임베디드 CPU에서 Winograd 컨볼루션 알고리즘에 기반한 Depthwise 컨볼루션의 구현

(Implementation of Depthwise Convolution based on Winograd Convolution Algorithm on an Embedded CPU)

노 수 민, 박 상 수, 이 민 영, 정 기 석*
한양대학교
(Soo-Min Rho, Sang-Soo Park, Min-Young Lee, Ki-Seok Chung)
(Hanyang Univ. Seoul)

Abstract : The depthwise convolution layer is often used in lightweight convolutional neural networks (CNNs). In this study, the Winograd convolution algorithm is used to carry out depthwise convolution, and the proposed method is implemented using NEON instructions on an ARM Cortex-A53 CPU. The performance of the proposed method is evaluated with respect to two tensor storing orders (NCHW and NHWC) and various chunk sizes. In addition, we evaluate end-to-end inference latency of MobileNet-V2 using our method. The results show our method achieves about 2x speedup against the conventional convolution lowering method.

Keywords : Depthwise convolution, Winograd convolution algorithm, SIMD, Tensor ordering

Ⅰ. 서 론

CNN 모델들은 많은 연산량과 파라미터 수를 가지고 있어, IoT 장치 등 제한된 성능의 임베디드 기기에서 빠르게 추론하는 것은 용이하지 않다 [1]. 따라서 추론은 일반적으로 데이터 센터의 컴퓨팅 자원에 의존하며, 데이터 센터와의 통신으로 인한 Latency 및 전력 소모, 개인정보 유출 등의 문제가 있다 [2].

이러한 문제를 해결하기 위해 임베디드 플랫폼 자체에서 추론을 실행하기 위해 모델을 경량화하는 것이 중요하다. 경량화된 CNN 모델 중 일부는, Depthwise 컨볼루션과 Pointwise 컨볼루션을 통해 특징을 추출하는데, Pointwise 컨볼루션 대비 Depthwise 컨볼루션은 연산 성능이 좋지 않아, 성능 저하를 초래한다 [5].

*Corresponding Author (kchung@hanyang.ac.kr)
노수민, 박상수, 이민영, 정기석: 한양대학교
※ 본 논문은 2022년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임 (No. 2022-0-00153, 빔포밍 경로 분산을 이용한 AI 네트워크관리 기반 인빌딩용 O-RU 개발).

본 논문에서는 Depthwise 컨볼루션의 연산 과정에서 발생되는 성능 저하를 해결하기 위해 Winograd 컨볼루션 기반의 연산 방법을 사용하였다. 또한 해당 방법을 적용한 Depthwise 컨볼루션을 딥러닝에서 tensor를 저장하는 두 가지 데이터 저장 포맷 (NCHW, NHWC)과 여러 chunk 크기를 다양하게 조합하여 사용할 수 있도록, ARM社의 Cortex-A53의 NEON 명령어를 사용해 구현하였으며, 다양한 Depthwise 컨볼루션 레이어를 대상으로 연산 성능을 비교하였다. 또한 경량화된 신경망의 대표적인 모델 중 하나인 MobileNet-V2 모델 [5]을 벤치마크로 하여 제안한 연산 방법을 통한 딥러닝 모델 추론 성능 개선을 확인하였다.

Ⅱ. 본 론

1. Background

1.1 Depthwise convolution

Depthwise separable 컨볼루션은 피처맵의 spatial 차원의 특징 추출과 depth 차원의 특징 추출을 나누어서 수행한다 [5]. 그림 1과 같이, width 차원 (W)과 height 차원 (H)의 특징을 추출하는 과

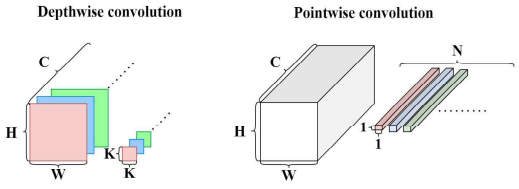

그림 1. Depthwise 컨볼루션 레이어 및 Pointwise 컨볼루션 레이어의 구조

정은 Depthwise 컨볼루션에 해당되며, channel 차원 (C)의 특징을 추출하는 과정이 Pointwise 컨볼루션에 해당한다.

Depthwise 컨볼루션은 모든 연산이 channel에 독립적으로 수행되며, 피처맵을 행렬의 형태로 변경하는 Dimension-lowering (IM2COL, IM2ROW) 기법을 적용하여 tensor의 컨볼루션 연산을 행렬-벡터 곱 연산으로 바꾼 뒤, BLAS 라이브러리에서 제공하는 행렬-벡터 곱 연산 함수를 사용하여 연산할 수 있다 [4].

1.2 Winograd convolution algorithm

Winograd 컨볼루션 알고리즘은 컨볼루션 연산의 곱셈 수를 2.25배 감소시키는 연산 방법으로, 수식 1 및 수식 2와 같은 형태를 가진다[3].

$$F = GfG^T, \quad X = B^T xB \tag{1}$$

$$Y = A^T[F \odot X]A \tag{2}$$

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \quad G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{-1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} \tag{3}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

Winograd 컨볼루션은 3 단계의 절차를 통해서 컨볼루션 연산을 수행한다 (수식 1 및 수식 2).

1) 입력 및 필터 데이터의 Winograd 도메인으로 변환 ($F = GfG^T$ 및 $X = B^T xB$).
2) element-wise 곱셈 연산 ($F \odot X$).
3) 2)의 결과를 Winograd 도메인에서 역변환 ($A^T(F \odot X)A$).

수식 1은 필터 데이터 (f)와 입력 데이터 (x)가 Winograd 변환 행렬 (G 및 B^T)과의 연산을 통해 Winograd 도메인으로 변환되는 과정을 나타내며, 수식 2는 Winograd 도메인으로 변환된 필터 및 입력 (F, X)을 이용한 element-wise 곱셈 연산의 결과를 Winograd 역변환 행렬 (A^T)를 통해서 역변환 하는 과정을 나타낸다. 수식 3은 Winograd 변환 행렬 및 역변환 행렬의 값을 보여준다.

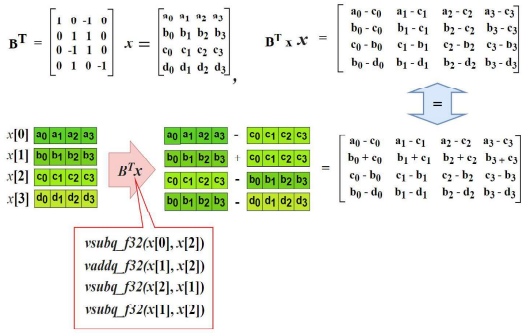

그림 2. 행렬 곱셈 기반의 Winograd 도메인 변환 방법과 벡터 덧셈/뺄셈 기반 변환 방법의 비교

본 논문에서 Winograd convolution은 F(2x2, 3x3, 4x4)로 표현하며, 각각의 인자는 수식 1 및 수식 2의 2x2 출력 행렬 (Y), 3x3 필터 행렬 (f), 4x4 입력 행렬 (x)의 차원을 의미한다.

2. Winograd convolution의 효율적인 연산 방법

본 논문에서 제시하는 Winograd 컨볼루션 기반 연산 방법은, [3]에서 제시한 Winograd 도메인 변환 연산 방법을 적용한다.

Winograd 도메인 변환인 $B^T xB$ 연산 과정 중 $B^T \times x$ 연산은 Winograd 변환 행렬 B^T와 입력 행렬 x의 행렬 곱셈으로 연산 할 수 있다. 반면 [3]에 따르면, 그림 2에 표현된 것처럼 행렬 x의 행 벡터 (x[0]-x[3]) 간 덧셈 및 뺄셈을 적용함으로써 $B^T \times x$ 의 행렬 곱셈과 같은 결과를 더 적은 연산량으로 구할 수 있다. $x \times B$ 연산의 경우도, 행렬 x의 column 벡터 간 덧셈 및 뺄셈을 적용함으로써 연산할 수 있다.

Winograd 컨볼루션의 element-wise 곱셈 ($F \odot X$)은 벡터 곱셈 명령어를 사용하여 연산을 수행하며, Winograd 도메인으로의 역변환 과정은 Winograd 도메인으로의 변환 과정과 유사한 방법으로 수행한다.

3. Tensor ordering 포맷 및 chunk 크기에 따른 Depthwise 컨볼루션의 구현

제시한 Winograd 기반 연산 방법은 두 종류의 tensor 저장 포맷과 여러 chunk 크기를 다양하게 조합하여 사용할 수 있도록 구현되었다. 또한 chunk 데이터는 4x4 행렬로 나누어 연산을 수행하게 되는데, 이 4x4 행렬을 Winograd region으로 정의하였고 F(2x2, 3x3, 4x4)를 수행하여 연산하게 된다.

3.1 NCHW 포맷의 Winograd 도메인 변환 구현

NCHW 포맷은 NxCxHxW 크기의 차원을 가지는 피처맵을 tensor의 width - height - channel - batch 의 순서로 저장하는 데이터 저장 포맷을 의미한다. NCHW 포맷에서는 그림 3과 같이 width 차원 순서로 데이터들을 벡터 레지스터에 저장할 수 있고, 4개의 벡터 레지스터 ($x[0]$-$x[3]$)를 사용해서 하나의 Winograd region (4x4 input)을 load 할 수 있다.

이때 각 벡터 레지스터는 Winograd region의 행 벡터에 해당하게 되므로 그림 2의 방법처럼 $B^T \times x$ 연산을 벡터 덧셈, 뺄셈을 이용하여 연산할 수 있지만, 벡터 레지스터가 열벡터에 해당하지는 않기 때문에 벡터 간 덧셈, 뺄셈 연산으로 $x \times B$의 연산을 수행할 수 없다.

$$S^T = B^T \times x, \quad B^T x B = (B^T \times (S))^T \tag{4}$$

본 논문에서는 [3]에서 제시한 수식 4를 이용해서 $B^T x B$ 연산을 행벡터 간 덧셈, 뺄셈 연산만으로 수행할 수 있는 $B^T \times x$ 연산, $B^T \times S$ 연산 및 행렬 전치 연산을 사용하는 형태로 변환하여, Winograd 도메인 변환 과정을 구현하였다.

3.2 NHWC 포맷의 Winograd 도메인 변환 구현

NHWC 포맷은 피처맵을 tensor의 channel - width - height - batch 의 순서로 저장하는 데이터 저장 포맷을 의미한다.

이 경우, 그림 3처럼 하나의 벡터 레지스터는 4x4x4 크기를 가지는 4개의 Winograd region에서 channel 방향으로 놓이게 되고, 4개의 region에서 동일한 인덱스를 가지는 스칼라 값들을 저장하게 된다. 이러한 벡터 레지스터 16개 ($x[0][0]$ - $x[3][3]$)를 spatial 차원으로 사용하는 것으로 4개의 Winograd region을 저장하게 된다.

NHWC 포맷에서는 데이터가 channel 차원으로 저장이 되어 있어 하나의 Winograd region의 모든 데이터가 메모리 상에서 연속적으로 존재하지 않게 된다. 따라서 $B^T x B$ 연산은 그림2의 벡터 방식이 아닌, 스칼라 덧셈 및 뺄셈을 사용하여 구현하게 된다.

반면 4개의 Winograd region을 한 번에 변환하는 경우, 그림 4와 같이 벡터 레지스터 및 벡터 명령어

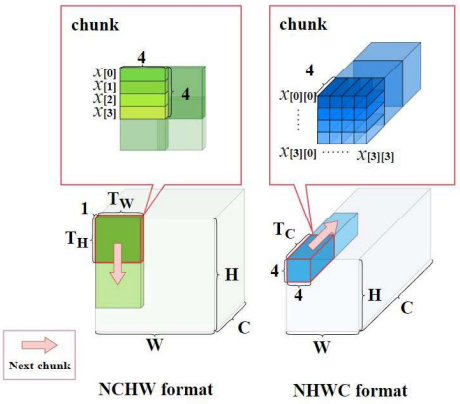

그림 3.NCHW 포맷 및 NHWC 포맷에서의 데이터 chunk 및 Winograd region

```
for (i = 0; i < 4; i++) {
  xB[i][0] = vsubq_f32(x[i][0], x[i][2]);
  xB[i][1] = vaddq_f32(x[i][1], x[i][2]);
  xB[i][2] = vsubq_f32(x[i][2], x[i][1]);
  xB[i][3] = vsubq_f32(x[i][1], x[i][3]);
}  // xB

for (j = 0; j < 4; j++) {
  Y[0][j] = vsubq_f32(xB[0][j], xB[2][j]);
  Y[1][j] = vaddq_f32(xB[1][j], xB[2][j]);
  Y[2][j] = vsubq_f32(xB[2][j], xB[1][j]);
  Y[3][j] = vsubq_f32(xB[1][j], xB[3][j]);
}  // Y = B^T(xB)
```

그림 4. NHWC 포맷에서 Winograd 도메인 변환 과정

를 사용하여 하나의 Winograd region의 $B^T x B$ 연산에서 사용된 스칼라 연산을 channel 방향으로 확장하여 4개의 Winograd region을 병렬로 처리할 수 있게 된다[3].

3.3 chunk 크기에 따른 구현

NCHW와 NHWC 포맷을 적용한 연산은 chunk 단위로 데이터의 load와 store를 수행하며, 이때 여러 개의 벡터 레지스터를 사용한다. 그림 2와 같이, 여러 벡터 레지스터에 나눠서 저장된 chunk는 여러 Winograd region으로 나뉜 뒤, 각각의 Winograd region에 F(2x2, 3x3, 4x4)을 적용함으로써 처리한다. NCHW 포맷의 경우 그림 2와 같이 T_WxT_Hx1의 형태로 chunk를 구성하였으며, 다음 chunk를 load 할 때 height 차원 방향으로 chunk를 load 하도록 구현하였다.

NHWC 포맷의 경우, 4x4xT_C 크기의 chunk를 가지도록 구현하였으며, 다음 chunk를 load 할 때는 channel 차원 방향으로 load 하도록 구현하였다. 실험에서는 NCHW 포맷의 chunk 차원인 T_W, T_H와 NHWC 포맷의 chunk 차원인 T_C를 다양하게 구성하여 진행했다.

4. 실험 결과 및 분석

4.1 두 포맷과 chunk 크기에 따른 성능 비교

제안하는 Winograd 컨볼루션 알고리즘 기반의 Depthwise 컨볼루션 연산 방법은 NCHW 및 NHWC 포맷에 대하여, chunk 크기 (T_W, T_H, T_C)를 다양하게 조합하며 성능을 측정 및 평가하였다. 실험은 Odroid-N2 보드에서 MobileNet-V2에서 사용되는 Depthwise 컨볼루션 레이어들을 대상으로 소요되는 시간을 측정하였다.

그림 5처럼 W와 H 차원이 28보다 큰 경우, 해당 레이어는 T_W, T_H의 값이 10인 NCHW 포맷을 사용하였을 때 다른 구현 방법 대비 성능이 가장 좋았고, 그 밖의 레이어에서는 T_W, T_H의 값이 각각 6, 4인 NCHW 포맷을 사용하였을 때 최적의 성능을 보였다. 반면 NHWC 포맷의 경우, 전반적으로 NCHW 포맷에 비해 성능이 좋지 않은 결과를 보였다.

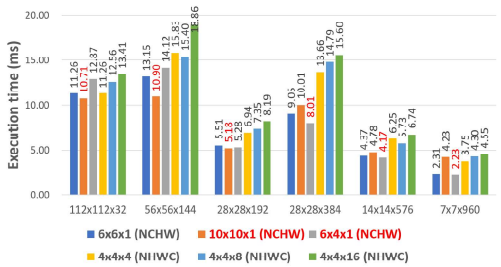

그림 5. tensor 포맷 및 chunk 크기에 따른 Depthwise 컨볼루션 레이어의 연산 성능 비교

4.2 MobileNet-V2 모델의 추론 시간 평가

4.1의 실험 결과를 반영하여 MobileNet-V2 모델의 end-to-end 추론 성능을 Darknet 프레임워크를 통해 평가하였다 [5]. 4.1의 실험 결과를 적용하여, tensor의 포맷은 모두 NCHW 포맷을 사용하였으며 Depthwise 컨볼루션의 W, H 차원 크기가 28을 초과하는 레이어는 T_W, T_H의 값을 10으로 지정하였고, 나머지 레이어는 T_W, T_H의 값을 각각 6, 4로 지정하였다.

Dimension-lowering 방법과 BLAS 라이브러리(OpenBLAS)를 사용한 모델 추론을 baseline으로 지정하여 성능을 비교하였고, 표 1의 결과와 같이, 제안한 Depthwise 컨볼루션 연산에 NCHW 포맷을 사용하여 모델 추론을 한 경우, baseline 대비 약 2배 정도의 추론 성능 향상을 내는 것을 확인할 수 있다.

표 1. MobileNet-V2 추론의 성능 비교

	baseline	ours (Winograd)	speedup
MobileNet-V2	9.75 ms	4.85 ms	2x

Ⅲ. 결 론

본 논문은 임베디드 CPU 환경에서 Depthwise 컨볼루션을 효과적으로 연산하기 위한 Winograd 컨볼루션 기반의 연산 방법 제안한다. 제안하는 방법을 두 가지 tensor 저장 포맷 (NCHW, NHWC)과 여러 chunk 크기에 대하여, 다양한 Depthwise 컨볼루션 레이어를 대상으로 연산 성능을 비교하였고 NCHW 포맷 기반의 방법이 NHWC 포맷 기반의 방법보다 전반적으로 높은 성능을 내는 것을 확인하였다. 제안하는 방법을 MobileNet-V2 모델은 기존의 Depthwise 컨볼루션 연산 방법을 사용한 모델 추론 대비, 2배 정도 성능 향상을 보였다.

References

[1] A. Canziani et al. “An analysis of deep neural network models for practical applications.” arXiv preprint arXiv:1605 .07678, 2016.

[2] Sze, Vivienne et al. “Efficient Processing of Deep Neural Networks: A Tutorial and Survey.” Proceedings of the IEEE 105 (2017): 2295-2329.

[3] Maji, Partha P. et al. “Efficient Winograd or Cook-Toom Convolution Kernel Implementati-on on Widely Used Mobile CPUs.” 2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2) (2019): 1-5.

[4] Chellapilla et al. “High Performance Convolutional Neural Networks for Document Processing.” (2006).

[5] Sandler et lal. “Mobilenetv2: Inverted residuals and linear bottlenecks.” CVPR, 2018.