Glow 컴파일러를 활용한 CPU상에서의 물체 탐지 가속화 연구

이제민 ○, 권용인, 유미선, 김영주, 김태호
한국전자통신연구원
{leejaymin, yongin.kwon, msyu, kr.yjkim, taehokim}@etri.re.kr

Accelerating Object Detection for CPU using the Glow Compiler

Jemin Lee○, Yongin Kwon, Misun Yu, Young-Joo Kim, Taeho Kim
Electronics and Telecommunications Research Institute

요 약

물체 탐지는 비전 응용에서 반드시 필요로 하는 핵심 기술이며 딥러닝을 적용한 알고리즘들이 보편화 되고 있다. 딥러닝 기반 물체 탐지 기술로는 YOLO가 있으며 다크넷 프레임워크를 통해서 공식적으로 지원 한다. 하지만 다크넷 프레임워크는 CUDA를 이용해서 GPU상에서의 가속화를 주로 지원한다. 기존 CPU 기반 인프라의 활용과 임베디드 환경에서의 CPU의 중요성을 고려해 볼 때 CPU상에서의 물체 탐지 추론 최적화는 필요하다. 본 연구에서는 딥러닝 컴파일러인 Glow가 제공하는 다양한 최적화 기법을 선택적으로 적용하여 TinyYOLOv2 모델로부터 CPU상에서 동작하는 여러 버전의 코드를 생성한 후, 다크넷에서 생성한 코드와 추론 지연 시간을 비교하였다 실험 결과 Glow는 CPU SIMD 연산에 맞춰서 메모리 레이아웃을 재배치하는 최적화로 인해 기존 다크넷 대비 2.7x 속도 향상을 가져왔다.

1. 서 론

물체 탐지는 물체의 위치와 이름을 인지하는 기술을 의미 한다. 물체 탐지는 얼굴 인식, 방범 감시, 자율주행 등의 다양한 컴퓨터 비전 시스템들에서 핵심적으로 사용되는 기술이다. 전통적인 물체 탐지 시스템들은 특징점 기반 방법[1], 서포트 벡터 머신 기반 방식[2] 등을 사용해서 만들어 졌다. 딥러닝 기술은 물체 탐지 분야에서도 많은 인식률 향상을 가져왔으며 여러 알고리즘들이 개발 되었다. 딥러닝 기반 물체 탐지 알고리즘 중에서 다크넷 프레임워크에서 제공하는 YOLO[3]는 많이 활용되고 있는 기술 중 하나이다.

하지만 다크넷 프레임워크는 NVIDIA CUDA를 활용하여 NVIDIA GPU에 대해서 추론 속도를 높이도록 최적화 되어있다. 기본적으로 서버, 엣지, 모바일 등에서 CPU 하드웨어는 보편적이며 특히 임베디드 GPU와 CPU는 성능 상의 차이가 크지 않다[4,5]. 따라서 이러한 점들을 고려할 때 CPU에서 딥러닝 추론 연산을 최적화 하는 것은 중요하다.

최근에 딥러닝 추론 연산을 컴파일러 기술을 이용해서 최적화하는 연구로 Glow[6]가 제안 되었다. Glow는 딥러닝 모델을 입력으로 받아서 CPU 타겟 하드웨어에 맞춰서 최적화된 실행 코드를 생성하는 기능을 제공한다.

본 연구에서는 CPU 환경에서 물체 탐지를 가속화를 위해서 Glow 컴파일러 최적화 기술을 이용한다. 물체 탐지 알고리즘으로는 TinyYOLOv2 모델을 사용하였고 기존 다크넷 프레임워크와 Glow로 최적화한 코드의 추론 지연 시간을 각각 비교 했다. Glow 컴파일러로 물체 탐지 수행을 위해서 기존 Glow에서 제공하는 이미지 분류 예제를 수정하여 물체 탐지 모델 수행이 가능 하도록 입력 값의 전처리와 출력 값의 후처리 기능을 구현했다. 최종 실험 결과 해당 방법은 기존 다크넷 보다 2.7x 빠른 실행 시간을 보였다. 성능 향상 분석 결과 Glow 컴파일러에서 수행하는 최적화들 중에서 가장 큰 역할을 한 것은 컨볼루션 연산을 Intel CPU SIMD(AVX2)에 맞춰서 메모리 레이아웃을 재배치한 것에 있었다.

2. 배경지식

2.1 ONNX 형식

Open Neural Network Exchange (ONNX)[7]는 딥러닝 모델을 표현하기 위해서 정의된 양식이다. 마이크로소프트를 중심으로 다수의 파트너 회사들이 협업하여 ONNX 양식을 관리하고 지원하고 있다. ONNX 형식으로 딥러닝 모델을 생성하면 TensorFlow나 PyTorch와 같은 잘 알려진 딥러닝 프레임워크에서 동작이 가능하다.

2.2 YOLO 알고리즘

다크넷 프레임워크는 딥러닝 기반 물체 탐지 알고리즘으로 YOLO 시리즈 모델을 지원한다. 본 연구에서는 YOLO 시리즈들 중에서 TinyYOLOv2 모델을 선택했다. TinyYOLOv2를 선택한 이유는 ONNX

표준 형식으로 변환시 원본 모델의 변형이 없기 때문에 다크넷과의 비교가 가장 정확하기 때문이다. 예를 들어 Reorg 레이어 등이 포함될 경우 ONNX 표준에 맞지 않아 Reshape, Transpose등의 조합으로 바꿔서 표현되고 이경우 다크넷과의 공정한 비교가 어렵다.

TinyYOLOv2의 상세한 구조는 표 1과 같다. TinyYOLOv2는 ONNX 표준 형식으로 완벽히 변환 가능한 Conv, LeakyReLU, BatchNorm. MaxPooling으로 신경망이 구성된다. 모델의 파라메터 사이즈는 약 63.5MB로 Conv와 BatchNorm.의 파라메터 값들이 차지한다. 이중 Conv7과 Conv8이 전체 파라메터 양에서 각각 29.7%, 59.5%의 비율을 나타낸다.

표 1 TinyYOLOv2 모델의 계층별 연산자 및 파라메터 크기

Type	Shape (CONV: DKKC, BN: SBMV)	Params (Byte)
Conv1	16x3x3x3	1,728
BN1	16x4	256
Conv2	32x3x3x16	18,432
BN2	32x4	512
Conv3	64x3x3x32	73,728
BN3	64x4	1,024
Conv4	128x3x3x64	294,912
BN4	128x4	2,048
Conv5	256x3x3x128	1,179,648
BN5	256x4	4,096
Conv6	512x3x3x256	4,718,592
BN6	512x4	8,192
Conv7	1024x3x3x512	18,874,368
BN7	1024x4	16,384
Conv8	1024x3x3x1024	37,748,736
BN8	1024x4	16,384
Conv9	125x1x1x1024	512,000

3. Glow 컴파일러 최적화 기술 분석

신경망 모델을 고속으로 처리하기 위한 컴파일러들로는 XLA, TVM, Glow가 있으며, 이러한 컴파일러들은 ONNX 형식으로 저장된 딥러닝 모델을 입력으로 받아 그래프와 중간 표현 그리고 백엔드 코드를 순차적으로 생성하여 최적화된 실행 코드를 생성 한다[8]. 본 논문에서는 Glow를 이용해서 TinyYOLOv2에 대해서 CPU 최적 코드를 생성하도록 구현 했다.

기본적으로 Glow는 TinyYOLOv2 모델을 그래프 형태로 변환하며 이 단계에서는 연산자 결합, 상수 폴딩, 데드 코드 제거, 공통 하위 표현식 제거 등을 수행한다. 해당 그래프는 다시 중간 표현으로 변환되고 이 단계에서는 Peephole 최적화, 데드 저장 제거, 메모리 할당 제거 끌어 올리기, 메모리 할당 연결하기, 불필요한 할당 제거, 파라메터 상수화, 버퍼 공유, 연산자 쌓기 최적화들을 수행 한다.

추가적으로 Glow 컴파일러는 백엔드가 CPU로 설정되었다면, transformPostLowering()를 실행 한다. 해당 패스는 기본 컨볼루션 연산을 타겟에 최적화된 컨볼루션으로 변경하는 작업으로 컨볼루션 필터 메모리 레이아웃을 SIMD 연산에 맞춰서 재조정한다. 기본 컨볼루션 데이터 레이아웃은 DKKC로 D는 필터의 개수로 출력 값의 깊이를 의미하며, C는 입력 채널을 뜻하며 K는 필터의 크기를 나타낸다. 이러한 메모리 레이아웃을 [D/8, K, K, C, 8]로 변경하게 된다. 이러한 최적화는 Core i7에서 지원하는 CPU SIMD 연산인 AVX2 수행에 있어서 효율적이다. AVX2는 256비트까지 한번에 벡터 처리가 가능하고 32비트 연산 수행시 8개까지 동시에 처리가 가능하다. 따라서 필터의 개수를 8로 분할하는 작업을 수행한다.

그림1은 TinyYOLOv2 모델에 대해서 transformPostLowering() 패스를 수행 했을 때 변경된 컨볼루션을 나타낸다. 필터의 개수(D)가 64의 배수인 경우에 대해서만 CPUConvDKKC8 연산자로 변환 된다. 따라서 표1에 기술된 TinyYOLOv2 Conv 9개 중에서 Conv1, Conv2를 제외한 7개의 Conv는 모두 변경 된다.

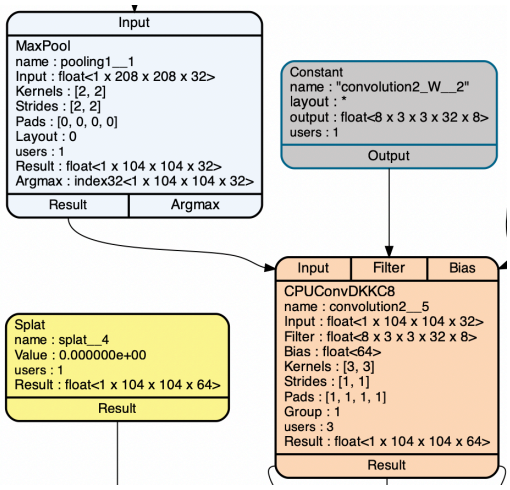

그림 1 TinyYOLOv2 모델에 transformPostLowering() 패스를 수행한 후의 Glow에서 생성한 TInyYOLOv2의 그래프의 일부분 모습

4. 성능 비교 분석

4.1 실험 방법

ONNX 형식으로 저장된 TinyYOLOv2 모델을 Glow 컴파일러로 실행한 결과와 기존 CFG기반 TinyYOLOv2 모델을 다크넷 프레임워크로 실행한 결과를 비교한다.

Glow 컴파일러를 이용해서 TinyYOLOv2를 실행하기 위해서 예제 코드를 새로 작성했다. Glow는 이미지 분류 예제를 제공하므로 해당 예제의 입력과 출력 부분을 변경하여 TinyYOLOv2가 실행 되도록 했다.

Glow 컴파일러의 각각의 최적화들의 효과를 분석하기 위해서 1) 모든 최적화 비활성화, 2) SIMD 메모리 레이아웃 최적화만 비활성화, 3) 모든 최적화 활성화 이렇게 세가지 경우로 나눠 실행 했다. 추론 지연 시간 측정은 같은 입력으로 각각의 조건에서 10번씩 반복 수행한 결과의 평균과 표준 오차를 사용 했다.

4.2 실험 환경

실험을 위해서 2018-Macbook-Pro 랩톱 인텔 i7 2.6 Ghz 메모리 32GB를 사용 했다. 소프트웨어는 모하비 10.14.6 운영체제 버전에 Glow는 llvm-8을 사용 했고 다크넷은 GCC4.2.1버전을 사용 했다.

4.3 성능 비교 및 분석

그림2는 다크넷 프레임워크를 이용해서 TinyYOLOv2를 실행한 결과 이미지를 나타낸다. 다크넷과 Glow를 이용한 실행 결과는 실수연산자의 연산 순서 차이에 따른 오차 범위 내에서 동일한 결과를 나타낸다.

그림3은 다크넷과 Glow 컴파일러를 이용해서 실행했을 때의 추론 지연 시간에 대한 평균값과 평균 오차를 나타낸다. Glow(noOpt)는 모든 최적화 패스들을 적용 하지 않았을 때의 결과이다. Glow(w/oDKKC8)은 3장에서 설명한 최적화를 전부 적용하고 오직 SIMD 메모리 레이아웃 최적화 패스만을 적용하지 않았을 때의 결과이고 Glow(Opt)는 모든 최적화 기능을 활성했을 때의 결과이다.

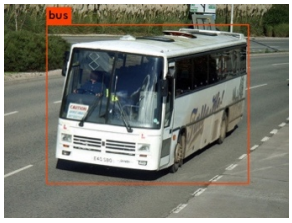

그림 2 TinyYOLOv2를 이용한 물체 탐지 결과

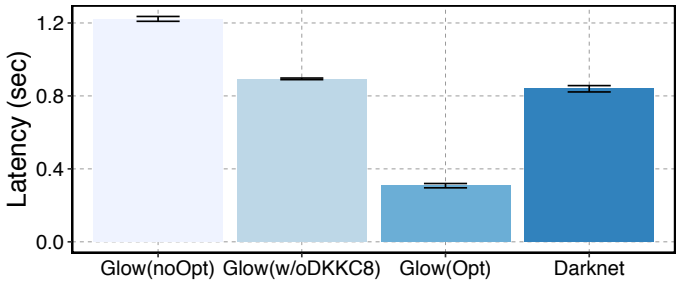

그림 3 실행 조건에 따른 TinyYOLOv2 모델의 추론 지연 시간

실험 결과 Glow(noOpt)는 평균 1.22초 추론 지연이 걸렸고 Glow(w/oDKKC8)은 평균 0.89초 추론 지연이 발생했다. 이것은 Glow(noOpt)보다 1.3x 향상된 결과이다. Intel i7 Core SIMD 연산인 AVX2를 위한 메모리 배치 최적화까지 모두 적용한 Glow(Opt)의 경우 추론 지연시간은 평균 0.31초로 Glow(w/oDKKC8) 보다 2.9x 향상 되었다. Glow(Opt)의 결과는 기존 다크넷 보다 2.7x 향상된 결과이며, Glow 컴파일러가 Intel i7 core CPU에 대해서 효율적인 실행 코드를 생성함 의미한다. 즉, 기존의 많은 CPU 인프라를 고려 했을 때 활용 가치가 높은 결과이다.

5. 결론

본 논문에서는 물체 탐지 딥러닝 모델 중 많이 활용되는 TinyYOLOv2를 Glow 컴파일러를 이용해서 CPU에서 가속화하는 방법을 다뤘다. 실험 결과, 기존 다크넷 프레임워크를 이용해서 실행하는 것보다 추론 시간 측면에서 2.7x 향상된 결과를 얻었다. Glow 컴파일러가 지원하는 CPU SIMD 연산 메모리 레이아웃 최적화 기능을 활용하면 다양한 CPU 타겟에 대해서 최적화된 코드를 생성 할 수 있으므로 여러 CPU 인프라에서 유용하게 활용 될 수 있을 것이다. 향후에는 임베디드 CPU를 포함한 여러 종류의 CPU에 대해서 최적화를 수행하는 패스를 Glow에 구현해 볼 계획이다.

감사의 글

이 논문은 2020년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임 (No.2018-0-00769,인공지능 시스템을 위한 뉴로모픽 컴퓨팅 SW 플랫폼 기술 개발)

참 고 문 헌

[1] P. Viola, and M. Jones, “Robust real-time object detection,” International Journal of Computer Vision, vol. 57, issue 2, pp. 137-154, 2004.
[2] N. Dalal, and B. Triggs, “Histograms of oriented gradients for human detection,” IEEE CVPR, pp. 886-893, 2005.
[3] J. Redmon, and A. Farhadi, “YOLO9000: Better, Faster, Stronger,” arXiv preprint arXiv:1612.08242, 2016.
[4] Yizhi Liu, et al., “Optimizing CNN model inference on CPUs,” USENIX ATC, pp. 1025-1040, 2019.
[5] Wang, Siqi, et al., “Neural Network Inference on Mobile SoCs,” IEEE Design & Test, Early Access, 2020.
[6] Rotem, Nadav, et al., "Glow: Graph lowering compiler techniques for neural networks." arXiv preprint arXiv:1805.00907, 2018.
[7] ONNX, https://github.com/onnx/onnx/, 2020.
[8] 유미선 외 3 명, “딥러닝 컴파일러 성능 비교,” 한국컴퓨터종합학술대회 pp. 1076-1078, 2019.