학사학위 청구논문

텐서플로우 라이트에서의 위임 정책 최적화

Delegation Optimizing in TensorFlow Lite

2023년 12월 15일

숭실대학교 IT대학

AI융합학부

20180391 이지훈

학사학위 청구논문

텐서플로우 라이트에서의 위임 정책 최적화

Delegation Optimizing in TensorFlow Lite

지도교수 : 이 길 호

이 논문을 학사학위 논문으로 제출함

2023년 12월 15일

숭실대학교 IT대학

AI융합학부

20180391 이지훈

이지훈의 학사학위 논문을 인준함

심사위원장 김 성 흠 (인)

심 사 위 원 이 길 호 (인)

2023년 12월 15일

숭실대학교 IT대학

감사의 글

논문이 완성되도록 도와주신 모든 분께 감사드립니다.

목 차

표 및 그림 목차 …… 1
국문초록 ……2

I. 서론 …… 1
II. TensorFlow Lite 기초 이론 …… 2
Ⅱ-1. TensorFlow Lite 내부 구조 이론 …… 2
Ⅱ-2. TensorFlow Lite 위임 이론 …… 3
Ⅲ. 새로운 위임 정책 제안 …… 4
Ⅲ-1. 기존 위임 정책의 문제점 분석…… 4
Ⅲ-2. 새로운 위임 정책의 설계 및 구현 …… 5
Ⅳ. 실험 및 평가 …… 7
Ⅳ-1. 실험 환경 …… 7
Ⅳ-2. 실험 결과 …… 8
Ⅳ-3. 실험 결과에 대한 분석 및 평가 …… 10
Ⅳ-4. 향후 연구 계획 …… 13
V. 결론 …… 15
참고문헌 …… 15
부록 …… 16

표 및 그림 목차

그림1 TensorFlow Lite 구조 …… 2
그림2 현 TensorFlow Lite의 위임 정책 …… 3
그림3 제안하는 위임 정책의 접근 방식 …… 5
그림4 프로파일링 프로세스 구현도 …… 5
그림5 프로파일링 프로세스 실험 결과 …… 8
그림6 각 위임 가능 조각별 경향성 분석 그래프 …… 11
그림7 실제 연산량과 추론 성능과의 경향성 분석 그래프 …… 12
그림8 현 위임 정책과 제안하는 위임 정책의 추론 성능 비교 그래프 ……12

국문초록

텐서플로우 라이트에서의 위임 정책 최적화

AI 융합학부 이지훈
지도교수 이길호

최근 Deep Nural Network(이하 DNN) 기술이 발전함에 따라, 모바일 및 임베디드 시스템 환경에서 DNN 어플리케이션의 실행을 위한 프레임워크의 중요성이 대두되고 있다. TensorFlow Lite (이하 TFLite)는 모바일 및 임베디드 시스템 환경에서 가장 많이 사용되는 머신러닝 프레임워크 중 하나이다. TFLite는 DNN 추론 과정에서 가속이 필요할 때, 가용 가속 연산장치(예: GPU,NPU,DSP)를 사용하도록 지원한다. 이러한 과정을 delegation(이하 위임이라 한다. 하지만, 현 TFLite의 위임 정책은 사용자의 편의성 관점과 추론 성능 관점에서 적절하지 못하다. 이에 본 논문에서는 보다 사용자 친화적이고 추론 성능이 뛰어난 위임 정책을 제안한다. 본 논문에서 제안한 위임 정책을 활용하면, 현 TFLite의 위임 정책과는 달리 여러 번 컴파일 및 테스트를 하지 않고도 단 한 번에 추론 성능이 가장 뛰어난 위임 조합을 찾을 수 있다. 또한 그러한 위임 조합은 현 TFLite의 위임 정책에서 도출 가능한 최적의 위임 조합의 추로 성능과 같거나 더 뛰어남을 보장한다. 본 논문에서 주장하는 위임 정책을 활용한다면 현 TFLite의 기본 위임 정책보다 최대 약 1000%의 DNN 가속 효과를 얻을 수 있다.

I. 서 론

최근 edge 컴퓨팅에 대한 필요성이 대두되며 모바일 및 임베디드 시스템의 급격한 발전이 이루어지고 있다. edge 컴퓨팅은 데이터를 원격 클라우드로 전송하지 않고 장치 자체에서 처리하기 때문에 지연시간이 크게 감소한다. 또한 cloud 컴퓨팅과 달리 네트워크의 개입을 최소화하기 때문에 대역폭의 효율이 좋은 편이다. 따라서 대역폭이 제한되거나 실시간성이 중요한 모바일 및 임베디드 시스템 환경 (예: 스마트폰, 자율주행 차량)에서 edge 컴퓨팅이 많이 활용되며 발전하고 있다.

또한 최근 Deep Nural Network(DNN) 기술이 발전함에 따라 모바일 및 임베디드 시스템 환경에서의 DNN 어플리케이션 실행을 위한 프레임워크의 중요성이 대두되고 있다.

TensorFlow Lite (이하 TFLite)는 모바일 및 임베디드 시스템 환경에서 가장 많이 사용되는 머신러닝 프레임워크 중 하나이다. TFLite는 흔하게 사용되고 있는 TensorFlow의 edge version이라 할 수 있다. DNN 모델을 학습시키는 기능은 지원하지 않지만, 만들어진 DNN 모델을 추론하는 기능을 지원한다.

TFLite는 이러한 DNN 추론 과정에서 가속이 필요할 때, 가용 가속 연산장치(예: GPU,NPU,DSP)를 사용하도록 지원한다. 하지만 TFLite의 가속 정책은 현재 이러한 가속 연산장치를 그리 효율적으로 쓰고 있지 않다. 이에 본 논문에서는 TFLite에 대해 먼저 간략히 소개한 뒤, 기존 TFLite의 가속 정책의 문제점을 제기하고, 추론 성능 관점에서 더욱 효율적인 가속 정책을 제안한다. 그 후 제안한 가속 정책에 대한 테스트 결과를 시각화하며 분석하고, 마지막으로 본 논문에 대한 결론과 향후 연구에 대해 간략히 다루며 마친다.

II. TensorFlow Lite 기초 이론

II-1. TensorFlow Lite 내부 구조 이론

TensorFlow Lite(이하 TFLite)는 모바일 및 임베디드 시스템에서 DNN 모델 추론을 위하여 TensorFlow를 경량화한 프레임워크이다. Android / iOS 등의 모바일 환경, 내장형 Linux 및 마이크로 컨트롤러 등과 같은 임베디드 환경 모두에서 구동이 가능하다. 또한 Java, C++, Python, Swift, Objective-C와 같은 다양한 언어를 지원하여 접근성이 쉬운 편이다.

그림 1에서 보듯, TFLite의 구조는 크게 Converter(이하 컨버터)와 Interpreter(이하 인터프리터)로 구성된다.

먼저 컨버터는, TensorFlow에서 학습된 DNN 모델을 받아와 TFLite 형식으로 변환하는 역할을 한다. TFLite에서는 DNN 모델 학습의 기능을 지원하지 않는다. 따라서 SavedModel 형태의 학습된 DNN 모델에 경량화 메커니즘을 적용하여 TFLite 형태의 DNN 모델을 생성한다. 대표적인 경량화 메커니즘으로는 quantization(이하 양자화)이 있다. 양자화란 모델의 파라미터를 더 적은 비트로 표현함으로써 모델의 크기를 줄이는 기법이다. 주로 64비트 부동 소수점 형태의 가중치를 16비트 부동 소수점 혹은 8비트 정수 형태로 변환하는 방식으로 사용된다.

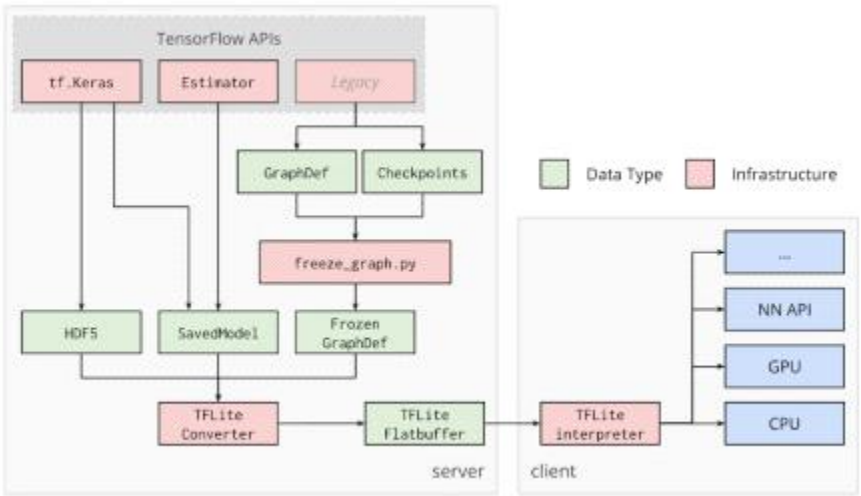

그림 1. TensorFlow Lite 구조

다음 인터프리터는, 변환된 TFLite 모델을 로드하고 실행하는 역할을 한다. 변환된 TFLite 모델은 Flatbuffer라는 특수한 형태의 포맷으로 저장된다. Flatbuffer는 구글에서 만든, 메모리 효율성과 속도가 뛰어난 특수한 형태의 데이터 구조이다. 이러한 형태의 모델을 로드하여, CPU / GPU / TPU 등의 연산장치에서 실행되도록 처리해 주는 것이 인터프리터의 역할이다. 또한 TFLite는 기본적으로 모든 연산을 CPU에서 실행되도록 설정하기 때문에, GPU / TPU와 같은 가속 연산장치에서 실행이 되기 위해서는 별도의 연산 변환 작업이 필요하다. 이를 TFLite에서는 delegation(이하 위임)이라 한다.

II-2. TensorFlow Lite 위임 이론

위임을 하기 위해서는, 우선 TFLite 형태의 모델을 구성하는 모든 layer(이하 계층)을 대상으로 가속 연산장치와 호환이 되는지를 검사해야 한다. 주로 사용되는 계층(예 : 합성곱 , 완전연결 계층)의 위임은 지원하지만, 일부 계층 (예: split , squeeze 계층)은 아직 위임이 지원되지 않는다. 위 특징을 가지는 계층을 Fallback 계층이라고 한다.

TFLite의 위임 정책은 입력 모델의 중간에 이러한 Fallback 계층이 존재할 경우, 그림 2와 같은 방식을 사용한다.

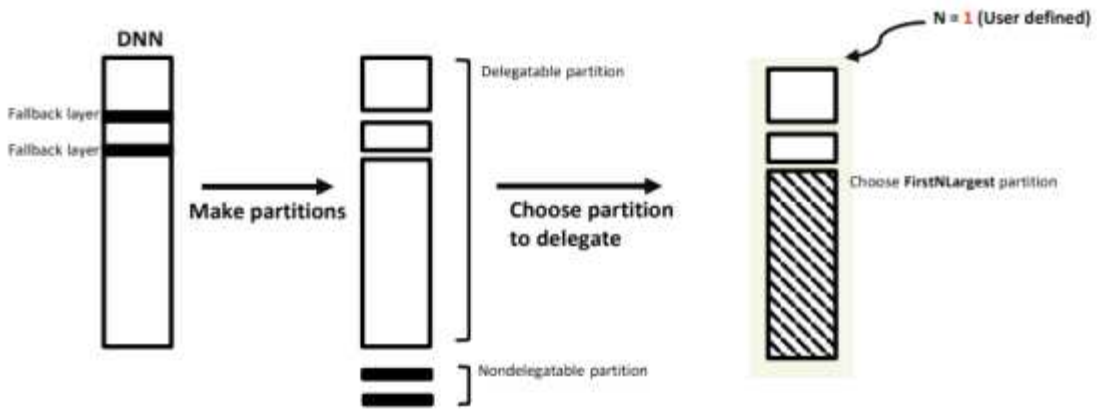

그림 2. 현 TensorFlow Lite의 위임 정책

첫 번째로 “Make partitions” 과정에서는, 입력 DNN을 위임 가능한 partition(이하 조각)과 위임 불가능한 조각으로 분할한다. 이때, 최대한 연속된 계층끼리 묶어 각각의 조각을 만든다. 그림 2와 같이, Fallback 계층이 아닌 연속된 계층들은 하나의 위

임 가능한 조각이 되며, Fallback 계층은 하나의 위임 불가능한 조각이 된다.
두 번째로 “Choose partition to delegate” 과정에서는, 앞서 만들어진 위임 가능한 조각들을 대상으로 몇 개를 선택하여 최종 위임할 것인지를 결정한다. 이때, 현 TFLite는 “FirstNLargest”의 위임 정책을 사용한다.
여기서 “N”은 사용자가 직접 설정하는 변수로, 몇 개의 조각을 선택할 것인지 결정한다. 또한 “First Largest”는 덩치가 큰 조각, 즉 포함하고 있는 la계층의 개수가 많은 조각을 우선적으로 고르는 정책을 의미한다. 정리하면, “FirstNLargest” 위임 정책이란 포함하고 있는 계층의 개수가 많은 조각을 우선적으로 N개(여기서 N은 사용자가 직접 정의) 고르는 것을 의미한다.

III. 새로운 위임 정책 제안

III-1. 기존 위임 정책의 문제점 분석

이러한 FirstNLargest 위임 정책은, 크게 2가지의 문제가 있다.
첫 번째로, 포함하고 있는 계층의 개수가 많은 조각을 우선적으로 고르게 하는 정책은 추론 성능 관점에서 효율적이지 못하다. 물론 대부분 포함하고 있는 계층의 개수가 많은 조각이 연산량이 많을 것이므로, 그러한 조각을 우선적으로 위임하면 가속의 효과가 더욱 뛰어날 것이다. 하지만 GPU에서의 DNN 워크로드 분석 및 스케줄링 연구인 S^3 DNN(Zhou et, al)[1]을 통해, DNN의 특정 계층들은 다른 계층들에 비해 월등히 높은 연산량을 보일 수 있음을 알 수 있다. 따라서 포함하고 있는 계층의 개수는 적지만 연산량은 상대적으로 아주 많은 조각이 존재할 수 있으며 반대로 포함하고 있는 계층의 개수는 많지만 연산량이 매우 적은 조각이 존재할 수 있다. 하지만 현 TFLite의 위임 정책은 이와 같은 특징이 반영되어 있지 않았다.
두 번째로, 몇 개를 선택할 것인지에 대한 변수(이하 N)를 사용자가 직접 설정하는 것은 비효율적이다. 사용자 입장에서는 어느 N값이 가장 효율적인지 알 수 없기 때문에 직접 여러 번 컴파일 및 테스트를 해봐야 한다. 아주 흔하게 사용되는 프레임워크임에도 불구하고, 사용자에게 불친절한 요소가 남아있다는 것이다. 또한, N의 디폴트 값은 “1”이다. 이는 CPU와 다른 가속 연산장치 (예: GPU) 사이의 데이터 교환 및 전환 비용이 크기 때문에[2], 이러한 전환을 최소화하며 동시에 최대한 많은 연산을 가속 연산장치에서 실행하기 위한 의도이다. 따라서 대부분의 사용자는 N을 디폴트 값인 1로 사용하며 DNN 추론작업을 진행할 것이다. 하지만, N의 값을 늘릴 때 가속 연산장치

에서의 연산량을 늘리는 이득이 데이터 교환 및 전환 비용으로 발생하는 손해보다 큰 경우가 존재할 수 있으며 이러한 경우의 존재를 사용자들은 놓칠 수 있다는 것이다.

III-2. 새로운 위임 정책의 설계 및 구현

현 TFLite의 위임 정책은 위와 같은 문제가 있기 때문에, 새로운 위임 정책을 제안하고자 한다. 제안하는 접근 방식은 그림 3과 같다.

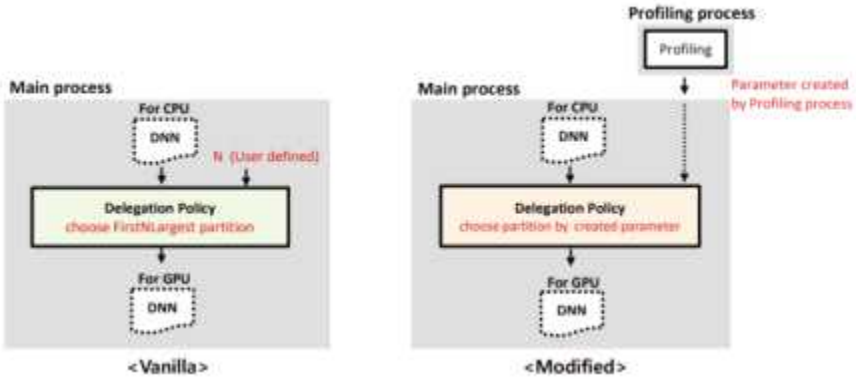

그림 3. 제안하는 위임 정책의 접근 방식

그림 3과 같이, 기존 TFLite의 위임 정책인 몇 개를 고를 것인지에 대한 변수를 사용자가 직접 설정하는 부분과 포함하고 있는 계층의 개수가 많은 순서대로 선택하는 부분을 없앤다. 그리고 실제 추론을 하는 메인 프로세스 이전에 프로파일링 프로세스를 둔다. 프로파일링 프로세스가 끝나면 메인 프로세스는 프로파일링 프로세스에서 전달된 파라미터를 그대로 활용하여 위임한다. 제안하는 접근 방식의 핵심은 프로파일링 프로세스인데, 이의 대략적인 구현도는 그림 4와 같다.

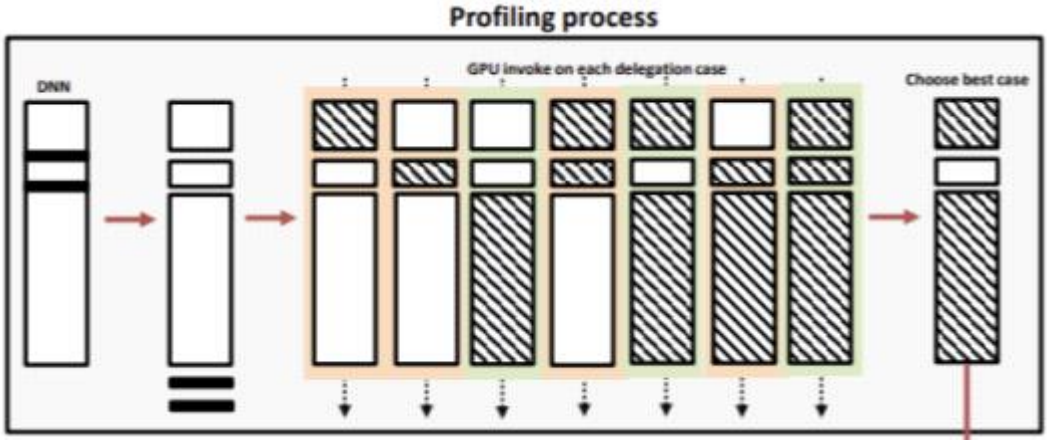

그림 4. 프로파일링 프로세스 구현도

그림 4처럼, 프로파일링 프로세스에서는 입력 DNN을 대상으로 가능한 모든 위임 조합을 만들어 놓는다. 그 후 생성해 놓은 각각의 조합으로 100번씩 추론하고 평균 추론 시간을 타임테이블에 기록한다. 생성해 놓은 모든 조합으로 추론 작업을 마친 후 완성된 타임테이블을 순회하며 가장 평균 추론 시간이 적었던 위임 조합을 파라미터 형식으로 메인 프로세스에 전달한다.

위와 같은 프로파일링 프로세스를 위해서는 "TFLite 프레임워크 내부 위임 정책 코드 수정"의 과정과 "TFLite 어플리케이션 코드 제작"의 과정이 필요하였다.

먼저 "TFLite 프레임워크 내부 위임 정책 코드 수정" 과정에서는, 반드시 포함하고 있는 계층이 많은 조각을 우선적으로 위임하는 정책을 사용자가 원하는 조각을 임의로 선택하여 위임하는 정책으로 변경하였다. 현 TFLite에서는 가속 연산장치로의 위임 과정에서 위임용 인스턴스를 생성한다. 이때 위임용 인스턴스의 생성 및 초기화 과정에서 인자로 structure 형태의 TFLiteDelegateOption (이하 위임조건용 구조체)을 사용한다. 이 위임조건용 구조체는 사용자와 가장 맞닿아 있는 부분부터, 내부 깊숙이 위임 정책이 존재하는 부분까지 서로 연결되어 있다. 이러한 위임조건용 구조체에 파라미터를 1개 추가하였고, 그 값을 사용자와 맞닿아 있는 부분에서 설정하도록 하였다. 따라서 사용자가 임의로 이 파라미터 값을 설정해 주면, 그 값이 위임조건용 구조체의 흐름에 따라 깊숙이 존재하는 위임 정책으로 이동할 수 있다. 위임 정책이 존재하는 부분에는, 조합 알고리즘을 사용하여 새롭게 추가되어 들어오는 파라미터의 값에 해당하는 인덱스의 위임 조합을 찾아 최종 위임하도록 한다. 예를 들어 위임 가능한 조각이 2개 존재하고 그중 1개를 선택하여 위임해야 하는 상황일 때, 파라미터 값을 0으로 하면 첫 번째 조각을 선택하여 위임하고, 1로 하면 두 번째 조각을 선택하여 위임하게 되는 것이다. 마찬가지로 위임 가능한 조각이 3개 존재하고 그중 2개를 선택하여 위임해야 하는 상황일 때, 파라미터 값을 0으로 하면 첫 번째와 두 번째 조각을 선택하여 위임하고, 1로 하면 첫 번째와 세 번째 조각을 선택하여 위임한다.

다음 "TFLite 어플리케이션 코드 제작" 과정에서는, 입력 DNN 모델의 프로파일링을 위한 어플리케이션 코드를 직접 제작하였다. 이 어플리케이션 코드의 구조는 "전처리", "위임", "추론", "결과 분석"으로 구성되어 있다.

먼저 "전처리"는, TFLite 추론을 위한 준비 단계이다. 우선 입력 모델과 이미지를 받아와 TFLite에서 요구하는 형식에 맞게끔 변환 과정을 거친다. 이후 TFLite에서 추론 작업을 도맡아 하는 핵심 인스턴스인 인터프리터를 생성 및 빌드한다.

다음 “위임”은, CPU가 아닌 가속 연산장치에서 추론 연산을 진행하도록 인터프리터를 수정하는 과정이다. “위임”의 과정을 통해 인터프리터를 수정하지 않고 바로 “추론” 단계로 넘어가면, 인터프리터가 기본 연산장치인 CPU 연산에 맞추어져 있기 때문에 CPU에서 추론 연산이 수행된다. 하지만 이 어플리케이션의 사용 목적은 “가속 연산장치를 가장 효과적으로 사용하기 위한 DNN 모델의 프로파일링”이기 때문에, 가속 연산장치를 사용하기 위해서는 “위임”의 과정을 통한 인터프리터의 수정이 필요하다.

앞서 “TFLite 프레임워크 내부 위임 정책 코드 수정” 과정에서, 새로운 파라미터가 추가된 위임조건용 구조체를 활용하여 사용자가 원하는 위임 방식으로 가속 연산장치를 사용할 수 있도록 위임 정책을 변경하였다. 이렇게 수정한 프레임워크 내부 구현부를 API 형태로 어플리케이션 코드에서 활용하도록 하였다. 자세히는 반복문을 선언하여, 반복문 안에서 각기 다른 파라미터 값을 가지는 위임조건용 구조체를 생성하고 이를 활용하여 각기 다른 위임 조합으로 인터프리터를 수정하게 하였다. 결론적으로 각각 다른 위임 조합으로 가속 연산장치를 활용한 추론을 할 수 있도록 준비하였다.

다음 “추론”은, 앞서 “위임” 단계를 통해 수정한 인터프리터를 활용하여 실제 추론 작업을 하는 단계이다. 각 위임 조합마다 100번씩 추론하고, 그에 대한 평균 추론 시간을 구한다. 그리고 위임된 조각의 번호와 해당 조각들로 위임했을 때의 평균 추론 시간을 묶어 1차원 벡터로 만들고, 이를 타임테이블에 기록한다. 타임테이블은 1차원 벡터를 각각의 인덱스로 가지는 2차원 벡터 형태의 자료구조이다. 최종적으로 가능한 모든 위임 조합으로 추론을 진행한 뒤 그 결과를 타임테이블에 저장한다.

다음 “결과 분석”은, “추론” 단계에서 최종 생성된 타임테이블을 순회하며 가장 평균 추론 시간이 적었던 조합의 위임된 조각의 번호를 찾는 단계이다. 그 후 이 데이터를 IPC 통신(소켓 통신)으로 메인 프로세스에 전달하게 된다.

이러한 TFLite 어플리케이션의 전체 과정은 [부록1]에서 참고할 수 있다.

Ⅳ. 실험 및 평가

Ⅳ-1. 실험 환경

실험은 nvidia사에서 만든 고성능 임베디드 보드인 jetson-nx[3]보드로 진행하였다. 사용된 DNN 모델은 대중적인 객체 인지 모델인 YOLOv4-tiny[4]이다. 실험에서는 GPU를 가속 연산장치로 활용하며, 백앤드 라이브러리로는 OpenGL을 사용하였다.

YOLOv4-tiny 모델은 총 152개의 계층으로 이루어져 있다. 이 중, GPU에 호환되지 않아 fallback 처리된 계층은 ADD, MUL, SPLIT, SPLIT_V이다. 이러한 특징을 가지는 입력 DNN으로부터, 그림 2와 같은 원리로 총 7개의 위임 가능한 조각이 생성된다. 이후 앞서 생성한 어플리케이션 형태의 프로파일링 프로세스를 실행시키면, 7개의 위임 가능한 조각으로부터 생성될 수 있는 모든 위임 조합을 대상으로 각각의 평균 추론 시간을 구할 수 있다.

IV-2. 실험 결과

그림 5는 프로파일링 프로세스를 실행한 결과를 시각화한 그래프이다. 그래프에서 "N"은 몇 개의 조각을 위임할 것인지에 대한 변수에 해당한다.

먼저 "Vanilla Case"는, Vanilla(이하 현) TFLite의 위임 정책을 사용하여 추론한 경우이다. 예를 들어 N이 1일 경우의 Vanilla case에서는 포함하고 있는 계층의 개수가 가장 많은 조각 1개만 위임하여 추론한다. 앞서 언급하였듯 현 TFLite에서는 N을 사용자가 직접 설정하며 디폴트 값은 1이다. 따라서 사용자 대부분은 N을 디폴트 값인 1로 설정하여 추론하지만, 결과를 보면 오히려 N이 커질수록 추론 성능이 좋아지는 경향성을 보인다.

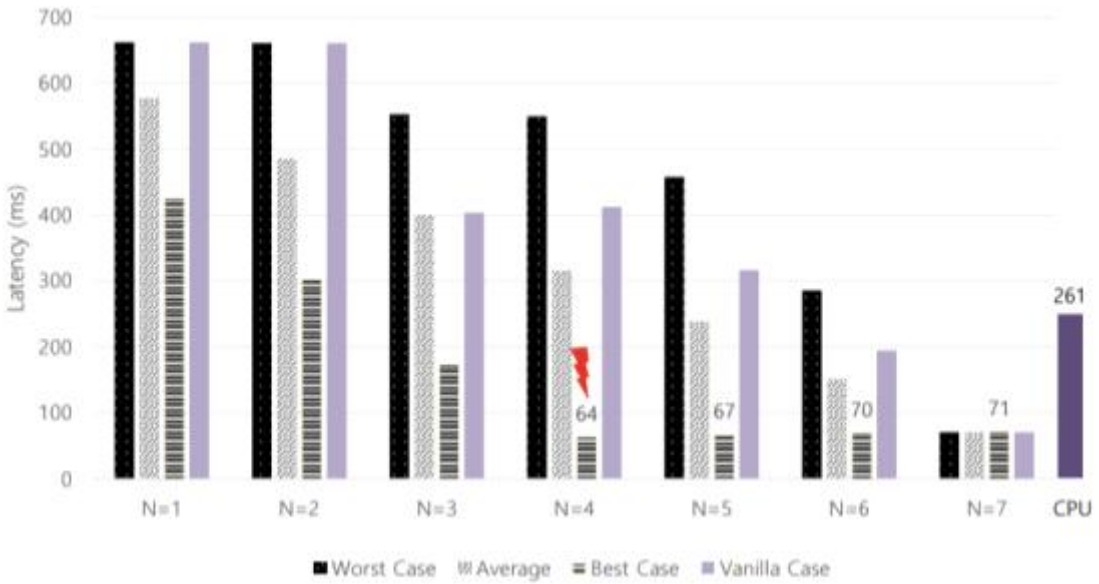

그림 5. 프로파일링 프로세스 실행 결과

심지어는 N이 1,2,3,4,5일 경우엔 오히려 가속 연산장치의 힘을 빌리지 않고 CPU만을 이용하여 추론할 경우보다 추론 성능이 좋지 않았다. 이러한 사실을 알 수 없는 사

용자들은 가속 연산장치의 효과를 더욱 높여보기 위하여 N값을 1이 아닌 다른 값들로 일일이 변경하여 컴파일하고 테스트해야 한다.

다음으로 “Worst Case”, “Best Case”, “Average”는, 본 논문에서 제안한 새로운 위임 정책을 사용하여 추론한 결과를 각 카테고리로 요약한 것이다.

먼저 “Worst Case”는 특정 N값에서 생성할 수 있는 모든 위임 조합으로 각각의 추론 시간을 구한 후 도출한, 전체 위임 조합 중 가장 추론 성능이 떨어진 경우이다. 결과 그래프를 보면, N이 증가할수록 Worst case에서의 추론 시간이 감소함을 알 수 있다.

다음 “Best Case”는 특정 N값에서 생성할 수 있는 모든 위임 조합으로 각각의 추론 시간을 구한 후 도출한, 전체 위임 조합 중 가장 추론 성능이 뛰어난 경우이다. 결과 그래프를 보면, N이 증가할수록 Best case에서의 추론 시간이 감소함을 알 수 있다.

마지막으로 “Average”는 특정 N값에서 생성할 수 있는 모든 위임 조합을 대상으로 각각의 추론 시간을 구한 후 도출한, 전체 위임 조합의 평균 추론 시간이다. 예를 들어 N이 1일 경우 위임 가능 조각이 7개이기 때문에 총 7가지의 위임 조합이 생길 수 있으며 이 7가지 위임 조합 각각의 추론 시간을 구한 뒤, 이들의 평균을 구한 값이 “Average”라고 할 수 있다. 결과 그래프를 보면, N이 증가할수록 Average 값이 작아짐을 볼 수 있다.

“Worst Case”, “Best Case”, “Average”의 경향성을 보면, 위임용 조각을 많이 선택하여 추론할수록 전반적인 추론 성능이 좋아짐을 알 수 있다. 또한 최종적으로 프로파일링 프로세스는 “Best Case” 중에서도 가장 높은 추론 성능을 보인 위임 조합을 찾아내게 되는데, 결과 그래프에서 보듯 최고 성능을 보인 위임 조합은 N이 4일 경우의 특정 조합이었으며 이 위임 조합을 활용한 추론 시간은 64ms이었다. 또한 정확성 평가 결과 각각의 위임 조합을 활용한 추론 결과의 정확도는 모두 일치하였다. 즉, 위임 조합을 어떻게 선택하여 추론하든 정확도의 손실은 발생하지 않았다.

IV-3. 실험 결과에 대한 분석 및 평가

이러한 실험 결과에 기반하여, 현 TFLite의 위임 정책에 비해 본 논문에서 제안한 프로파일링 프로세스를 활용한 새로운 위임 정책의 이점은 다음과 같다.

첫째, 본 논문의 새로운 위임 정책은 현 TFLite의 위임 정책보다 사용자의 입장에서 더 편리하다. 현 TFLite에서는 몇 개의 위임 조합을 선택하여 추론할 것인지에 대한 변수(이하 N)를 사용자가 직접 설정하기 때문에, 여러 번 N값을 수정하며 테스트를 수행한 후 비로소 추론 성능이 가장 뛰어난 위임 조합을 찾는다. 앞선 실험 결과를 보더라도, N값을 디폴트 값인 “1”로 설정했을 때 추론 성능이 좋지 못했으며 심지어는 N값이 “5” 이하일 경우 기본 연산장치인 CPU를 사용하여 추론했을 때보다 추론 성능이 좋지 않아 가속 연산장치를 사용하는 이득을 전혀 보지 못하였다. 이러한 내막을 전혀 모르는 사용자의 입장에서는 당혹스러울 수 있으며 현 TFLite의 위임 정책이 불편하게 다가올 것이다. 하지만 본 논문에서 제안한 새로운 위임 정책에 해당하는 프로파일링 프로세스를 단 한 번만 실행시키면, 추론 성능이 가장 뛰어난 위임 조합을 자동으로 찾아주게 된다.

둘째, 본 논문의 새로운 위임 정책에 해당하는 프로파일링 프로세스를 통해 선택된 최적의 추론 성능을 보인 위임 조합은 현 TFLite의 위임 정책으로 선택할 수 없는 조합일 수 있다. 앞선 실험 결과에서 보듯 현 TFLite의 위임 정책으로 생성할 수 있는 최적의 위임 조합의 추론 시간은 71ms이었지만, 프로파일링 프로세스를 통해 얻을 수 있는 최적의 위임 조합의 추론 시간은 64ms이었다. 이 위임 조합은 N이 4일 경우의 특정 조합이었는데, 이 조합은 현 TFLite의 위임 정책에서 N이 4일 경우 선택할 수 있는 조합이 아니었다. 오히려 N이 “4”일 경우의 현 TFLite 위임 정책대로, 포함하고 있는 계층의 개수가 가장 많은 조각 4개를 위임하여 추론하였을 때의 성능은 Best Case보다 Worst Case에 가까웠다.

다음으로, 구체적으로 어떠한 특징을 가지는 위임 조합으로 추론할 때 추론 성능이 향상되는지 경향성을 분석해 보았다. 각 위임 조합은 위임 가능한 조각들 중에서 각기 다른 방식으로 선택하여 얻어지기에, 먼저 위임 가능한 조각들의 특성을 분석해 보았다. 앞서 언급하였듯 테스트 DNN 모델은 YOLOv4-tiny이며, GPU를 가속 연산장치로 사용할 경우 그림 2와 같은 원리로 총 7개의 위임 가능한 조각이 생기게 된다. 그림 6의 그래프들은 차례대로 각 위임 가능한 조각이 보유하고 있는 계층의 수, 각 위

임 가능한 조각의 FLOPs (실제연산량), 각 위임 가능한 조각을 1개씩 선택해 위임하였을 경우의 latency (추론 시간)을 보여준다.

그림 6. 각 위임 가능 조각별 경향성 분석 그래프

그래프들을 분석하여 보면, 보유하고 있는 계층의 수는 추론 성능과 큰 연관성이 없었으며 오히려 실제 연산량이 추론 성능과 연관성이 있었다. 예를 들어 실제 연산량이 가장 많은 “4” 번 조각을 위임하여 추론하였을 경우 추론 성능이 가장 좋았으며 실제 연산량이 조금은 있는 “1”,“2”,“3” 번 조각을 각각 위임하여 추론하였을 경우 각 추론 성능이 상위권에 위치하였다. 또한 실제 연산량이 거의 존재하지 않는 “5”,“6”,“7” 번 조각을 각각 위임하여 추론하였을 경우 각 추론 성능이 하위권에 위치하였다.

정리하면, 현 TFLite의 위임 정책대로 보유하고 있는 계층의 개수가 많은 조각을 고르는 것이 아니라 실제 연산량이 많은 조각을 고를수록 추론 성능이 좋아지는 경향성을 보임을 알 수 있다. TFLite 개발자들은 대부분의 경우 포함하고 있는 계층의 개수가 많은 조각이 실제 연산량이 많을 것이라 예측하여 이러한 위임 정책을 개발했을 것이다. 하지만 위 그래프에서 포함하고 있는 계층의 개수가 많은 조각일지라도 실제 연산량은 적을 수 있다는 사실을 보인 것이다.

하지만 사실 그림 6은, 위임 가능한 조각 중 1개만 선택하여 위임하였을 경우의 경향성을 보인 것이다. 정말 실제 연산량이 많은 조각을 고를수록 추론 성능이 좋아지는지를 확인하기 위해선, 생성할 수 있는 모든 위임 조합에 대하여 경향성을 분석해야 한다. 그림 7은, 위임 가능한 모든 조합에 대하여 “각 위임 조합의 실제 연산량 총합”과 “각 위임 조합을 활용한 추론 성능”과의 경향성을 분석한 그래프이다.

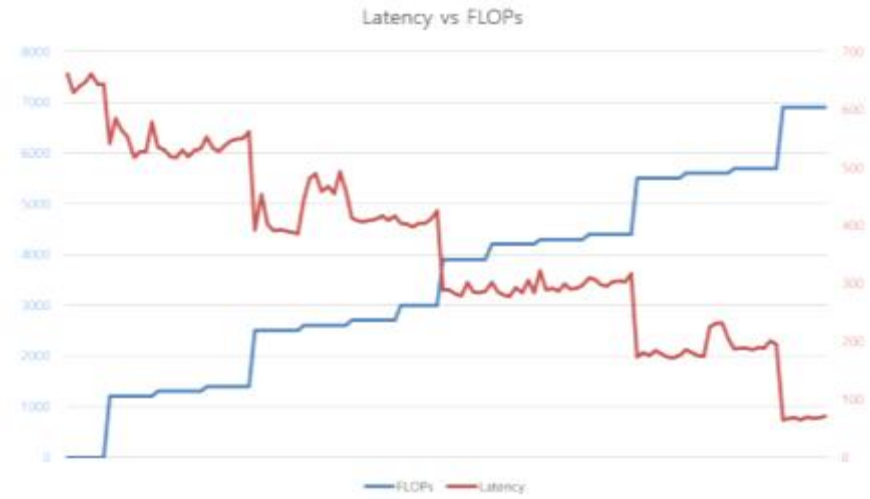

그림 7. 실제 연산량과 추론 성능과의 경향성 비교 그래프

그래프를 보면, 실제 연산량이 많은 위임 조합을 고를수록 latency (추론 시간)가 전반적으로 감소한다는 것을 알 수 있다. 즉 실제 연산량의 총합이 큰 위임 조합을 고를수록 전반적으로 추론 성능이 좋아진다는 것이다. 그렇다면, 현 논문에서는 프로파일링 프로세스를 두는 방식으로 새로운 위임 정책을 제안하였지만, 단순히 실제 연산량의 총합이 큰 위임 조합을 고르도록 위임 정책을 변경하는 것이 더 간편하고 직관적이지 않을까? 이에 대한 해답은 그림 8에서 찾을 수 있다.

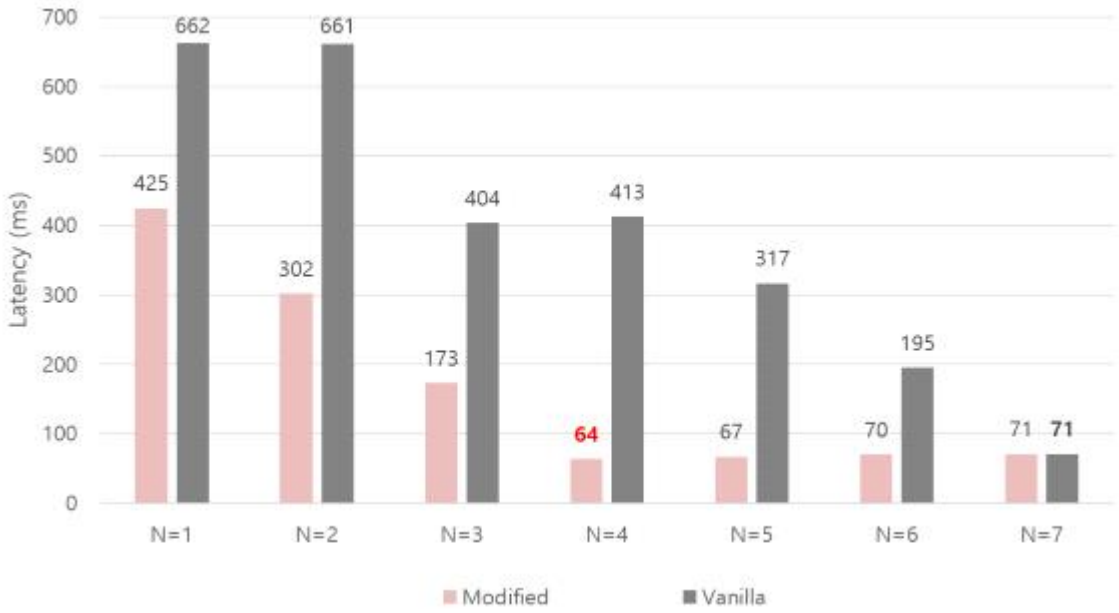

그림 8. 현 위임 정책과 제안하는 위임 정책의 추론 성능 비교 그래프

그림 8은, 현 TFLite의 위임 정책을 활용한 추론 성능과 본 논문에서 제안한 프로파일링 프로세스를 통한 위임 정책을 활용한 추론 성능을 비교한 그래프이다. 그래프의 ”Vanilla“ 항목은 현 TFLite의 위임 정책을 사용하여 추론한 결과이며, ”Modified“ 항목은 본 논문에서 제안한 프로파일링 프로세스를 통한 위임 정책을 사용하여 추론한 결과이다.

그림 8의 결과 그래프는 앞서 언급하였던 대로, 현 TFLite의 위임 정책에 비하여 본 논문에서 제한한 위임 정책 방식의 장점을 시각화하여 보여주고 있다. 즉 현 TFLite의 위임 정책과는 달리 단 한 번의 프로파일링 프로세스 실행으로 최적의 추론 성능을 보이는 위임 조합을 찾을 수 있으며 그 조합이 현 TFLite의 위임 정책으로 생성할 수 없는 위임 조합일 수 있음을 보인 것이다.

하지만 이 그래프에서 가장 눈여겨보아야 할 부분은, 앞서 그림 8에서 보인 경향성과는 다르게 실제 연산량의 총합이 가장 크도록 위임 조합을 선택하여 추론하였을 경우에 해당하는 N이 7일 경우의 위임 조합이 가장 추론 성능이 뛰어난 위임 조합이 아니라는 사실이다. 위임 가능한 조각이 총 7개이고, 실제 연산량의 총합이 가장 크도록 위임 조합을 선택한다면 당연히 7개의 조각 모두를 위임 조합으로 선택하는 것이 옳을 것이다. 하지만 위 경우의 추론 시간은 71ms에 불과하였고, 오히려 7개의 조각 중 실제 연산량이 많았던 상위 4개의 조각을 선택하여 위임할 경우의 추론 시간이 64ms로 가장 낮았다.

정리하면, 대부분 실제 연산량의 총합이 큰 위임 조합을 고를수록 추론 성능이 좋아지는 경향성을 그림 7에서 보이고 있지만, 그림 8에서는 그렇다고 실제 연산량의 총합이 가장 큰 위임 조합이 가장 추론 성능이 좋은 위임 조합은 아니라는 사실을 보인 것이다.

IV-4. 향후 연구 계획

본 논문에서는 사용자의 편의성 관점과 추론 성능 관점에서 현 TFLite의 위임 정책보다 더욱 효율적인 위임 정책 방식인 프로파일링 프로세스를 제안하였다.

하지만, 본 논문에서 제안하는 새로운 위임 정책을 사용하기 위해서는 위와 같은 프로파일링 프로세스를 반드시 최초에 한 번 실행시켜야만 한다는 단점이 있다. 이에, 후속 연구로 프로파일링 프로세스를 사용하지 않는 위임 정책을 개발해 볼 수 있다.

그림 7과 8에서 보였듯, 대부분 실제 연산량의 총합이 큰 위임 조합을 고를수록 추론 성능이 좋아지는 경향성을 보이지만 그렇다고 실제 연산량의 총합이 가장 큰 위임 조합이 가장 추론 성능이 좋은 위임 조합은 아닐 수 있다. 즉, 최적의 위임 조합을 생성해 주는 위임 정책을 만들기 위해서는, 실제 연산량만이 아니라 다른 변수도 고려해야 한다는 것이다. 그러한 또 다른 변수는 기본 연산장치와 가속 연산장치 간의 데이터 교환 비용일 가능성이 높다. 실제 연산량의 총합이 크도록 위임 조합을 고르면, 자연스레 선택하는 위임용 조각의 수가 늘어나게 된다. 선택하는 위임용 조각의 수가 증가하면 가속 연산장치를 더욱 사용하게 되므로 겉보기엔 추론 성능이 반드시 좋아질 것 같지만, 항상 그렇지는 않다. 선택하는 위임용 조각의 수가 많을수록, 그만큼 기본 연산장치와 가속 연산장치 간의 데이터 교환 횟수가 많아지기 때문이다. 예를 들어 위임용 조각들 중 1개를 선택하여 위임할 경우 기본 연산장치와 가속 연산장치 간의 데이터 교환 횟수가 2번이지만, 위임용 조각들 중 2개를 선택하여 위임할 경우 기본 연산장치와 가속 연산장치 간의 데이터 교환 횟수가 3번으로 늘어나게 된다. 이러한 기본 연산장치와 가속 연산장치 간의 데이터 교환 비용은 상당히 큰 편에 속한다[2]. 따라서, 가속 연산장치를 더욱 사용하게 될 때 발생하는 이득(위임용 조각을 가속 연산장치에서 추론함으로써 발생하는 단축 시간)과 손해(기본 연산장치와 가속 연산장치 간의 데이터 교환 시간) 간의 격차를 고려해야 한다는 것이다. 사실 이러한 이득과 손해는, 실험 하드웨어 환경의 영향을 크게 받는다. 실험 하드웨어 환경의 가속 연산장치의 성능이 월등히 뛰어나면, 위임용 조각을 가속 연산장치에서 추론하였을 때 발생하는 단축 시간이 많아서 기본 연산장치와 가속 연산장치 간의 데이터 교환 시간은 상대적으로 적어 보이게 된다. 이러한 환경에서는 실제 연산량이 큰 위임 조합을 되도록 많이 고르게 하는 정책이 유리할 것이다. 반대로 실험 하드웨어 환경의 가속 연산장치의 성능이 좋지 못하면, 마찬가지 원리로 실제 연산량의 총합이 큰 위임 조합을 되도록 적게 고르게 하는 정책이 유리할 것이다. 또한, 사용자가 편리하게 위임 정책을 사용할 수 있도록 현 TFLite의 위임 정책과 같이 사용자가 직접 위임용 변수의 튜닝을 해야 하는 번거로운 과정을 없애야 할 것이다.

이렇듯 현 TFLite의 위임 정책보다 사용자 편의성 및 추론 성능 면에서 뛰어난 최적의 위임 정책을 개발하기 위해서는 각 실험 하드웨어의 사양에 맞춘 유동적인 알고리즘이 필요하다고 할 수 있다. 후속 연구로, 다양한 실험을 통해 유동적인 알고리즘을 찾아 최적 위임 정책을 개발할 수 있을 것이라 기대한다.

Ⅴ. 결 론

본 논문을 정리하면, 현 TFLite의 위임 정책의 비효율성을 주장하며 사용자의 편의성 관점과 추론 성능 관점에서 더욱 효율적인 위임 정책에 해당하는 프로파일링 프로세스를 제시하였다. 이 프로파일링 프로세스는 기존 방식과는 달리 사용자가 최적 위임 조합을 찾기 위하여 특정 변수 값을 튜닝해야 하는 일을 없애고, 가장 뛰어난 추론 성능을 보이는 위임 조합을 단 한 번의 실행으로 찾아 제공해 준다. 또한 프로파일링 프로세스를 통하여 도출된 최적의 위임 조합은 현 TFLite의 위임 정책으로 생성할 수 없는 위임 조합일 수 있으며 더욱 추론 성능이 뛰어날 수 있다.

향후 연구로는, 각 하드웨어의 사양을 고려한 유동적인 위임 정책을 개발하여 별도의 프로파일링 과정이 없는, 더욱 사용자 친화적이며 뛰어난 추론 성능을 보장해 주는 위임 정책을 개발할 수 있을 것으로 기대한다.

최종적으로, 현 TFLite의 위임 정책의 비효율성을 개선한 새로운 위임 정책을 개발하여 TFLite의 공식 contributor가 되기를 기대한다.

참고문헌

[1] Zhou et al. : Supervised Streaming and Scheduling for GPU-Accelerated Real-Time DNN Workloads. IEEE Real-Time Embedded and Applications Symposium (RTAS), 2018

[2] J. Lee et al. On Device Neural Net Inference with Mobile GPUs. in Computer Vision and Pattern Recognition Workshops (CVPR), 2019.

[3] Jetson Xavier NX [Online]. Available: https://www.nvidia.com/ko-kr/autonomous-machines/embedded-systems/jetson-xavier-nx/

[4] Alexey Bochkovskiy. Darknet: Open Source Neural Networks in Python. 2020. Available online: https://github.com/AlexeyAB/darknet (accessed on 2 November 2020).

부 록

1. TensorFlow Lite application code ····· 17

부록 1. TensorFlow Lite application code

```
/* Copyright 2018 The TensorFlow Authors. All Rights Reserved.

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
==============================================================
================*/
#include <cstdio>
#include <vector>
#include <iostream>
#include <fstream>
#include "tensorflow/lite/interpreter.h"
#include "tensorflow/lite/kernels/register.h"
#include "tensorflow/lite/model.h"
#include "tensorflow/lite/optional_debug_tools.h"
#include "tensorflow/lite/delegates/gpu/delegate.h"
#include "opencv2/opencv.hpp"
#include "opencv2/opencv_modules.hpp"
```

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/core/core.hpp"
#include "yolo_with_DOT.h"
#include <chrono>

using namespace std;

#define YOLO_INPUT "../../mAP_TF/input/images-optional/"
#define Partition_Num 7  // nCr --> "n"  // for YOLOv4-tiny
// #define Max_Delegated_Partitions_Num 1  // nCr --> "r"  // hyper-param // Not use in full-auto
#define GPU
#define IMG_set_num 100 // "300" for mAP , "100" for DOT

std::vector<float> time_table;
std::vector<std::vector<float>> DOT_table;

#define TFLITE_MINIMAL_CHECK(x)                              ₩
  if (!(x)) {                                                ₩
    fprintf(stderr, "Error at %s:%d₩n", __FILE__, __LINE__); ₩
    exit(1);                                                 ₩
  }

uint64_t millis() {
    uint64_t ms = std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::high_resolution_clock::now().time_since_epoch()).count();
    return ms;
}
std::vector<std::vector<int>> mother_vec;
```

```
std::vector<std::vector<int>>make_mother_vec(int start , std::vector<int> vec,
int n, int total) {
    if (vec.size() == n) {
            mother_vec.push_back(vec);
            return mother_vec;
        }
        for (int i = start + 1; i < total; i++) {
            vec.push_back(i);
            make_mother_vec(i, vec, n, total);
            vec.pop_back();
        }
        return mother_vec;
}
void printCombination(int n, int k, int kth) {
    std::vector<int> result;
    mother_vec = make_mother_vec(-1, result, k, n);
    result = mother_vec[kth];
    std::cout <<"[";
    for (auto &data : result) std::cout << data << " ";
    std::cout <<"]";
    mother_vec.clear();
    return;
}
void print_time_table(std::vector<float> time_table){
  std::cout << "₩033[0;31mLatency for each case in DOT₩033[0m : "
<<std::endl;
  double min = *min_element(time_table.begin(), time_table.end());
  float bias = 0.3;
  for (int i=0;i< time_table.size(); i++){
      if(time_table.at(i) < min + bias) {
```

```
            printf("₩033[0;31m%d case's latency is %0.2fms₩033[0m₩n",i,time_table.at(i));
        }
        else{
            std::cout << i << " case's latency is : " << time_table.at(i) << "ms" << std::endl;
        }
    }
}
void print_DOT_table(std::vector<std::vector<float>> DOT_table){
    std::cout << "₩033[0;31m//////////////////////////////////////////////////////////₩033[0m" <<std::endl;
    std::cout << "₩033[0;31m//////////////////////////////////////////////////////////₩033[0m" <<std::endl;
    std::cout << "₩033[0;31m/////Print DOT Table//////₩033[0m" <<std::endl;
    for (int i=0;i<DOT_table.size();i++){
        std::cout << "N = " << i+1 <<std::endl;
        std::vector<float> time_table = DOT_table[i];
        float min = *min_element(time_table.begin(), time_table.end());
        float bias = 0.3;
        float sum = 0.0;
        for (int j=0;j< time_table.size(); j++){
            printCombination(Partition_Num,i+1,j);
            if(time_table[j] < min + bias) {
                printf("₩033[0;31mcase's latency is %0.2fms₩033[0m₩n",time_table[j]);
            }
            else{
                std::cout << "case's latency is " << time_table[j] << "ms" << std::endl;
```

```
            }
            sum += time_table[j];
        }
        printf("₩033[0;32m[END]...Choose_%d's average latency is %0.2fms₩033[0m₩n",i+1,sum/time_table.size());
    }
}
void find_best_case(std::vector<std::vector<float>> DOT_table){
    float min_value = DOT_table[0][0];
    int min_row = 0;
    int min_col = 0;
    for (int i = 0; i < DOT_table.size(); ++i) {
        for (int j = 0; j < DOT_table[i].size(); ++j) {
            if (DOT_table[i][j] < min_value) {
                min_value = DOT_table[i][j];
                min_row = i;
                min_col = j;
            }
        }
    }
    printf("Minimum CAES's value : %.2fms₩n", min_value);
    printf("Minimum CASE's N : %d₩n",min_row+1);
    printf("Minimum CASE's th : %d₩n",min_col);
    printf("Minimum CASE's combination : ");
    printCombination(Partition_Num, min_row+1,min_col);
    printf("₩n");
}
int combination(int n, int r) {
        if(n == r || r == 0) return 1;
        else return combination(n - 1, r - 1) + combination(n - 1, r);
```

```
}
void read_image_opencv(string image_name, vector<cv::Mat>& input){
        cv::Mat cvimg = cv::imread(image_name, cv::IMREAD_COLOR);
        if(cvimg.data == NULL){
                std::cout << "=== IMAGE DATA NULL ===₩n";
                return;
        }
        cv::cvtColor(cvimg, cvimg, cv::COLOR_BGR2RGB);
        cv::Mat cvimg_;
        cv::resize(cvimg, cvimg_, cv::Size(416,416));
        input.push_back(cvimg_);
}

int main(int argc, char* argv[]) {
  if (argc != 2) {
    fprintf(stderr, "minimal <tflite model>₩n");
    return 1;
  }
  const char* filename = argv[1];
  vector<cv::Mat> input;
  for (int N=1;N<=Partition_Num;N++){
  ////////////////////////////////////////////////////////////////////////////////////
  // Outer loop [1<=N<=Max]
    printf("₩033[0;33mLOOP START [N=%d]...₩033[0m₩n", N);
    int DOT = combination(Partition_Num, N);
    for (int dot = 0; dot<DOT; dot++){
        ////////////////////////////////////////////////////////////////////////////////////
        // Build interpreter on each dot case
        int image_number = 1;
        uint64_t average_time = 0;
```

```
        printf("₩033[0;32mDOT %d 's case starting...₩033[0m₩n", dot);

        // Load model
        std::unique_ptr<tflite::FlatBufferModel> model =
            tflite::FlatBufferModel::BuildFromFile(filename);
        TFLITE_MINIMAL_CHECK(model != nullptr);

        // Build interpreter
        tflite::ops::builtin::BuiltinOpResolver resolver;
        tflite::InterpreterBuilder builder(*model, resolver);
        std::unique_ptr<tflite::Interpreter> interpreter;
        builder(&interpreter);
        TFLITE_MINIMAL_CHECK(interpreter != nullptr);

        #ifdef GPU
        // Modify interpreter::subgraph when using GPU
        TfLiteDelegate *MyDelegate = NULL;
        const TfLiteGpuDelegateOptionsV2 options = {
              .is_precision_loss_allowed = 0,   //1
            .inference_preference                                          =
TFLITE_GPU_INFERENCE_PREFERENCE_FAST_SINGLE_ANSWER,
            .inference_priority1                                           =
TFLITE_GPU_INFERENCE_PRIORITY_MAX_PRECISION,
            .inference_priority2 = TFLITE_GPU_INFERENCE_PRIORITY_AUTO,
            .inference_priority3 = TFLITE_GPU_INFERENCE_PRIORITY_AUTO,
            .priority_partition_num = dot, // loop_num
            .experimental_flags = 1,
            .max_delegated_partitions = N, // default is "1"
        };
        MyDelegate = TfLiteGpuDelegateV2Create(&options);
```

```
TFLITE_MINIMAL_CHECK(interpreter->ModifyGraphWithDelegate(MyDelegate)
== kTfLiteOk);
        #endif GPU

        // Allocate tensor buffers.
        TFLITE_MINIMAL_CHECK(interpreter->AllocateTensors()          ==
kTfLiteOk);
        printf("=== Pre-invoke Interpreter State ===₩n");
        ////////////////////////////////////////////////////////////////////////////////
        // Push test image to input_tensor
        for (int loop_num=0;loop_num<IMG_set_num;loop_num++){
          // Load image
          std::string        image_name        =        YOLO_INPUT        +
std::to_string(image_number) + ".jpg";
          read_image_opencv(image_name, input);

          // Push image to input tensor
          auto input_tensor = interpreter->typed_input_tensor<float>(0);
          for (int i=0; i<416; i++){
            for (int j=0; j<416; j++){
              cv::Vec3b pixel = input[0].at<cv::Vec3b>(i, j);
              *(input_tensor + i * 416*3 + j * 3) = ((float)pixel[0])/255.0;
              *(input_tensor + i * 416*3 + j * 3 + 1) = ((float)pixel[1])/255.0;
              *(input_tensor + i * 416*3 + j * 3 + 2) = ((float)pixel[2])/255.0;
            }
          }
          // Run inference
          uint64_t START = millis();
          TFLITE_MINIMAL_CHECK(interpreter->Invoke() == kTfLiteOk);
```

```
            uint64_t END = millis();
            uint64_t Invoke_time = END - START;
            printf("₩n₩n=== Interpreter Invoke ===₩n");
            average_time += Invoke_time;

            // Output parsing
            TfLiteTensor* cls_tensor = interpreter->output_tensor(1);
            TfLiteTensor* loc_tensor = interpreter->output_tensor(0);
            yolo_output_parsing(cls_tensor, loc_tensor);

            // Output visualize
            #ifndef GPU
            yolo_output_visualize(image_name, image_number);
            #endif

            // Make txt file to get mAP
            //              make_txt_to_get_mAP(yolo::YOLO_Parser::result_boxes,
image_number, dot, Max_Delegated_Partitions_Num);

            // Re-initialize
            image_number+=1;
            input.clear();
            // interpreter->~Interpreter(); // Not use
          }
          ////////////////////////////////////////////////////////////////////////////////
          // Push result to time table
          // printf("₩033[0;31mDOT %d 's case's average invoke time [choose
N=%d]: %0.2fms₩033[0m₩n", dot, float(average_time/IMG_set_num), N);
          time_table.push_back(float(average_time/IMG_set_num));
          if (dot == DOT -1){
```

```
            // print_time_table(time_table);
          }
        }
        // Push each N's time table to parent time table
        DOT_table.push_back(time_table);
        time_table.clear();
    }
    //////////////
    // Search Best case recorded in DOT_table
    print_DOT_table(DOT_table);
    find_best_case(DOT_table);
    cv::waitKey(0);
          cv::destroyAllWindows();
    return 0;
}
```