Journal of The Institute of Electronics and Information Engineers Vol.60, NO.4, April 2023 https://doi.org/10.5573/ieie.2023.60.4.39
ISSN 2287-5026(Print) / ISSN 2288-159X(Online)

논문 2023-60-4-5

멀티헤드 어텐션 병렬화를 통한 트랜스포머 기반 객체추적 모델의 실행속도 향상

(Improving the Execution Speed of Transformer-based Object Tracking Models through Multi-head Attention Parallelization)

김 인 모*, 김 명 선**

(Inmo Kim and Myungsun Kim©)

요 약

최근 딥러닝 기반 객체추적 기술이 발전함에 따라 스포츠 경기 분석, 영상 보안, 증강현실 등 다양한 응용 분야에서 객체추적 기술이 사용되고 있다. 사용자들은 높은 객체추적 정확도뿐만 아니라 빠른 객체추적 속도에 따른 높은 QoS를 요구한다. 본 연구에서는 현재 객체추적 솔루션 중 최고로 꼽히는 트랜스포머기반 CSWinTT 모델의 객체추적 속도를 향상시킨다. 이 모델의 인코더 레이어 내 Multi-Head Attention(MHA)의 head연산들은 전체 트랜스포머 추론 과정에서 가장 많은 실행시간을 차지하고, 각 head들은 각각 다른 입력 값을 가지지만 직렬로 실행된다. 이를 개선하기 위하여 본 연구에서는 각각의 head연산들을 병렬적으로 실행시킨다. 병렬 실행을 위하여 하나의 모듈로 이루어진 MHA연산을 head 개수만큼 서브 모듈로 분리하고, 분리된 각각의 모듈을 멀티쓰레드 환경에서 실행한다. 이때 순수 Python 환경에서는 불가능한 멀티쓰레드 환경을 C++ 실행 환경으로 개선하여 가능하게 한다. 또한 각 쓰레드들이 비동기적으로 전달하는 커널들을 GPU 내부에서 최대한 동시에 실행될 수 있도록 한다. 다양한 실험을 통해 MHA 병렬실행의 효과를 확인한 결과, 추론정확도는 거의 동일하게 유지하면서 기존 실행환경에 비하여 인코더의 평균 실행시간은 56.8% 감소하였고, 평균 FPS는 63.3% 증가하였다.

Abstract

With the recent advance of deep learning-based object tracking technology, it is being used in various application fields such as sports game analysis, video security, and augmented reality. Users require high object tracking accuracy as well as high QoS according to fast object tracking speed. In this study, we improve the object tracking speed of CSWinTT(transformer-based object-tracking model), which is currently considered as the best object tracking solution. The head operations of the Multi-Head Attention(MHA) in the encoder layer of this model occupy the most execution time in the entire inference procedure of the transformer. Each head has a different input value, but is executed in a serial manner. To overcome this, in this study, each head operation is executed in parallel. For parallel operation, the MHA consisting of one module is divided into sub-modules by the number of heads, and each separated sub-module is executed in a multi-threading environment. The pure Python environment does not guarantee a complete multi-threaded run. We thus improve to a C++ implementation environment to enable complete multi-threading. In addition, kernels transmitted asynchronously by each thread can be executed as concurrently as possible inside the GPU. As a result of checking the effect of MHA parallel execution through various experiments, the average execution time of the encoder decreased by 56.8% and the average FPS increased by 63.3% compared to the existing method while maintaining almost the same inference accuracy.

Keywords : Transformer, Multi-head attention, CSWinTT, Object tracking, Multi-threading

*학생회원, 한성대학교 IT융합공학과(Department of IT Convergence Engineering, Hansung University)
**정회원, 한성대학교 AI응용학과(Department of Applied Artificial Intelligence, Hansung University)
© Corresponding Author(E-mail : kmsjames@hansung.ac.kr)
※ 본 연구는 한성대학교 교내학술연구비 지원과제임.
Received ; December 5, 2022 Revised ; February 20, 2023 Accepted ; February 28, 2023

Ⅰ. 서 론

최근 딥러닝 모델의 발전으로 높은 정확도를 가지는 모델들이 많아짐에 따라 자율주행 자동차, 이미지 세그멘테이션(Image Segmentation) 등 다양한 분야에서 딥러닝 모델을 접목시켜 좋은 성능을 내고 있다. 객체추적(Visual Object Tracking)도 그러한 분야 중 하나이며, VIT[1]의 발표이후 본래 자연어처리[2] 분야에서 많이 쓰이던 트랜스포머(Transformer)[3]를 접목시켜 높은 성능을 보이는 객체추적 모델들이 발표되고 있다.

객체추적은 이전 프레임에서의 객체 위치 정보를 기반으로 현재 프레임에서의 객체의 위치를 도출하는 방식을 일컫는다. 프레임은 하나의 이미지를 의미하며, 하나의 데이터는 연속된 여러 프레임의 집합으로 이루어져 있다. 객체추적 추론 시 주어지는 유일한 정보는 첫 번째 프레임에서의 찾고자하는 객체의 위치 정보이다. 그리고 추론이 진행됨에 따라 이전 프레임에서의 객체의 위치 정보를 이용하여 현재 프레임에서의 객체의 위치를 찾아낸다.

이러한 객체추적 기술 중 2022년 CVPR (Computer Vision and Pattern Recognition) 학회에서 발표된 CSwinTT[4]는 트랜스포머를 기반한 객체 추적 모델이다. CSwinTT는 VOT2020[5] 데이터셋을 이용한 성능 측정에서 다른 9개의 객체 추적 모델과 비교하였을 때 가장 높은 예상 평균 중첩도(EAO, Expected Average Overlap) 값을 가져 최고 성능을 나타내고 있다. 또한 CSWinTT는 객체 추적의 성능을 높이기 위하여 CSWinTT 모델 내부 트랜스포머 인코더(Encoder)에 자연어처리(예를 들어 BERT[6])에서 사용한 멀티헤드 어텐션(Multi-Head Attention[7], 이하 MHA)이 아닌 Multi-Scale Window Partition과 Cyclic Shift를 적용한 독특한 방식의 MHA를 사용하였다. 이러한 방식은 Attention연산을 하는 최소단위가 픽셀의 집합으로 이루어진 패치이기 때문에 픽셀간의 Attention 연산에 비하여 추적하려는 객체를 더 적게 손상시킨다는 장점이 있다. 또한 Cyclic Shift를 통해 추적하려는 객체와 객체를 찾아야하는 후보 이미지 영역에 대한 더 많은 위치정보를 가지는 장점이 있다.

CSWinTT 모델과 같은 객체추적 기술에서 추론속도는 매우 중요한 요소이다. 하나의 프레임에 대한 추론 속도는 이미지가 로드되고부터 Bounding Box 결과 값이 도출될 때 까지 걸리는 시간이다. 객체추적의 특성상 임의의 i번째 프레임에 대한 추론이 진행되려면

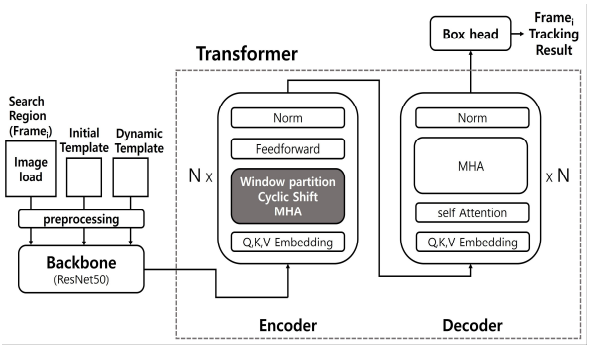

그림 1. 트랜스포머 기반 CSWinTT
Fig. 1. Transformer-based CSWinTT.

$i-1$번째 프레임에 대한 정보가 필요하다. $i-1$번째 프레임에 대한 추론속도가 빨라져 i번째 프레임에 필요한 정보전달이 빨라질수록 추론시작시각이 앞당겨지고, i번째 프레임까지의 추론시간이 감소한다. 이는 $i+1$번째 프레임부터 마지막 프레임까지 연쇄적으로 작용하여, 하나의 데이터에 대한 추론시간이 감소하는 결과로 이어진다.

본 논문에서는 객체추적을 수행하는 모델의 추론시간을 감소시켜서 초당 처리하는 프레임 수를 증가시키는 방법에 대하여 기술한다. 먼저 CSWinTT를 대상 모델로 하여 추론과정에서 소요되는 시간을 정량적으로 분석한다. 분석 결과를 토대로 추론 속도에 가장 많은 영향을 주는 기능 블록을 가려내고 해당 기능 블록을 실행함에 있어서 실행 시간을 단축시킬 수 있는 방안을 제안한다. 제안된 기법을 적용하여 실험한 결과 추론 시 1초당 처리하는 프레임의 수가 기존 대비 최대 63% 이상 증가함을 알 수 있었다.

Ⅱ. 연구배경 및 문제 정의

1. CSWinTT의 구성 및 동작 방식

그림 1은 CSWinTT 모델을 사용하여 i번째 프레임에서 객체 추적의 결과 값인 Bounding Box 좌표 값을 도출하는 전체 과정을 나타낸다. 입력 값은 총 3가지이며 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 좌상단의 x, y좌표 값과 x, y좌표를 기준으로 한 너비, 높이 값이 주어져 총 4개의 값으로

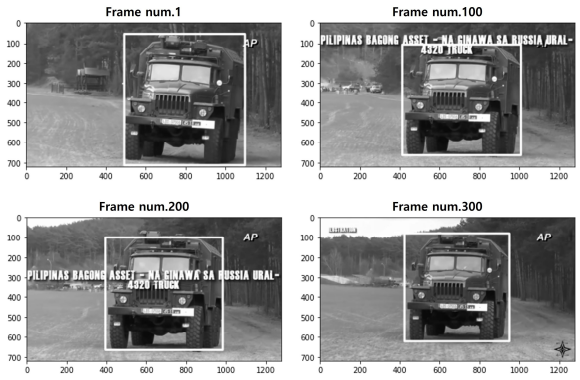

그림 2. CSWinTT를 사용한 객체 추적 과정
Fig. 2. Object tracking example using CSWinTT.

구성되어 있다. Dynamic Template는 최초에는 Initial Template와 동일하지만, 추론이 진행되면서 일정 프레임수마다 특정 조건을 만족하면 현재 프레임에서 찾은 Bounding Box좌표 값이 Dynamic Template가 된다. (CSWinTT에서는 Confidence Score가 0.5 이상인 조건을 Dynamic Template를 Bounding Box 좌표 값으로 교체하는 조건으로 여긴다.) Confidence Score는 하나의 프레임에 대한 추론의 결과인 Bounding Box안에 찾으려는 대상 객체 존재 유무에 대한 확률이다. 1의 값에 가까울수록 Bounding Box안에 객체가 있을 확률이 높음을 의미한다.

i번째 프레임에 대한 추론이 시작되면 프레임에 대한 전처리가 먼저 진행된다. Initial Template는 첫 프레임 추론 시에, Dynamic Template는 조건을 만족하여 bounding box 좌표 값으로 교체 시에 전처리가 진행된다. 전처리를 마친 3개의 입력 값은 Backbone으로 전달된다. Backbone은 ResNet50[8] 모델의 첫 번째 레이어 부터 세 번째 레이어까지로 구성된 모델이며 Backbone의 결과 값은 3개의 입력 값에 대한 특성 맵(Feature Map)이다. 특성 맵은 트랜스포머 모델로 전달되어 인코더의 첫 번째 레이어를 통과하게 된다. 트랜스포머의 인코더와 디코더는 총 N개의 레이어로 구성되어 있으며, CSWinTT에서는 $N=6$으로 설정하였다. 인코더의 i번째 레이어 출력 값은 $i+1$번째 레이어의 입력 값이 된다. 인코더의 마지막 6번째 레이어 출력 값은 디코더의 첫 번째 레이어 입력 값이 된다. 디코더도 마찬가지로 디코더의 i번째 레이어 출력 값이 디코더의 $i+1$번째 레이어 입력 값이 된다. 디코더의 마지막 레이어를 통과한 결과 값은 Box head[9]로 전달된다. Box head는 여러 개의 콘볼루션(Convolution) 레이어로 이루어져 있으며, 추론의 최종 결과인 Bounding Box 좌표 값을 도출한다.

표 1. 객체 추적에 소요되는 각 기능 블록의 실행 시간 비율
Table 1. Percentage of execution time for each functional block required for object tracking.

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

그림 2는 CSWinTT 모델을 사용하여 326개의 프레임으로 구성된 데이터에서 객체추적이 진행되는 과정을 보여준다. 이해를 돕기 위해 정수 값으로 이루어진 Bounding Box 좌표정보를 하얀색 직사각형으로 나타내었다. 첫 번째 프레임에 표시된 Bounding Box는 이후 프레임에서 객체추적을 위해 주어지는 정보이다. 100번째, 200번째, 300번째 프레임에 표시된 Bounding Box는 추론결과가 표시된 것이다. 해당 데이터의 Confidence Score는 전체 프레임 평균 0.9874로 그림에서도 알 수 있듯이 우수한 객체추적 성능을 보인다.

2. 문제 정의

표 1은 CSWinTT 모델에서 하나의 프레임에 대한 Bounding Box 결과 도출 시 진행되는 각각의 과정에 대한 시간비율을 분석한 결과이다. 트랜스포머 모델 추론 과정이 하나의 프레임에 대한 Bounding Box 좌표 값을 도출하는 전체 과정 중 차지하는 시간 비율이 65%로 가장 높고, 그 중에서도 인코더의 실행시간이 전체 처리시간의 58%로 대부분을 차지한다. 따라서 인코더 부분의 처리속도 향상은 전체 트랜스포머의 처리속도 향상의 기회를 제공한다고 볼 수 있다.

CSWinTT 내부 6개의 인코더 레이어들 각각은 MHA를 수행하고 MHA 연산 과정은 n개의 head 연산들을 수행한다. 이때 각 head 연산들의 입출력 값은 모두 독립적이다. 하지만 이 head 연산들은 데이터 의존성이 없음에도 직렬실행 된다. MHA 및 각 head 연산 과정은 다음 장에서 자세히 설명한다. 본 연구에서는 이 연산들을 최대한 병렬적으로 수행하도록 하여 인코더 실행 시간을 단축시켜 전체 추론시간을 단축시키고자 한다.

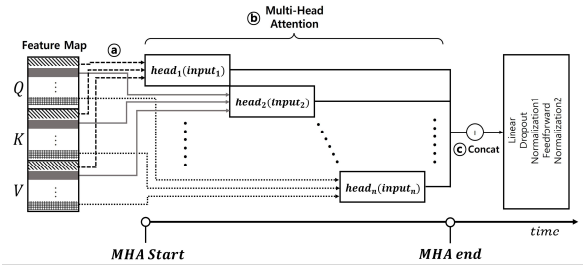

그림 3. 인코더 내 한 레이어 실행 과정
Fig. 3. One layer execution procedure within the encoder.

Ⅲ. 병렬 MHA 연산 기반 트랜스포머 인코더

1. MHA 연산 과정

그림 3은 인코더의 한 레이어가 실행되는 과정을 시간 축과 함께 나타낸 것이다. Q, K, V는 Backbone의 결과값인 특성 맵을 특정 차원을 기준으로 3분할 한 것이다. MHA의 임의의 head연산 $head_i(1 \le i \le n)$의 입력 값 $input_i$는 Q, K, V 각각을 head연산의 개수인 n개로 나눈 것 중 하나이며, 식 (1)은 이를 수식으로 나타낸 것이다.

$$\begin{aligned} Q &= Concat(Q_1, Q_2, \dots, Q_n) \\ K &= Concat(K_1, K_2, \dots, K_n) \\ V &= Concat(V_1, V_2, \dots, V_n) \\ input_i &= Q_i, K_i, V_i (1 \le i \le n) \end{aligned} \tag{1}$$

ⓐ는 $input_i$가 $head_i$연산의 입력 값이 되는 과정을 나타낸다. 식 (1)과 ⓐ에 의해 MHA의 각 head연산들은 각각 독립된 입력 값을 가지게 된다. ⓑ는 기존 실행 환경에서 MHA의 head연산들이 직렬실행으로 이루어져 있음을 나타낸다. ⓑ에서 $head_i$연산이 시작되는 시점은 $head_{i-1}$연산이 종료된 이후이다. 식 (2)는 i번째 head연산을 나타낸 수식이며 d_k는 Key의 차원이다.

$$\begin{aligned} & head_i(input_i) \\ &= Attention(Q_i, K_i, V_i) \\ &= softmax(\frac{Q_i K_i^T}{\sqrt{d_k}}) V_i \end{aligned} \tag{2}$$

ⓒ는 마지막 n번째 head연산이 종료되면 모든 head연산의 결과 값이 하나로 텐서로 합쳐지는 것을 나타낸다. 이후 합쳐진 텐서는 레이어의 남은 과정(Linear~Normalization2)을 순차적으로 수행한다. Ⅱ장에서 언급했듯이, 레이어 실행과정에서 MHA의 모든 head연산은 독립적인 입출력 값을 가지기 때문에 연산간의 의존성이 없다. 따라서 각 head연산들은 병렬실행을 위한 전제조건을 만족한다.

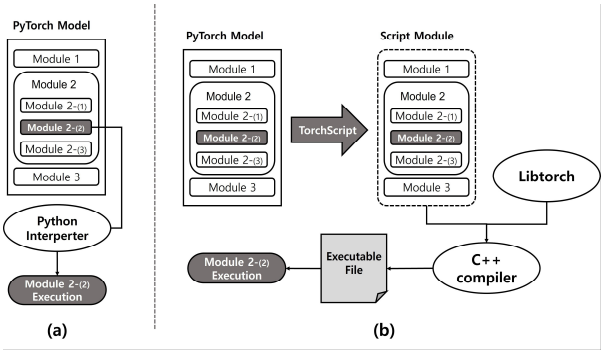

그림 4. PyTorch 모델의 실행 환경: (a) 기존 실행 환경; (b) 본 논문에서 제안하는 실행환경
Fig. 4. Execution environment of PyTorch models: (a) Original execution environment; (b) proposed execution environment in this paper.

2. 모델 변경

CSWinTT의 트랜스포머 인코더를 구성하는 6개 레이어는 각각 PyTorch[10] 모델 관점에서 보면 하나의 모듈이다[11]. 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. 더불어 우리는 완전한 멀티쓰레딩의 구현을 위하여 기존환경의 순수 Python으로 구성된 PyTorch 모델을 C++기반 모델로 우선 변경한다. 이를 위하여 트랜스포머 인코더의 각 레이어를 TorchScript[12]를 사용하여 ScriptModule형태로 변환하였다.

그림 4는 여러 모듈의 집합인 PyTorch 모델이 실행되는 과정을 나타낸다. 모듈 2의 경우 하나의 모듈 또한 여러 모듈의 집합인 경우를 예시한다. (a)는 기존 환경에서의 모델 실행 과정이고, (b)는 멀티쓰레딩 구현을 위해 본 논문에서 제안하는 환경에서의 모델 실행 과정이다. 기존 실행 환경에서는 컴파일 과정 없이 Python 인터프리터가 각각의 모듈을 순차적으로 실행시킨다. 멀티쓰레딩 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.

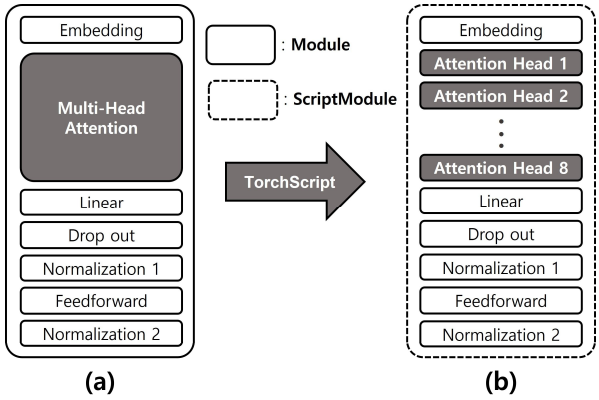

그림 5. TorchScript를 통한 인코더 레이어 변경
Fig. 5. Modification of the encoder layer with TorchScript.

그림 5의 (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 레이어속 MHA과정은 하나의 모듈로 이루어져 있어 하나의 쓰레드 안에서 실행되었다. 복수개의 쓰레드를 활용하여 MHA속 각각의 head 연산들을 병렬실행하기 위하여 하나의 모듈로 이뤄져있던 MHA과정을 하나의 head연산을 기준으로 여러 개의 모듈로 분리한 ScriptModule을 생성했다. 또한 C++ 환경에서 전체 CSWinTT 모델을 구동시키기 위해 Backbone, 트랜스포머 디코더, Box head 모듈 또한 ScriptModule 형태로 변환했다.

3. 병렬 MHA 연산 적용

본 절에서는 멀티쓰레딩을 활용한 MHA 병렬실행 시 기존 실행환경이 아닌 C++ 기반의 실행환경을 선택한 이유에 대하여 설명한다. 그림 6, 7은 3개의 코어로 구성된 CPU에서 4개의 쓰레드로 구성된 하나의 프로세스가 동작하는 예시이다. 각각의 쓰레드가 실행되면서 GPU에 모듈실행을 담고 있는 커널을 전달한다.

그림 6은 Python 환경에서 별도의 개별적인 스트림 기술[13]을 적용하지 않고 디폴트 스트림 하나를 사용하여 각각의 쓰레드들이 커널을 전달하는 과정을 보여준다. Python 환경에서는 GIL(Global Interpreter Lock) 정책에 의하여 특정 시점에서 하나의 쓰레드만이 Python 인터프리터를 실행시킬 수 있다[14]. 하나의 쓰레드가 GIL을 획득하면 다른 쓰레드들은 해당 쓰레드

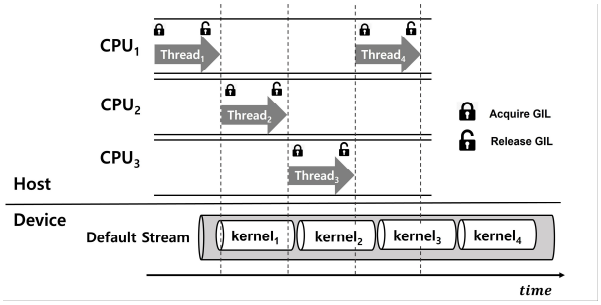

그림 6. GIL(Global Interpreter Lock) 정책 하의 멀티쓰레드들의 커널 전달 과정
Fig. 6. Transferring kernels through multi-threading under GIL(Global Interpreter Lock).

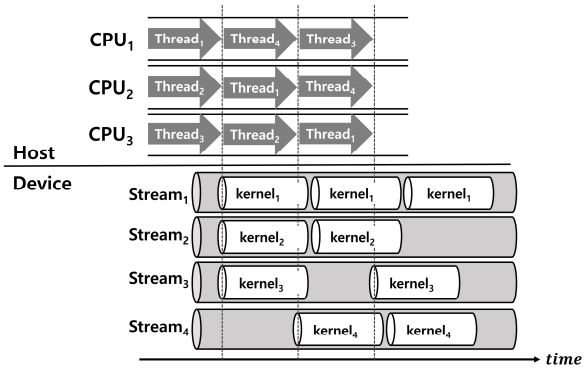

그림 7. GIL에 구애받지 않고 멀티 스트림 활용이 가능한 멀티쓰레드들의 커널 전달 과정
Fig. 7. Multi-threaded kernel delivery that enables multi-stream utilization regardless of GIL.

가 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 6에서 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 7은 이러한 문제를 해결한 예시를 보여준다. C++ 실행환경을 사용함으로써 멀티쓰레딩이 가능하며 각각의 쓰레드들이 병렬적으로 수행됨을 알 수 있다. C++ 환경에서는 Python 환경에서 발생하는 GIL 제약사항의 영향을 받지 않기 때문에 3개의 CPU에서 각각 동시에 쓰레드가 실행되는 것이 가능하다. 또한 커널 전달 역시 개별적인 스트림을 활용하여 GPU에 전달되

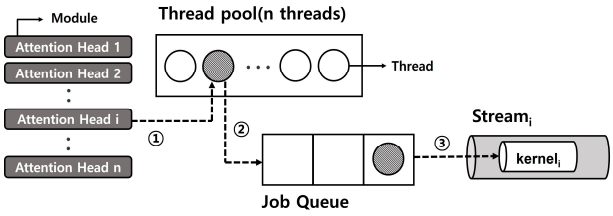

그림 8. 병렬실행을 위해 추가된 기능블록들의 실행 과정
Fig. 8. Execution procedure of functional blocks added for parallel execution.

는 과정을 보여준다. 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 8은 MHA head연산들을 병렬실행하기 위하여 추가한 병렬실행 기능블록의 실행 과정을 나타낸다. 그림 4에서 분리한 각 Attention Head 모듈 n개 중 i번째($1 \leq i \leq n$) 모듈의 실행과정이다. $input_i$가 $head_i$에 전달되고 나면 ① 쓰레드 풀에 생성된 쓰레드중 하나에서 $head_i$연산이 실행된다. ② $head_i$연산이 실행되는 쓰레드는 Job Queue로 전달된다. Job Queue에는 실행될 연산이 지정된 쓰레드가 순서대로 들어간다. ③ Job Queue를 빠져나오면 연산은 $Stream_i$에 커널 형태로 할당되어 GPU에서 실행된다. $head_i$연산이 커널 형태로 GPU에서 실행되기 시작한 직후 $head_{i+1}$연산이 ① 과정을 시작한다. 위와 같은 방식으로 MHA연산이 진행되면 그림 5와 다르게 $head_{i+1}$은 $head_i$가 연산중인 것과 관계 없이 $input_{i+1}$을 전달받는다.

그림 9는 쓰레드별 스트림을 지정한 멀티쓰레드 환경에서 MHA을 병렬실행 했을 때 시간에 따른 인코더의 한 레이어 실행과정을 나타낸다. 특성 맵에서 각각의 head연산에 입력 값을 할당하는 과정은 그림 4와 동일한 과정을 따른다. 각각의 head연산 시작지점이 조금씩 차이가 나는 이유는 head연산의 입력 값이 순차적으로 전달되어, 입력 값의 전달시점이 각 head마다 다르기 때문이다. $head_i$는 $Stream_i$에 커널을 보내고 $head_n$이 종료되면 각 head연산의 결과 값을 하나로 합치고 그림 5와 동일하게 레이어의 남은 과정을 순차적으로 수행한다.

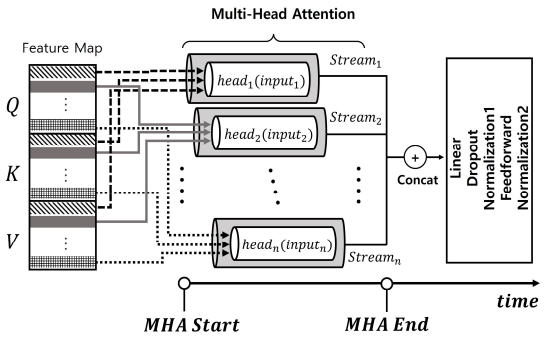

그림 9. MHA 병렬연산 적용 후 인코더 내 한 레이어 실행 과정
Fig. 9. Execution example of one layer in the encoder with MHA parallelization.

Ⅳ. 실 험

본 장에서는 기존 실행 환경에서의 CSWinTT 모델과 Ⅲ장에서 언급한 기법에 따라 멀티쓰레드를 사용하여 MHA의 각 head연산을 병렬실행으로 변경한 CSWinTT 모델을 비교하는 실험 내용을 다룬다. 실험은 성능차이 비교 실험과 추론시간 비교 실험으로 나누어 수행한다.

1. 실험환경

실험에 사용된 시스템은 표 2의 CPU 32개와 GPU 4개로 구성된 리눅스 운영체제 기반 서버이다. 실험 시에는 1개의 GPU만을 사용하였다.

표 2. 타겟 시스템 구성
Table 2. Target system.

GPU (NVIDIA RTX A6000)	336 Tensor Cores 10,752 CUDA Cores 48GB Memory 309.7 TFLOPS (FP32)
CPU (AMD Ryzen Threadripper PRO 3955WX)	16-core 64MB L3 Cache
OS	Ubuntu 20.04
CUDA ver.	CUDA 11.6

실험은 총 3가지 경우에 대한 비교를 다룬다. Case-A는 기존실행 환경에서 CSWinTT 모델을 추론하였을 때의 결과이다. Case-B는 TorchScript를 사용하여 Backbone, 트랜스포머 인코더와 디코더, Box head 블록을 ScriptModule로 변환한 것을 이용하여 멀티쓰레딩을 위한 C++ 환경에서 CSWinTT 모델을 추

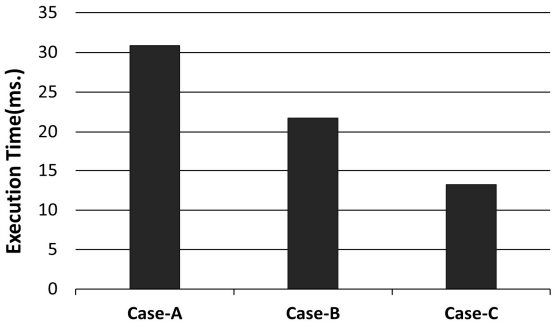

그림 10. 인코더 평균 실행시간
Fig. 10. Average running time of the encoder.

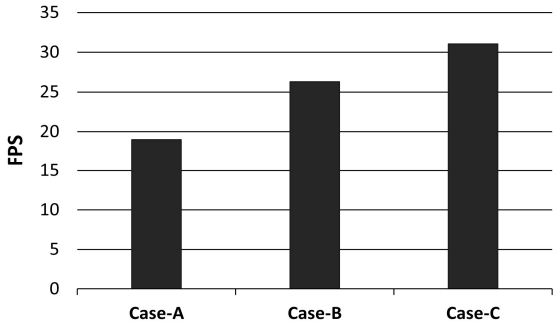

그림 11. 평균 FPS 비교
Fig. 11. Comparison of average FPS.

론하였을 때의 결과이다. Case-C는 Case-B에 추가적으로 그림 3에 나타난 MHA 분리작업을 거친 ScriptModule을 사용하여 멀티쓰레드로 MHA의 각 head연산을 병렬실행하고 각각의 쓰레드가 사용할 스트림을 할당하였을 때의 결과이다.

추론 시에는 TrackingNet[15] 데이터셋을 사용하였다. TrackingNet 데이터셋은 총 511개의 데이터로 구성되어 있고, 각각의 데이터는 순서가 정해져있는 프레임 이미지들과 첫 번째 프레임에서의 추적대상 객체의 위치정보(좌상단의 x, y좌표 값과 x, y좌표 값을 기준으로 한 너비, 높이 값)로 구성되어 있다. 프레임 개수는 데이터마다 다르며 최소 96개～최대 2368개, 평균 441.5개이다.

2. 추론시간 비교 실험

추론시간 비교 실험은 하나의 프레임에 대한 Bounding Box 도출 시 트랜스포머 인코더의 평균 실행시간 비교(그림 10), 데이터 셋의 전체 데이터에 대한 평균 FPS값 비교(그림 11), 마지막으로 1프레임을 추론하는 전체 실행시간에서 이미지 로드와 전처리 과정과 트랜스포머 인코더 과정이 차지하는 실행시간 비율에 대한 비교를 다룬다(그림 12).

그림 10의 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++ 환경이지만 트랜스포머 인코더 MHA을 모듈단위로 분리하고, 멀티쓰레딩을 적용한 병렬실행과 커널이 실행될 스트림을 지정한 Case-C의 실행시간이 13.23ms로 Case-B에 비하여 39% 감소하였다. 이를 통해 MHA연산을 head 연산 단위로 분리하여 병렬처리 한 것이 효과가 있음을

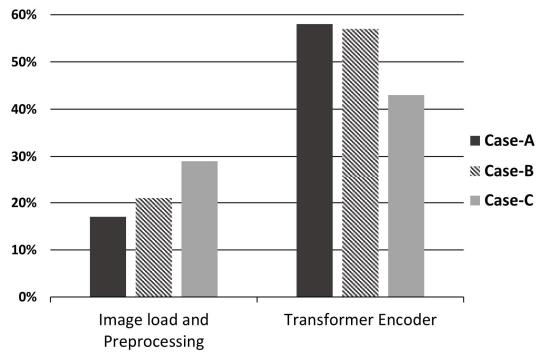

그림 12. 이미지 전처리와 인코더 실행시간 비교
Fig. 12. Image preprocessing and encoder execution time comparison.

알 수 있다. 또한 Case-A 대비 트랜스포머 인코더의 한 레이어 실행시간이 56.8% 감소하여 기존 환경과 비교하여 레이어 실행시간이 절반이상 줄어드는 효과가 있었다.

그림 11은 FPS값을 지표로 하여 각 Case에 대한 모델 전체의 추론속도 비교를 다룬다. FPS값은 추론 시 1초당 처리하는 프레임의 개수를 의미하며 FPS값이 높을수록 전체 추론속도가 빠른 것을 의미한다. 그림 11의 Case-A와 Case-B를 비교하였을 때, Case-A의 FPS값은 19, Case-B의 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-A에 대비하여 FPS값이 63.3% 증가하여 기존 환경에서 실행되던 것에 비해 ScriptModule의 최적화 효과와 MHA 각 head연산을 병렬 처리한 효과가 있음을 확인하였다.

그림 12의 Case-A의 경우 표 1과 같은 값을 가진다. 그래프에서 트랜스포머 인코더가 전체 실행시간에서 차지하는 비율은 Case-A와 Case-B는 각각 58%,

57%로 비슷한 비율을 가진다. 하지만 Case-C에서는 43%로 Case-A와 Case-B에 비하여 낮아졌다. 이는 트랜스포머 인코더의 실행시간이 줄어들어 전체 실행시간 중 인코더가 차지하는 비율이 줄어들었기 때문이다. 반대로 이미지 로드와 전처리가 전체 추론과정 중 차지하는 비율은 Case-C에서 29%로 가장 높게 나타났다. 이미지 로드와 전처리 과정의 실제 수행시간은 3가지 Case 모두 비슷하기 때문에 전체 추론시간이 줄어든 Case-B와 Case-C가 Case-A에 비하여 높은 비율 값을 가지게 된다.

3. 추론 정확도 비교 실험

추론 정확도 비교는 전체 데이터 셋을 Case-A, Case-C에서 추론하였을 때를 비교한다. 두 Case의 오차를 측정하기 위한 지표로 AUC(Area under the curve), P(Precision), P_{norm}(Normalized precision)을 사용하였다[16].

표 3은 위의 3가지 지표에 대한 추론 정확도 결과이다. 기존 실행환경(Case-A)과 논문에서 제시한 실행환경(Case-C)을 비교할 때 AUC, P, P_{norm}의 지표에서 각각 1.583, 1.444, 1.687의 아주 작은 차이를 나타내었다. 이를 통하여 본 논문에서 제안한 기법은 기존 대비 추론 정확도 측면에서 거의 동일함을 알 수 있다.

표 3. 추론 정확도 비교
Table 3. Comparison of inference accuracy.

	AUC	P	P_{norm}
Case-A	90.041	88.636	91.469
Case-C	88.458	87.192	89.782

V. 결 론

본 논문에서는 객체추적 모델인 CSWinTT를 타겟모델로 하여 실행속도를 향상시키는 방법에 대하여 제안한다. CSWinTT의 MHA 각 head연산은 모두 독립된 입력 값을 가지기 때문에 직렬로 실행하는 기존 방식보다는 병렬로 실행했을 때 실행속도 향상 이점을 가져올 수 있었다. MHA를 병렬처리하기 위하여 하나의 모듈로 이루어진 MHA를 각 head 연산이 하나의 모듈이 되도록 8개의 모듈로 분리하였다. 기존 실행환경에서는 GIL 정책에 의해 완전한 병렬수행이 불가능하여 C++ 환경에서 모델을 실행시키고자 하였다. 이를 위해 TorchScript를 이용하여 각 인코더의 각 레이어를 ScriptModule형태로 변환하였고 Libotrch 라이브러리와 함께 C++ 환경에서 컴파일 하였다. 또한 각각의 head연산이 커널을 실행할 스트림을 head연산 마다 다르게 지정하여 병렬성을 최대한 상승시켰다. 그 결과 추론 정확도는 거의 동일하게 유지하면서 기존 실행 환경에서 실행되던 CSWinTT에 비하여 추론 시 인코더의 실행시간이 크게 감소하였고, 1프레임을 추론하는데 걸리는 시간이 감소하여 FPS값이 19에서 31.04로 증가하였다.

REFERENCES

[1] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit & N. Houlsby, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale" arXiv:2010.11929, 2010.

[2] P. M. Nadkarni, L. O. Machado & W. W. Chapman, "Natural Language Processing: an Introduction" Journal of the American Medical Informatics Association, Vol. 18, no. 5, pp. 544-551, September 2011.

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser & I. Polosukhin, "Attention is All You Need" Advances in Neural Information Processing Systems, Vol. 30, December 2017.

[4] Z. Song, J. Yu, Y. P. P. Chen & W. Yang, "Transformer Tracking With Cyclic Shifiting Window Attention" Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition(CVPR), pp. 8791-8800, Vancouver, Canada, November 2022.

[5] M. Kristan, A. Leonardis, J. Matas, M. Felsberg, R. Pflugfelder, J. K. Kamarainen, M. Danelljan, L. C. Zajc, A. Lukezic, O. Drbohlav, et al., "The eighth visual object tracking VOT2020 challenge results" European Conference on Computer Vision, pp. 547-601, Glasgow, United Kingdom, August 2020.

[6] J. Devlin, M. W. Chang, K. Lee & K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding" arXiv: 1810.04805, 2018.

[7] https://pytorch.org/docs/stable/generated/torch.nn.MultiheadAttention.html

[8] K. He, X. Zhang, S. Ren & J. Sun, "Deep

Residual Learning for Image Recognition" Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770-778, Las Vegas, Nevada, USA, June 2016.
[9] B. Yan, H. Peng, J. Fu, D. Wang & H. Lu, "Learning Spatio-Temporal Transformer for Visual Tracking" Proceedings of the IEEE/CVF International Conference on Computer Vision, pp. 10448-10457, Montreal, QC, Canada, October 2021.
[10] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in Proc. of Advances in Neural Information Processing Systems (NeurIPS), pp. 8024-8035, Vancouver, Canada, July 2019
[11] https://pytorch.org/docs/stable/modules.html
[12] https://pytorch.org/docs/master/jit.html
[13] https://pytorch.org/cppdocs/notes/tensor_cuda_stream.html
[14] https://realpython.com/python-gil/
[15] M. Muller, A. Bibi, S. Giancola, S. Alsubaihi, B. Ghanem, "TrackingNet: A Large-Scale Dataset and Benchmark for Object Tracing in the Wild" Proceedings of the European Conference on Computer Vision, pp. 300-317, Munich, Germany, September 2018.
[16] Y. Wu, J. Lim, M. Yang, "Online Object Tracking: A BenchMark" Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition(CVPR), pp. 2411-2418, Portland, USA, June 2013.

저 자 소 개

김 인 모(학생회원)
2021년 한성대학교 IT응용시스템공학과 학사 졸업.
2023년 한성대학교 IT융합공학과 석사 졸업.

<주관심분야 : GPU Programming, DNN framework, Deep Learning, Object Tracking>

김 명 선(정회원)
2000년~2002년 LG전자 액세스 네트워크 연구소 주임연구원
2002년~2011년 삼성전자 DMC 연구소 책임연구원
2016년 서울대학교 전기컴퓨터 공학부 박사 졸업.
2016년~2019년 삼성전자 SR연구소 수석연구원
2019~현재 한성대학교 AI응용학과 조교수
<주관심분야 : AI Computing System, Linux kernel, HW/SW Co-design, NPU 등>