는 영향을 값으로 표현한 E_{mbr}를 계산한다. 이후 GPU_{dest}로 옮겨질 미니배치 데이터 비율 r을 SD_i와 SD_{set}의 최댓값과 최솟값의 평균값의 차이를 E_{mbr}로 나누어 구한다. 마지막으로 기존의 DR_i에서 GPU_{src}의 인덱스 요소 값은 r만큼 빼고 GPU_{dest}의 인덱스 요소 값은 r만큼 더하여 새로운 DR_i를 만든다.([그림 3-9]의 라인 14~18)

DR_i값이 변경되는 또 다른 조건이 하나 더 존재한다. App_i가 Manager에게 Notice 신호를 보냈을 때 학습이 진행중인 응용의 개수가 시스템의 전체 GPU 개수와 같거나 적은 경우이다. 이 경우 1개 GPU에서 1개의 응용만 실행된다면 더 이상 모든 응용에게서 Slow-Down이 발생하지 않는다. 이 때 App_i는 활용률이 가장 작은 GPU에서 모든 미니배치 데이터를 실행하도록 DR_i가 변경된다. App_i이후에 Manager에게 Notice신호를 보내는 다른 응용들은 App_i가 사용중인 GPU를 제외하고 가장 활용률이 낮은 GPU부터 순서대로 모든 미니배치 데이터가 할당된다. ([그림 3-8]의 라인 2~10)

제 4 장 실험

제 1 절 병렬 MHA 연산 적용에 따른 추론시간 감소 실험

본 절에서는 Python 환경에서 실행되던 CSWinTT 모델을 앞서 언급한 기법에 따라 멀티쓰레드를 사용하여 MHA의 각 head연산을 병렬실행 하였을 때의 결과를 보여주는 실험 내용을 다룬다.

1) 실험환경

실험에 사용된 시스템은 [표 4-1]의 CPU 32개와 GPU 4개로 구성된 리눅스 운영체제 기반 서버이다. 실험 시에는 1개의 GPU만을 사용하였다.

GPU (NVIDIA RTX A6000)	10,752 CUDA Cores 336 Tensor Cores 84 RT Cores 48GB Memory 768GB/sec Memory Bandwidth 309.7 TFLOPS (FP32) PCI Express 4.0x16
CPU (AMD Ryzen Threadripper PRO 3955WX)	16-Core 32 Threads 64MB L3 Cache PCI Express ver. 4.0
OS	Ubuntu 20.04
CUDA ver.	CUDA 11.6

[표 4-1] 단일 DNN 추론 실험환경

실험은 총 3가지 경우에 대한 비교를 다룬다. Case-A는 Python 환경에서 기존의 CSWinTT 모델을 추론하였을 때의 결과이다. Case-B는 TorchScript를 사용하여 Backbone, 트랜스포머 인코더와 디코더, Box head를 ScriptModule로 변환한 것을 이용하여 C++환경에서 CSWinTT모델을 추론하였을 때의 결과이다. Case-C는 Case-B의 환경에서 추가적으로 그림 3에 나타난 MHA 분리작업을 거친 ScriptModule을 사용하여 멀티쓰레드로 MHA각 head연산을 병렬실행하고 각각의 쓰레드가 사용할 스트림을 할당하였을 때의 결과이다.

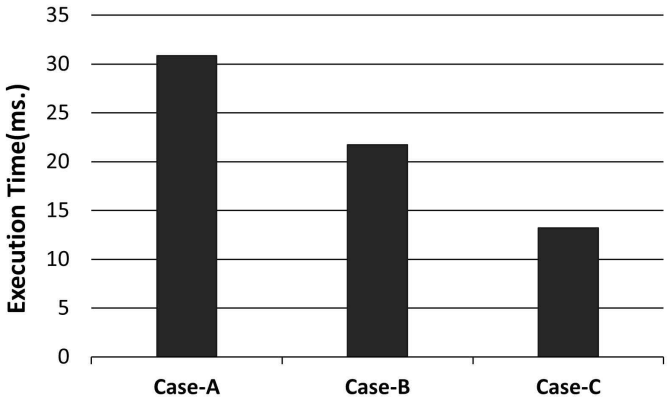
추론 시에는 TrackingNet 데이터셋(Muller, et al., 2018)을 사용하였다. TrackingNet 데이터셋은 총 510개의 데이터로 구성되어 있고, 각각의 데이터는 각각 다른 프레임수로 구성되어 있다.

2) 실험결과

실험결과는 하나의 프레임에 대한 bounding box 도출 시 트랜스포머 인코더의 평균 실행시간 비교([그림 4-1]), 데이터셋의 전체 데이터에 대한 평균 FPS값 비교([그림 4-2]), 마지막으로 1프레임을 추론하는 전체 실행시간에서 이미지 로드와 전처리 과정 과 트랜스포머 인코더 과정이 차지하는 실행시간 비율에 대한 비교를 다룬다([그림 4-3]).

가) 각 Case의 인코더 실행시간 비교

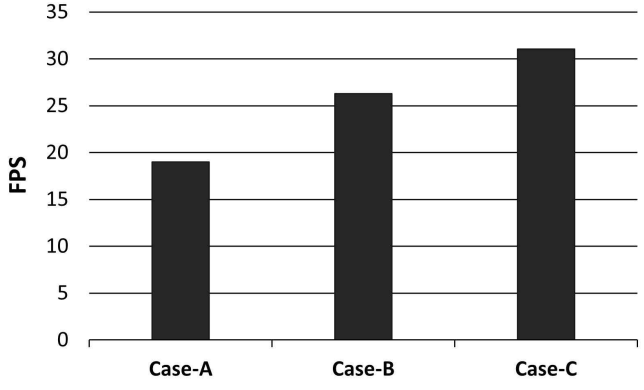
[그림 4-1]의 Case-A와 Case-B를 비교하였을 때 TorchScript를 이용한 ScriptModule을 C++환경에서 실행하는 것으로 트랜스포머 인코더의 실행시간이 30.84ms에서 21.71ms로 29.6% 감소하였다. Case-B와 Case-C를 비교하였을 때 동일한 C++ 환경에서 트랜스포머 인코더 MHA를 모듈단위로 분리하고, 멀티쓰레딩을 적용한 병렬실행과, 커널이 실행될 스트림을 지정해준 Case-C의 실행시간이 13.23ms로 Case-B에 비하여 39% 감소하였다. 이를



[그림 4-1] 인코더 평균 실행시간

통해 MHA연산을 head연산 단위로 분리하여 병렬처리 한 것이 효과가 있음을 알 수 있다. 또한 Case-A 대비 트랜스포머 인코더의 한 레이어 실행시간이 56.8% 감소하여 기존의 Python 환경에서 CSWinTT를 추론하였을 때와 비교하여 인코더 레이어의 실행시간이 절반이 이상 줄어드는 효과가 있었다.

나) 각 Case의 평균 FPS 비교

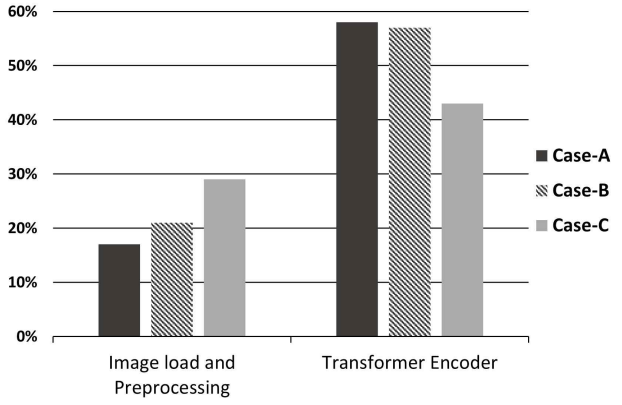


[그림 4-2] 평균 FPS 비교

FPS값은 추론 시 1초당 처리하는 프레임의 개수를 의미하며 FPS값이 높

을수록 한 프레임에 대한 추론속도가 빠른 것을 의미한다. [그림 4-2]의 Case-A에서 FPS값은 19, Case-B에서 FPS값은 26.3으로 ScriptModule을 이용하여 C++환경에서 실행하였을 때 FPS값이 38.4% 증가하였다. Case-B와 Case-C를 비교하였을 때, Case-C의 FPS값은 31.04로 Case-B에 대비하여 17.8% 증가하였다. 또한 Case-C는 Case-A 대비 FPS값이 63.3% 증가하여 기존 Python 환경에서 실행되던 것 보다 ScriptModule의 최적화 효과와 MHA각 head연산의 병렬실행 효과가 있음을 확인하였다.

다) 각 Case의 이미지 전처리와 인코더 실행시간 비율 비교



[그림 4-3] 전체 실행과정 중 이미지 전처리와 인코더 실행시간의 점유율 비교

[그림 4-3]의 그래프에서 트랜스포머 인코더가 전체 실행시간에서 차지하는 비율은 Case-A와 Case-B 각각 58%, 57%로 비슷한 비율을 가진다. 하지만 Case-C에서는 43%로 Case-A와 Case-B에 비하여 낮은 비율을 가진다. 이는 트랜스포머 인코더의 실행시간이 줄어들어 전체 실행시간 중 인코더가 차지하는 비율이 줄어들었기 때문이다. 반대로 이미지 로드와 전처리가 전체 추론과정 중 차지하는 비율은 Case-C, Case-B, Case-A 각각 29%, 21%, 16.5%로 Case-C, Case-B, Case-A 순으로 높게 나타났다. 이미지 로드와

전처리 과정의 실제 수행시간은 3가지 Case모두 비슷하기 때문에 전체 추론 시간이 가장 많이 감소한 Case-C가 Case-B와 Case-A에 비하여 이미지 로드와 전처리의 비율이 높게 나타났다.

제 2 절 다수의 DNN 학습 시 GPU 자원배분 실험

본 절에서는 다수의 DNN 학습 시 [그림 3-8], [그림 3-9]의 DR Management 알고리즘을 적용하였을 때 학습이 진행 중인 모든 응용이 GPU 자원을 최대한 공정하게 할당받는 것을 증명하는 실험 내용을 다룬다.

1) 실험환경

실험에 사용된 시스템은 [표 4-2]의 CPU14개와 GPU4개로 구성된 리눅스 운영체제 기반 서버 이다.

GPU (Quadro RTX 6000)	4608 CUDA Cores 576 Tensor Cores 72 RT Cores 24GB Memory 16.3 TFLOPS (FP32) 672GB/sec Memory Bandwidth PCI Express 3.0x16
CPU (Intel i9-10940X)	14-Croes 28 Threads 19.25MB L3 Cache PCI Express ver. 3.0
OS	Ubuntu 18.04
CUDA ver.	CUDA 10.2

[표 4-2] 다수의 DNN 학습 실험환경

실험에는 DenseNet121(Huang, et al., 2017), ResNet50, AlexNet(Krizhevsky, et al., 2017), ResNext(Xie, et al., 2017), ShuffleNet v2(Ma, et al., 2018), MnasNet(Tan, et al., 2019) 총 6종류의 DNN 모델을 사용하여 결과를 확인한다. 이들 모델들은 이후 실험부에서 표기의 간소화를 위하여 순서대로 Dense, Res, Alex, ResNext, Shuffle, Mnas로 나타낸다. 학습에 사용하는 데이터셋은 ImageNet 데이터셋(Deng, et al., 2009)을 사용하였다.

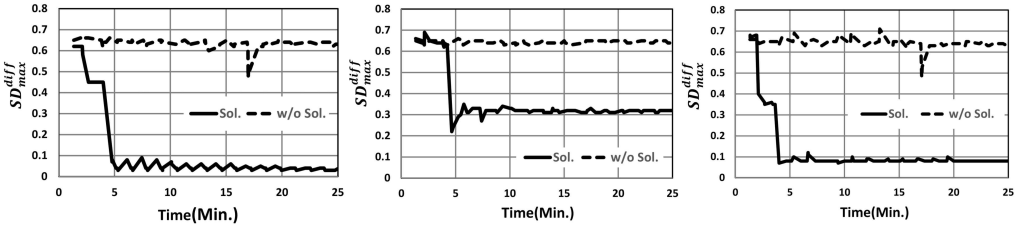
실험은 총 3가지 경우로 나누어 진행하였다. 3종류 DNN을 학습하는 6개 응용을 4개의 GPU에서 실행시키는 경우([그림 4-4 ~ 그림 4-6]의 (a)), 3종류 DNN을 학습하는 5개 응용을 3개의 GPU에서 실행시키는 경우([그림 4-4 ~ 그림 4-6] 의 (b)), 그리고 6종류 DNN을 학습하는 6개 응용을 4개의 GPU에서 실행시키는 경우([그림 4-4 ~ 그림 4-6] 의 (c))이다. 3종류의 DNN 만을 사용할때는 Dense, Res, Alex를 사용하였다. (a)의 경우 종류별로 각각 2개씩, (b)의 경우 Alex하나를 제외하여 Dense 2개, Res 2개, Alex 1개를 사용하였다. GPU 활용률이 가장 적은 모델인 Alex하나를 제외하는 대신 사용하는 전체 GPU의 개수를 하나 줄여서 전체 GPU 자원이 더욱 부족한 경우에도 공정한 자원의 배분이 이뤄지는지 실험하였다. 실험의 결과를 표현한 [그림 4-4 ~ 그림 4-6]에서 Sol. 은 DR Management 알고리즘을 적용한 경우를 나타내고, w/o Sol. 은 DR Management 알고리즘을 적용하지 않은 경우를 나타낸다.

2) 실험결과 및 분석

실험 결과는 다수의 응용이 동시에 실행된 후, 시간의 흐름에 따른 지표의 변화를 기준으로 분석한다. 관찰하는 지표는 총 3가지이고, [그림 4-4 ~ 그림 4-6]은 각 지표를 기준으로 결과 그래프를 표현한 것이다. 3가지 지표는 GPU자원을 공정하게 배분 하였는지를 판단하기 위한 $SD_{\max}^{diff}$ 변

화([그림 4-4]), 전체적인 속도저하 개선여부를 확인하기 위한 평균 SD값의 변화([그림 4-5]), 마지막으로 전체 GPU 활용률 평균값의 변화([그림 4-6]) 이다.

가) 시간에 따른 $SD_{\max}^{diff}$ 변화

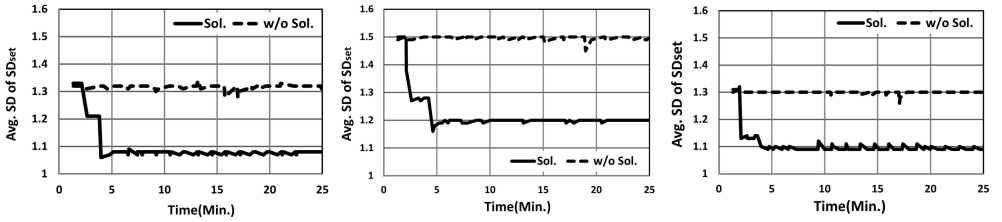


[그림 4-4] 학습시간에 따른 $SD_{\max}^{diff}$ 변화

[그림 4-4]의 (a),(b),(c) 모두 알고리즘을 적용하지 않았을 때 학습 초기(0~5Min.) 가 약 0.65의 $SD_{\max}^{diff}$ 값을 가지고 이 $SD_{\max}^{diff}$ 값이 학습이 진행되는 동안 비슷하게 유지된다. 제안된 알고리즘을 적용 시, 학습 초기의 $SD_{\max}^{diff}$ 는 미적용 시와 비슷하게 약 0.65의 값을 가진다. 하지만 학습이 진행되면서 실험군 (a)는 $SD_{\max}^{diff}$ 가 0.1 이하까지 감소하고 (b)는 약 0.3, (c)는 약 0.1 까지 감소하여 학습 종료 시까지 감소된 값이 유지되는 것을 확인할 수 있다. 이것으로 (a)~(c) 세 개의 실험군 모두 DR Management 알고리즘을 적용한 경우가 알고리즘을 적용하지 않은 경우 보다 더 공정하게 GPU 자원을 배분한다고 볼 수 있다. 실험군 (a)와 (b)를 비교하면 GPU자원이 상대적으로 더 적은 (b)의 경우에도 알고리즘을 적용하면 최대한 모든 응용이 공정하게 GPU 자원을 배분받는 것을 알 수 있다.

나) 시간에 따른 평균 SD값의 변화

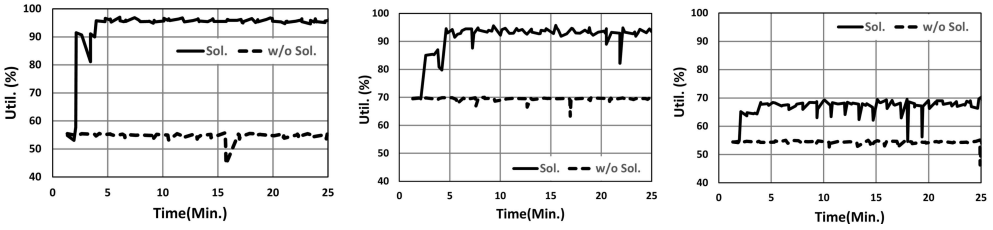
[그림 4-5]의 (a),(b),(c) 모든 경우에서 시작 시점에는 SD 평균값이 알고리즘을 적용했을 때와 적용하지 않았을 때의 값이 동일하고, 알고리즘 미적용



[그림 4-5] 학습시간에 따른 전체 학습 응용 SD 평균값의 변화

의 경우는 시작 시점에서의 값이 거의 변화 없이 종료 시까지 유지된다. 하지만 제안된 알고리즘을 적용하면 실험군 (a)는 약 1.1로 약 15% 감소, (b)는 약 1.2로 약 20% 감소, (c)는 약 1.1로 약 15% 감소하는 것을 확인할 수 있다. V-2-가 의 결과와 함께 분석해보면, 제안된 알고리즘을 적용 시 전체 DNN 학습 응용들의 학습이 진행되는 동안 SD값 차이가 줄어듦과 동시에 평균 SD값이 낮아짐을 알 수 있다. 이는 실행 중인 학습 응용들이 GPU를 공정하면서도 효율적으로 분배한다는 것을 의미한다.

다) 시간에 따른 전체 GPU의 평균 활용률 변화



[그림 4-6] 학습시간에 따른 GPU 활용률 평균값의 변화

마지막 실험으로, 제안된 알고리즘이 전체 GPU 활용률에 있어서도 더 효율적임을 밝히고자 한다. [그림 4-6]에 서 볼 수 있듯이 모든 실험군에서 알고리즘 미적용 때와 비교하면 제안된 알고리즘을 적용 시 GPU 활용률의 평균값이 학습이 진행되는 동안 더 높음을 확인할 수 있다. 학습이 진행되는 동안 실험군 (a)는 약 40%, (b)는 약 25% 높은 평균 GPU 활용률을 나타내었다.

(c)의 경우는 6개의 모델 중 ResNext, Shuffle, Mnas 모델이 포함되어 있다. 이들은 모델 자체의 연산특성이 Dense, Res와 같은 모델에 비하여 GPU 내부의 SM(Streaming Multiprocessor)(Nickolls, et al., 2010)들을 단위 시간당 집중적으로 사용하지는 않는다. 즉, 모델 자체적으로 GPU 활용률이 낮다. 따라서 (a), (b)의 경우보다 전체적인 GPU 활용률이 낮다. 하지만 이 경우에도 약 15% 더 향상된 평균 GPU 활용률을 나타내었다.

제 5 장 결론

본 논문에서는 단일 DNN 추론 과정에서 커널 분배를 통한 추론시간 단축과 다수의 DNN 학습 과정에서 전체 학습 응용의 공정한 GPU자원 배분 상황 유지 알고리즘을 제안한다. 단일 DNN 추론 시에는 트랜스포머 기반의 객체추적 모델인 CSWinTT의 트랜스포머 내부 MHA를 병렬실행 하여 효율성을 향상시켰다. head연산이 모두 독립된 입력값을 가지기 때문에 직렬실행을 사용하는 기존 방식보다는 병렬실행으로 변경하였을 때 추론시간을 단축시키는 이점을 가져올 수 있었다. 하나의 모듈로 이루어진 MHA모듈을 각 head연산이 하나의 모듈이 되도록 분리하고 GIL에 의해 기존 Python환경에서는 완전한 병렬수행이 불가능하여 C++환경에서 모델을 실행하였다. TorchScript를 이용하여 인코더의 각 레이어를 ScriptModule형태로 변환하였고 Libtorch 라이브러리와 함께 C++환경에서 컴파일 하였다. 각각의 head 연산이 커널을 실행할 스트림 또한 head연산 별로 다르게 지정하여 병렬성을 최대한 상승시켰다. 그 결과 기존 Python 환경에서 실행되던 CSWinTT에 비하여 인코더의 실행시간이 56.8% 감소하였고, FPS값은 63.3% 증가하였다.

다수의 DNN 학습 시에는 전체 응용이 학습 진행 시 필요로 하는 GPU 자원이 클러스터 내부 GPU자원보다 많은 경우 GPU자원에 대한 경쟁이 필연적으로 발생하고, 이 경쟁으로 인하여 공정한 GPU자원 분배가 이뤄지지 못하는 문제가 발생한다. 본 연구에서는 모든 학습응용에 대하여 최대한 공정한 GPU자원 분배가 학습 시 유지되는 것을 목표로 하였다. 어떤 학습 응용이 오프라인에서 한 개의 GPU를 독점하여 학습을 진행하였을 때 학습이 완료되는 시간을 기준으로 현재 상태에서의 예측되는 학습완료시간의 비율을 SD로 정의한다. SD값을 공정한 GPU 자원 배분의 지표로 사용하여, 학습이 진행중인 전체 응용의 SD값 간의 차이를 최대한 줄이는 방법을 통해 공정한 GPU 자원 배분이 이뤄지도록 하였다. 각 학습 응용이 GPU에 할당하는 미니 배치 데이터 비율을 주기적으로 조절하여 응용 간 SD값 차이를 줄일 수 있었다.

참 고 문 헌

1. 국외문헌

Deng, J., Dong, W., Socher, R., Li, L. J., Li, K., & Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In 2009 IEEE conference on computer vision and pattern recognition. 248-255.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., ... & Houlsby, N. (2020). An image is worth 16x16 words: Transformers for image recognition at scale. arXiv preprint arXiv:2010.11929.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition. 770-778.

Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). Densely connected convolutional networks. In Proceedings of the IEEE conference on computer vision and pattern recognition. 4700-4708.

Kristan, M., Leonardis, A., Matas, J., Felsberg, M., Pflugfelder, R., Kämäräinen, J. K., ... & Ma, Z. (2020). The eighth visual object tracking VOT2020 challenge results. In European Conference on Computer Vision. 547-601.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. Communications of the ACM, 60(6), 84-90.

Liu, Z., Lin, Y., Cao, Y., Hu, H., Wei, Y., Zhang, Z., ... & Guo, B.

(2021). Swin transformer: Hierarchical vision transformer using shifted windows. In Proceedings of the IEEE/CVF International Conference on Computer Vision. pp. 10012-10022.

Ma, N., Zhang, X., Zheng, H. T., & Sun, J. (2018). Shufflenet v2: Practical guidelines for efficient cnn architecture design. In Proceedings of the European conference on computer vision (ECCV) . 116-131.

Modules - PyTorch 1.13 documentation. Available online: https://pytorch.org/docs/stable/notes/modules.html

Muller, M., Bibi, A., Giancola, S., Alsubaihi, S., & Ghanem, B. (2018). Trackingnet: A large-scale dataset and benchmark for object tracking in the wild. In Proceedings of the European conference on computer vision (ECCV). 300-317.

MultiheadAttention - PyTorch 1.13 documentation. Available online: https://pytorch.org/docs/stable/generated/torch.nn.MultiheadAttention.html

Nadkarni, P. M., Ohno-Machado, L., & Chapman, W. W. (2011). Natural language processing: an introduction. Journal of the American Medical Informatics Association, 18(5), 544-551.

Nickolls, J., & Dally, W. J. (2010). The GPU computing era. IEEE micro, 30(2), 56-69.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... & Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems, 32.

Song, Z., Yu, J., Chen, Y. P. P., & Yang, W. (2022). Transformer Tracking with Cyclic Shifting Window Attention. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 8791-8800.

Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A., & Le, Q. V. (2019). Mnasnet: Platform-aware neural architecture search for mobile. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2820-2828.

Tensor CUDA Stream API - PyTorch master documentation. Available online:https://[ytprch.org/cppdocs/tensor_cuda_stream .html

TorchScript - PyTorch master documentation. Available online: https://pytorch.org/docs/master/jit.html

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. Advances in neural information processing systems, 30.

What Is the Python Global Interpreter Lock (GIL)? - Real Python. Available online: https://realpython.com/python-gil/

Xie, S., Girshick, R., Dollár, P., Tu, Z., & He, K. (2017). Aggregated residual transformations for deep neural networks. In Proceedings of the IEEE conference on computer vision and pattern recognition. 1492-1500.

Yan, B., Peng, H., Fu, J., Wang, D., & Lu, H. (2021). Learning spatio-temporal transformer for visual tracking. In Proceedings of the IEEE/CVF International Conference on Computer Vision. 10448-10457.

ABSTRACT

Improving the Efficiency of DNN Model Inference and Training through Distribution Mechanisms

Kim, In Mo

Major in IT Convergence Engineering

Dept. of IT Convergence Engineering

The Graduate School

Hansung University

With the advance of deep learning technology, DNN models are being used in various application fields. DNN models used in these fields require high accuracy and fast inference time. Furthermore, complex DNN models with high accuracy require a large number of GPU resources in the learning phase. The execution environment of DNN models is classified into two categories. One is a single DNN execution environment, and the other is multi DNN execution environment. In this paper, we propose methodologies for improving efficiency in two environments by applying distribution mechanisms. In the case of running a single DNN, we apply intra-model distribution mechanism to confirm the efficiency improvement through the reduction of inference time for the model inference phase. The other case, running multiple DNN models, we present an inter-model distribution algorithm that efficiently distributes GPU resources allocated during the training phase. This algorithm allows the entire learning application to receive a fair share of

GPU resources. In the single DNN model inference case, CSWinTT, an object tracking model, is the target DNN model. We propose a method to reduce inference time by changing heads in the Multi-Head Attention(MHA) of the transformer encoder serial to parallel execution. We propose a method to reduce inference time by changing the execution of the MHA heads inside the transformer encoder from sequential to parallel. For parallel execution, one MHA module is separated into several head modules, and each separated module is executed in a multi-threaded manner. In this case, it runs on C++, not the Python, to support complete multi-threaded execution. As a result of parallel execution, the average execution time of the encoder decreased by 56.8%, and the average FPS increased by 63.3%. In multiple DNN model training cases, competition between applications to occupy GPU resources is inevitable if the total amount of GPU resources is less than that of resources required by all learning applications. We propose a method for solving the problem of applications with relatively long learning times. Based on the current GPU resource utilization status of each application, the ratio of the predicted learning completion time and the learning completion time when each application uses GPU exclusively is calculated and defined as the current Slow-Down of the current application. The ratio of mini-batch data per GPU of each learning application is periodically adjusted and a fair allocation of GPU resources is achieved so that the Slow-Down values of currently executed learning applications become similar. As a result of confirming through the experiment, the maximum Slow-Down decreased by more than 53%, and the overall GPU utilization rate increased by 25%.

【KEYWORD】 Multi-head attention, Multi-threading, Multi-GPU training, Slow-Down