creative commons

COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학석사 학위논문

모바일 GPU에서의 딥 러닝 모델 추론을 위한 메모리 관리 및 최적화

Memory Management and Optimization for Deep Learning Model Inference on Mobile GPU

2022년 8월

서울대학교 대학원

컴퓨터공학부

강 수 연

모바일 GPU에서의 딥 러닝 모델 추론을 위한 메모리 관리 및 최적화

Memory Management and Optimization for Deep Learning Model Inference on Mobile GPU

지도교수 이 재 진

이 논문을 공학석사 학위논문으로 제출함
2022년 5월

서울대학교 대학원
컴퓨터공학부
강 수 연

강수연의 석사 학위논문을 인준함
2022년 7월

위 원 장 _____ 김 진 수 _____ (인)

부위원장 _____ 이 재 진 _____ (인)

위 원 _____ 이 재 욱 _____ (인)

초록

모바일에서의 이미지 처리 가속 기술이 나날이 중요해지고 있다. 모바일에서는 CPU와 GPU 뿐 아니라 이미지 처리를 위한 ISP(Image Signal Processing) 및 DSP(Digital Signal Processing)를 함께 탑재하고 있어, 각 프로세서가 담당하는 이미지 처리 알고리즘에 따라 이미지를 처리한다. 하지만 모바일에서의 이미지 처리 과정이 점점 복잡해짐에 따라 ISP나 DSP를 구축하는데 많은 노동력과 비용이 들어가게 된다. 그럼에도 불구하고 ISP를 대체할 수 있는 모바일 GPU를 활용한 이미지 처리 기술 개발이 부족한 실정이다. ISP를 대체하는 딥 러닝 모델인 PyNET이 존재하지만 딥 러닝 프레임워크로 개발된 모델은 모바일에서 메모리 부족 문제로 추론이 불가능하다.

본 연구에서는 ISP를 대체하는 딥 러닝 모델인 PyNET을 모바일 GPU에서 추론하기 위한 최적화를 진행한다. 이 과정에서 딥 러닝 모델을 추론할 때 발생하는 메모리 부족 문제와 latency 문제를 해결하기 위한 다양한 메모리 관리 기법과 OpenCL 기반의 convolution 최적화 기법을 제시한다. 최적화를 통해 딥 러닝 프레임워크로는 모바일에서 추론이 불가능한 모델을 실제 모바일 디바이스에서 실험함으로써 이를 검증한다.

주요어 : 모바일 GPU, 딥 러닝, 메모리 관리 기법, Convolution 최적화, OpenCL.
학 번 : 2012-20726

목 차

제 1 장 서론 ····· 1

제 2 장 관련 연구 ····· 3

제 3 장 모바일 디바이스 선정 ····· 5

제 4 장 딥 러닝 모델 분석 ····· 7

제 1 절 PyNET 모델 개요 ····· 7

제 2 절 모델 구조 및 추론 과정 ····· 7

제 3 절 Pytorch 기반 PyNET 애플리케이션 분석 ···· 9

제 5 장 메모리 관리 기법 ····· 12

제 1 절 모델 구조 변경 ····· 12

제 2 절 입력 분할 ····· 13

제 3 절 메모리 풀 ····· 14

제 4 절 메모리 스와핑 ····· 15

제 5 절 Zero copy ····· 15

제 6 절 Half-precision floating-point ····· 16

제 6 장 Convolution 최적화 ····· 17

제 1 절 Direct convolution ····· 17

제 2 절 Implicit GEMM Convolution ····· 18

제 3 절 Winograd Convolution ····· 20

제 7 장 실험 결과 및 분석 ····· 31
제 1 절 모바일 실험 환경 ····· 31
제 2 절 Convolution Configuration Set ····· 31
제 3 절 실험 결과 ····· 33

제 8 장 결론 ····· 47

참고문헌 ····· 38
Abstract ····· 40

표 목 차

[표 3-1] ····· 5
[표 4-1] ····· 10
[표 4-2] ····· 11
[표 4-3] ····· 11
[표 6-1] ····· 20
[표 6-2] ····· 29
[표 6-3] ····· 29
[표 6-4] ····· 30
[표 6-5] ····· 30
[표 7-1] ····· 31
[표 7-2] ····· 32

그 림 목 차

[그림 3-1] ····· 5
[그림 4-1] ····· 8
[그림 4-2] ····· 9
[그림 4-3] ····· 9
[그림 5-1] ····· 12
[그림 5-2] ····· 13
[그림 5-3] ····· 14
[그림 5-4] ····· 14

[그림 5-5] ····· 15
[그림 5-6] ····· 16
[그림 6-1] ····· 18
[그림 6-2] ····· 18
[그림 6-3] ····· 19
[그림 6-4] ····· 21
[그림 6-5] ····· 22
[그림 6-6] ····· 23
[그림 6-7] ····· 24
[그림 6-8] ····· 25
[그림 6-9] ····· 26
[그림 6-10] ····· 27
[그림 6-11] ····· 27
[그림 6-12] ····· 28
[그림 6-13] ····· 31
[그림 7-1] ····· 33
[그림 7-2] ····· 34
[그림 7-3] ····· 34
[그림 7-4] ····· 35
[그림 7-5] ····· 36

제 1 장 서론

오늘날 텍스트/음성 중심의 의사소통에서 이미지/영상 중심의 의사소통으로 주된 소통의 방법이 변화하고 있다. 모든 스마트폰에는 필수적으로 이미지 처리 가속 기술이 탑재 및 활용되고 있고, 이러한 기술은 스마트폰 시장 규모에서 경제적, 산업적으로 중요한 의미를 가진다. 각 모바일 제조사가 주력으로 판매하는 플래그십 모델의 경우 고성능의 카메라가 탑재되어 있으며, 이에 따라 이미지 처리 가속 기술의 중요성이 더욱 증대되고 있다. 대표적 모바일 기기인 스마트폰의 경우에는 Soc에 CPU, GPU 뿐 아니라 이미지 처리를 위한 ISP(Imagee Signal Processing) 및 DSP(Digital Signal Processing)를 함께 탑재하고 있어, 각 프로세서가 담당하는 이미지 처리 알고리즘에 따라 이미지를 처리한다.

하지만 이러한 ISP나 DSP와 같은 이미지 처리 프로세서에도 한계점이 존재하는데 프로세서를 구축하는 데 있어서 알고리즘 및 센서에 대한 복잡한 사전 지식, 프로세서 설계 비용, 새로운 알고리즘 개발을 위한 노동력 등이 요구된다는 점이다. 또한, 각 프로세서는 담당하고 있는 이미지 처리 알고리즘 외에 다른 새로운 이미지 처리 알고리즘은 처리할 수 없다. 이러한 한계점이 존재함에도 불구하고, ISP나 DSP를 대체하여 모바일 GPU를 통하여 이미지를 처리하는 실용적인 연구가 미미한 실정이다. 특히, ISP의 경우 Raw 이미지를 처리하는 데 있어 모바일 GPU를 활용한 사례를 찾기가 어렵다.

모바일 ISP의 기능을 대체하는 PyNET이라는 end-to-end 딥 러닝 모델이 존재하지만, 모바일 디바이스의 메모리 부족 문제와 지연 시간 문제 때문에 모바일에서는 성공적인 추론을 하기란 매우 어렵다. 이런 경우에는 모델의 특성과 모바일 디바이스의 특성을 고려하여 모델을 직접 구현하고 최적화하여야 한다. 예를 들어, 모델에서 주로 사용하는 연산과 성능 병목 지점 등을 파악하고 모바일 기기의 계산 성능과 메모리 대역폭, 메모리 용량 등의 자원이 어느 정도인지를 고려하여야 하며 모델을

경량화 하거나, 메모리 사용량을 줄이거나 지연 시간을 낮추는 등의 최적화가 적용되어야만 모바일 환경에서 성공적인 추론을 실행할 수 있다.

따라서 본 논문에서는 PyNET 모델을 모바일 GPU를 통해 추론하기 위한 다양한 메모리 관리 기법 및 OpenCL에 기반한 convolution 최적화 기법을 제시한다.

이어지는 2장에서는 관련 연구들에 대해서 알아보고. 3장과 4장에서는 본 논문에서 선정한 모바일 디바이스와 딥 러닝 모델을 분석한다. 5장과 6장에서는 논문에서 제시하는 메모리 관리 기법 및 OpenCL에 기반한 convolution 최적화에 관한 내용을 살펴보고, 7장에서 실험을 진행하여 마지막 8장에서 논문을 결론짓는다.

제 2 장 관련 연구

이 장에서는 모바일에서 딥 러닝 모델을 실행하는 데 있어서 발생하는 문제점들을 해결하고자 시도한 선행 연구들을 소개한다.

Deepmon[2]이라는 딥 러닝 모바일 프레임워크는 TensorFlow나 PyTorch로 개발된 비디오 스트리밍을 처리하는 딥 러닝 모델을 변환하여 모바일 GPU에 최적화하여 실행해 준다. 딥 러닝 모델의 최적화를 위하여 병목 지점인 convolution 레이어의 지연 시간과 메모리 사용량을 줄이는 것이 중요하다고 지적하고 있다. 제시하는 최적화 기법으로는 Caching, Convolution decomposition, Half-precision이 있다. Caching은 비디오 스트리밍에서 연속된 프레임이 존재할 때, 프레임 간 변화가 없는 같은 영역을 찾아 같은 영역은 convolution 연산을 하지 않고 이전 계산 값을 caching하여 재활용하는 방식이다. 또한, Convolution decomposition은 Tucker's algorithm을 이용해 convolution 레이어를 작은 레이어 여러 개로 분해하여 실행함으로써 convolution 레이어의 파라미터 수와 메모리 사용량을 줄였다. 또한, Half floating point precision을 사용하여 모델을 경량화하였다. Caching 기법은 비디오 스트리밍에만 적용할 수 있는 제한적인 기법이고, 실험에 사용한 가장 deep한 딥 러닝 모델의 convolution 레이어 수는 24개로, 본 논문에서 채택한 모델보다 현저히 적다. 본 논문에서는 훨씬 더 많은 convolution 레이어와 더 큰 filter 크기를 가지는 딥 러닝 모델을 선정하였으며 Implicit GEMM 알고리즘 및 Winograd 알고리즘을 사용하여 모바일에서 실행하기 위한 최적화 기법을 제시한다.

MCDNN[5]은 모바일 디바이스에서 딥 러닝 모델을 실행하기 위해서 모바일 디바이스의 자원 부족 문제를 해결하기 위하여 클라우드와 작업 분담을 하는 최적화 기법을 제안하고 있는데, 본 논문에서는 큰 딥 러닝 모델의 전체 추론 과정을 오직 모바일 CPU와 GPU만을 사용하여 추론한다.

Transfer Learning[4]은 모바일에서 딥 러닝 모델을 학습할 시, 모바

일 디바이스의 연산 성능 부족과 메모리 부족 현상을 해결하기 위하여 CPU와 GPU가 협업하여 학습할 수 있는 방식이다. 이 논문에서도 알 수 있듯이 모바일에서는 GPU만으로는 자원이 부족할 경우 CPU와 작업분담을 하여서 해결하여야 한다.

이 외에도, OpenCL 기반의 딥 러닝 모델의 추론 속도를 가속하는 연구[3]에서는 모바일 GPU에서 성능 저하를 보이는 애플리케이션들에 대해 프로파일링하고 최적화하는 기법들을 소개하였다. 기존 모바일 추론 프레임워크들이 적절한 워크-그룹 사이즈를 선정하지 못하여 GPU에 작업 분배가 제대로 되지 않아 성능 저하가 발생한다고 지적하였다. 그리하여 모델의 최적화를 위해서는 딥 러닝 모델과 모바일 하드웨어 특성을 분석하여 최적의 워크 그룹 사이즈를 선정해야 한다고 강조한다. 워크-그룹 사이즈 선택이 미치는 성능 변화를 비교하여 워크-그룹 사이즈 선택에 대한 가이드를 제시하였다. 이에 따라 본 논문에서도 실험적으로 최적의 워크-그룹 사이즈를 선택하였다.

제 3 장 모바일 디바이스 선정

타겟 모바일 디바이스는 Qualcomm Snapdragon 888 Mobile Hardware Development Kit로 선정하였다. Qualcomm Kryo 680 CPU와 Qualcomm Adreno 660 GPU를 탑재하고 있으며, 메모리는 12 GB이고 운영체제로는 Android 11이 제공된다.

CPU는 코어 8개의 빅-리틀 코어로 구성되어 있으며, Kryo 680 Prime(Cortex-X1) 1개, Kryo 680 Gold(Cortex-A78) 3개, Kryo 680 Silver(Cortex-A55) 4개로 이루어져 있다. GPU로는 Qualcomm Adreno 660 GPU가 탑재되어 있는데, 840 MHz에서 Peak Performance는 FP32 기준 1290 GFLOPS, FP16 기준 2580 GFLOPS이다. GPU 메모리는 CPU와 GPU가 공유하는 공유 메모리 시스템으로 크기는 4.3 GB이다. 이처럼, Discrete GPU에 비해 성능과 메모리 용량이 현저히 낮은 것을 알 수 있다.

Qualcomm Adreno 660 GPU	
Clock	840 MHz
Peak Performance (FP 32)	1290 GFLOPS
Peak Performance (FP 16)	2580 GFLOPS
ALUs	768
Memory	4.3 GB
OpenCL version	2.0
Max Memory Object size	1 GB

[표 3-1] Qualcomm Adreno 660 GPU 상세 스펙

메모리 제약사항으로, GPU 메모리 크기가 4.3 GB에 불과하고 최대 할당 가능한 메모리 오브젝트의 크기가 1 GB로 제한이 되어 있다. 따라서 모델을 구현할 때, 이러한 메모리 제약사항들을 위반하지 않도록 메모리 사용량을 최적화하여야 한다.

또한, Adreno 660 GPU에서 OpenCL을 사용할 때 참고해야 하는 사

항으로, private array를 사용하면 private register가 아닌 Local memory에 저장이 되고, 이마저도 Local memory 용량이 부족하면 Global memory에 저장이 되기 때문에 private array 사용을 지양하고 vector 사용을 지향할 것을 권고하고 있다. 따라서 이러한 모바일 디바이스의 특성에 맞는 최적화가 필요하다.

제 4 장 딥 러닝 모델 분석

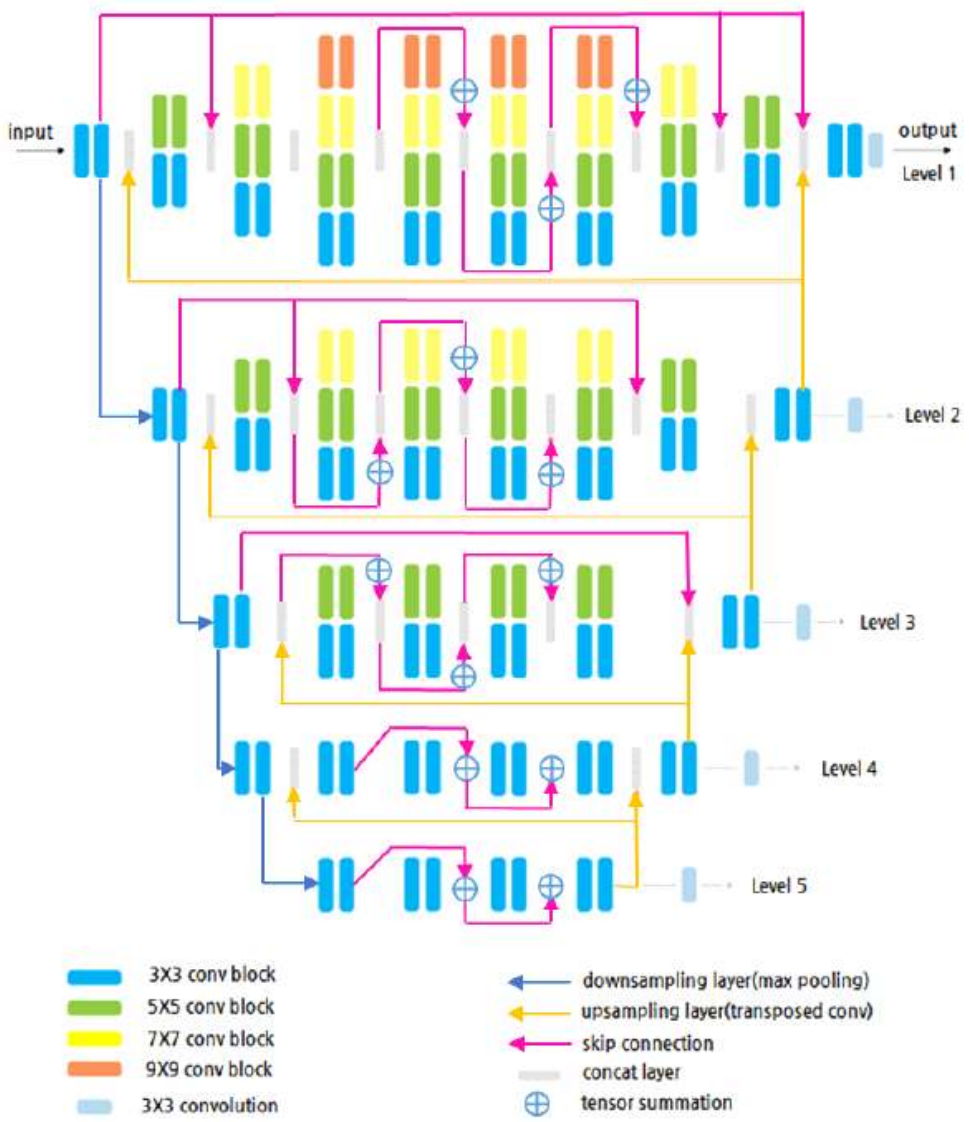
본 논문에서 채택한 딥 러닝 모델은 PyNET[1]이라는 CNN 구조의 딥 러닝 모델을 선정하였다.

제 1 절 PyNET 모델 개요

PyNET 모델은 모바일 ISP의 기능을 대체하는 end-to-end 딥 러닝 모델이다. ISP는 demosaicing, denoising, white balancing, color and contrast correction, demoireing 등의 여러 단계를 거쳐 이미지 복원을 담당하는 프로세서로서, Raw 이미지를 Full color 이미지로 복원하여 최종 사용자에게 제공한다. 이러한 복잡한 알고리즘들을 처리하기 위해 별도의 이미지 처리 프로세서인 ISP에 의존해야 하는 단점을 해결하기 위하여 제시한 딥 러닝 모델이다. ISP의 모든 이미지 처리 과정을 PyNET 모델의 추론 과정을 통해 처리하게 된다. 이로써 특정 모바일 기기의 센서나 복원 알고리즘에 대한 사전 지식 없이도 ISP보다 더 좋은 품질의 Full color 이미지를 복원하는 데 성공하였다.

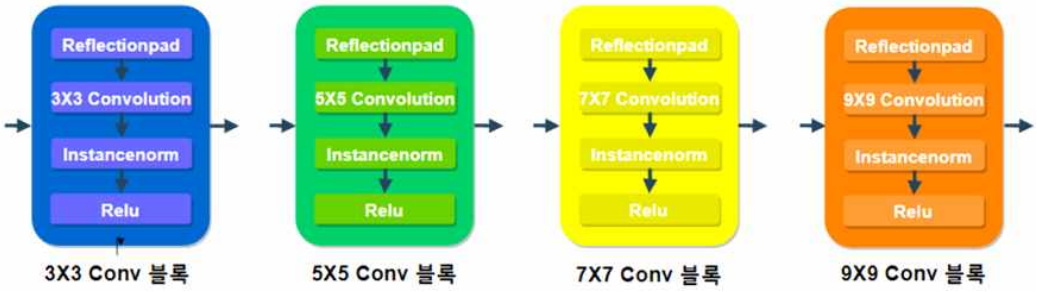
제 2 절 모델 구조 및 추론 과정

PyNET은 pyramidal CNN 구조의 딥 러닝 모델이며 구조는 아래 그림과 같다[1]. 네트워크의 전체적인 형상은 뒤집힌 피라미드 모양을 이루고 있으며, 5개의 레벨(레벨 5~레벨 1)로 나누어져 있다. 가장 하위 계층을 레벨 5로 지칭하며 가장 상위 계층은 레벨 1이다. 레벨에 따라 Raw 이미지를 다섯 가지 스케일로 처리하여 각 레벨의 입력으로 넣어주게 되는데, 레벨 1에서는 Raw 이미지의 원본을 들어가고, 하위 레벨로 이동할 때마다 이미지를 다운 샘플링하여 입력으로 들어간다.



[그림 4-1] PyNET 모델

대부분 Conv 블록([그림 4-2])들로 구성되어 있는데 블록 내부에는 ReflectionPad 레이어, Convolution 레이어, InstanceNorm 레이어, Relu 레이어를 포함하고 있다. Convolution 레이어의 filter 크기는 3x3, 5x5, 7x7, 9x9 총 4가지 종류가 존재한다.



[그림 4-2] Conv 블록의 내부 레이어 구성

하위 레벨에서는 다운 샘플링된 이미지가 들어감에 따라 이미지의 글로벌한 특징들을 캡쳐하게 되고, 상위 레벨로 갈수록 원본 이미지에 가까운 해상도를 갖기 때문에 이미지의 세부적인 특징들을 추출하게 된다.

추론은 가장 하위 레벨부터 시작되며, 각 레벨을 순차적으로 추론하여 최종적으로 Full color 이미지를 생성하게 된다. [그림 4-3]는 각 레벨마다 생성되는 이미지이다.

[그림 4-3] 레벨별 생성되는 출력 이미지 - 레벨5(left) ~ 레벨0(right)

제 3 절 PyTorch 기반 PyNET 애플리케이션 분석

오픈소스로 제공되는 PyTorch 기반 PyNET 애플리케이션을 프로파일링 하였다. 프로파일링은 PyTorch에서 제공하는 프로파일러를 사용하였다. 애플리케이션 실행 환경은 AMD EPYC 7502와 RTX 3090을 탑재한 워크스테이션에서 진행하였고, 추론에 사용한 입력 이미지는 47 MB (가로 3968, 세로 2976)인 Raw 이미지를 사용하였다. 최종적으로 141 MB(가로 3968, 세로 2976)의 Full color 이미지를 생성한다.

1. 주요 연산 프로파일링

모델이 이미지 한 장을 추론할 때, 연산이 불리는 횟수와 전체 실행시간에서 이 연산이 차지하는 실행 시간의 비율을 프로파일링 하였다. [표 4-1]는 이를 정리한 내용이다. 표에서 알 수 있듯이, 모델이 굉장히 많은 레이어들로 구성이 되어 있으며, 가장 많은 컴퓨팅 파워를 요하는 연산인 convolution 연산의 호출 횟수가 147회로 전체 실행시간의 77.1%를 차지하여, 주요 병목 지점이었다.

주요 연산	호출 횟수	연산 실행시간 비율 (%)
MaxPool	4	0.20
ReflectionPad	127	10.02
Upsample	9	0.60
InstanceNorm	124	7.01
Convolution	147	77.10
Relu	141	4.16
Sigmoid	6	0.04

[표 4-1] PyTorch 프레임워크 주요 연산 프로파일링

2. 추론 시간 프로파일링

해당 애플리케이션을 FP32를 사용하여 CPU만으로 이미지 한 장을 추론하는 경우(CPU, FP32), FP32를 사용하여 GPU만으로 이미지 한 장을 추론하는 경우(GPU, FP32), FP16를 사용하여 GPU만으로 이미지 한 장을 추론하는 경우(GPU, FP16), 총 세 가지 경우의 추론 시간을 측정하였다. [표 4-2]에서 확인할 수 있듯이, 워크스테이션 환경에서도 추론 시간이 많이 소요되는 것을 알 수 있다.

	이미지 한 장을 추론하는 데 소요되는 시간 (초)
CPU, FP32	49.8
GPU, FP32	5.4
GPU, FP16	2.7

[표 4-2] Pytorch 기반 PyNET 애플리케이션의 추론 시간

3. GPU 메모리 사용량 프로파일링

FP32를 사용하여 GPU만으로 추론하는 경우(GPU, FP32)에 GPU 메모리 사용량은 19.4 GB로 측정이 되었고, FP16을 사용하는 경우(GPU, FP16)는 그의 반인 9.7 GB가 측정이 되었다. PyTorch에서 이처럼 많은 메모리 사용량이 발생하는 이유는, 모델이 처리하는 이미지의 크기가 크고 PyTorch 내부에서 불필요한 메모리의 해제가 적절하게 이루어지지 않고, 메모리를 효율적으로 재활용하지 못하기 때문이다. 실험에 쓰인 모바일 디바이스의 GPU 메모리가 4.3 GB임을 고려하였을 때 PyTorch로 구현한 PyNET 애플리케이션은 주어진 입력에 대해서 모바일에서는 메모리 공간 부족으로 실행할 수 없음을 의미한다.

	GPU 메모리 사용량 (GB)
GPU, FP32	19.4 GB
GPU, FP16	9.7 GB

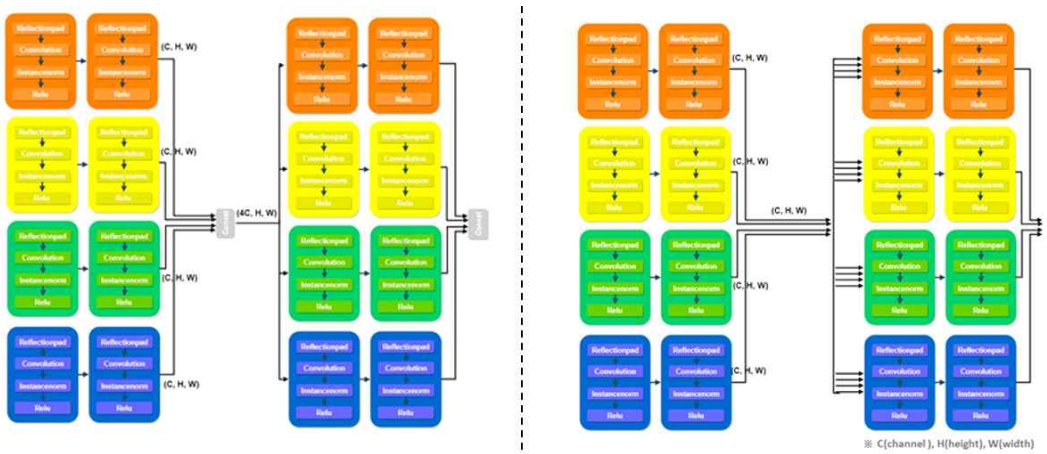
[표 4-3] Pytorch 기반 PyNET 애플리케이션의 GPU 메모리 사용량

제 5 장 메모리 관리 기법

모바일 디바이스에서 딥 러닝 모델을 실행하기 위해 다양한 메모리 관리 기법을 통해 메모리 사용량을 최적화하였다.

제 1 절 모델 구조 변경

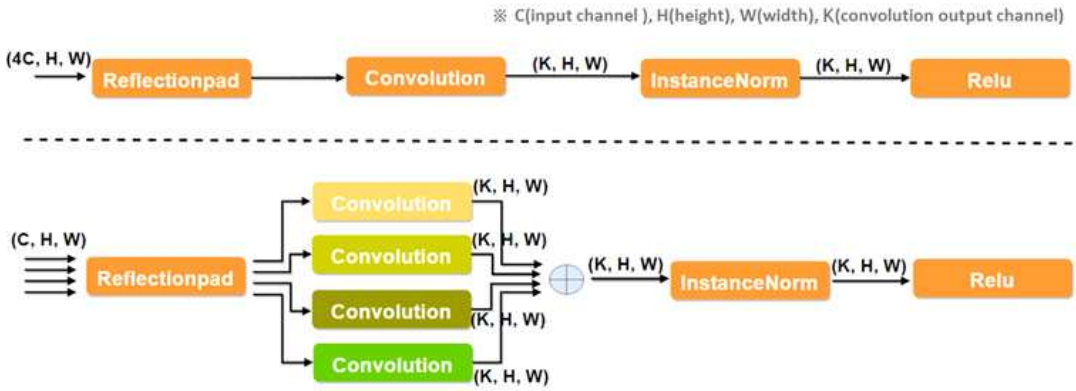
실험에 사용한 모바일 디바이스에서 할당할 수 있는 최대 GPU 메모리 오브젝트 크기는 1 GB이다. 하지만 [그림 5-1]에서 보듯이, 기존 모델 구조에서는 Concat 레이어가 channel-wise로 concatenation을 수행하게 되는데 이 때문에 메모리 오브젝트 크기가 최대 할당 가능한 크기인 1 GB를 초과하는 경우가 발생한다. 이를 방지하기 위하여 Concat 레이어를 제거하였다.



[그림 5-1] PyNET 모델 일부-Concat 레이어 제거 전(left)과 후(right)

이에 따라, Concat 레이어를 통해 하나의 입력으로 처리되던 데이터가 여러 개로 분할된다. 분할된 입력은 channel-wise로 나누어져 Reflectionpad 레이어와 Convolution 레이어를 차례로 거치게 되는데, Convolution 레이어의 경우는 가중치를 가지기 때문에 레이어를 입력 개

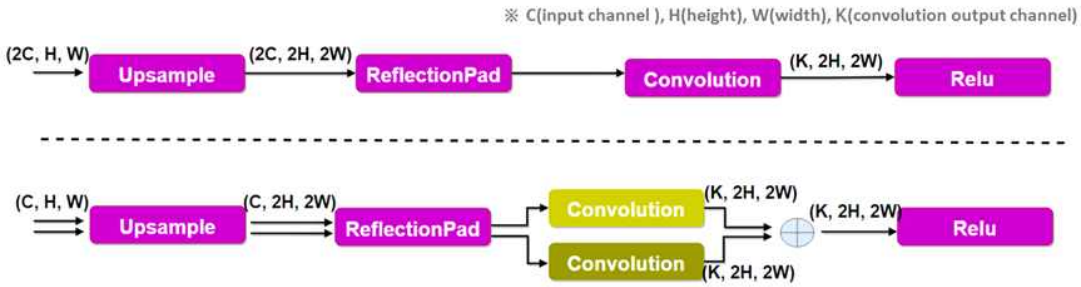
수에 맞게 channel-wise로 분할하고 모델의 가중치들을 재배열하여 저장하였다. Convolution 레이어를 거친 후에는 나눠진 출력들을 element-wise sum 연산을 통해 합치는 작업을 수행하여야 원래의 의미대로 전체 입력에 대한 convolution 연산을 한 결과와 같아진다. Convolution 레이어를 거친 후에는 출력 버퍼의 크기가 줄어들기 때문에 출력을 메모리 오브젝트에 저장할 수 있게 된다.



[그림 5-2] Concat 레이어 제거 전(top)과 후(botton)

제 2 절 입력 분할

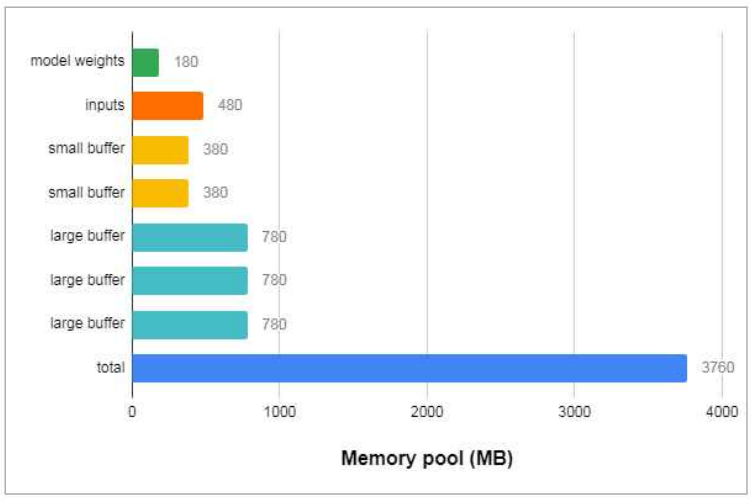
레이어의 출력 버퍼 크기가 최대 할당 가능한 메모리 오브젝트 크기를 초과하는 경우, 레이어의 입력 버퍼를 임의로 분할하여 처리하였다. 예를 들어, Upsample 레이어를 통과하면서 출력 버퍼 크기가 4배로 증가하게 되어 1 GB를 초과하는 경우 발생한다. 이를 해결하기 위하여 입력을 channel-wise로 임의로 분할 처리하여 출력 버퍼 크기가 1 GB를 초과하지 않도록 유지하였다.



[그림 5-3] UpsampleConv 블록의 입력 분할 전(top)과 후(bottom)

제 3 절 메모리 풀

실험에 쓰이는 모바일 디바이스의 GPU 메모리의 크기는 약 4.3 GB에 불과하다. 메모리 사용량이 4.3 GB를 초과하지 않기 위해 4 GB 크기의 메모리 풀을 설정하고 그 안에서 버퍼들을 할당하여 재활용하였다. 모델의 모든 레이어의 입력 및 출력 버퍼 크기를 분석한 결과, 모델에서 필요한 가장 큰 버퍼 크기는 780 MB 이고, 가장 작은 버퍼 크기는 380 MB이다. 이에 따라, 전체 메모리풀 4 GB에서 가중치 및 입력 데이터를 제외한 나머지는 큰 버퍼 3개와 작은 버퍼 2개를 할당하여 재활용하였다.

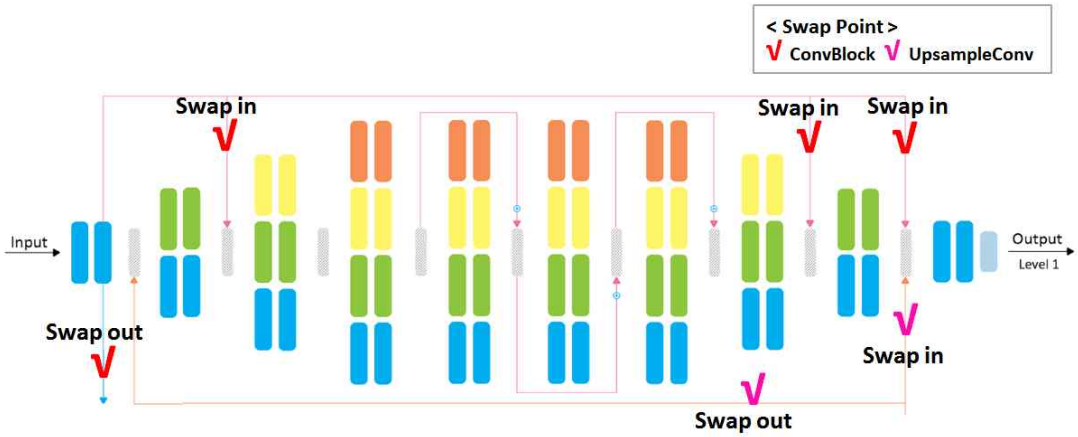


[그림 5-4] 메모리풀 세부 설정

제 4 절 메모리 스와핑

디바이스의 GPU 메모리는 4.3 GB인데 반해 모델에서 요구하는 메모리 사용량은 이를 초과하므로 이를 해결하기 위해서는 CPU 메모리를 GPU 메모리의 스왑(Swap) 메모리로 사용하여야 한다. 즉, 다시 필요한 데이터들은 스왑 아웃(Swap-out)하여 CPU에 저장해 놓고, 필요한 시점마다 GPU로 스왑 인(Swap-in) 하여 사용해야 한다.

PyNET의 레벨 1의 두번째 ConvBlock 출력 데이터의 경우, 추론의 가장 초기에 계산되지만, 가장 하위 레벨 5부터 상위 레벨 2까지 추론 과정에서는 쓰이지 않다가, 가장 상위 레벨인 레벨 1을 마지막으로 추론할 때 데이터가 다시 쓰이게 된다. 때문에 이를 따로 GPU 메모리에 저장하지 않고, 데이터를 스왑 아웃 한 후 다시 필요할 때마다 스왑 인 하여 사용한다. [그림 5-5]은 스왑 아웃, 스왑 인하는 지점을 표시한 그림이다.

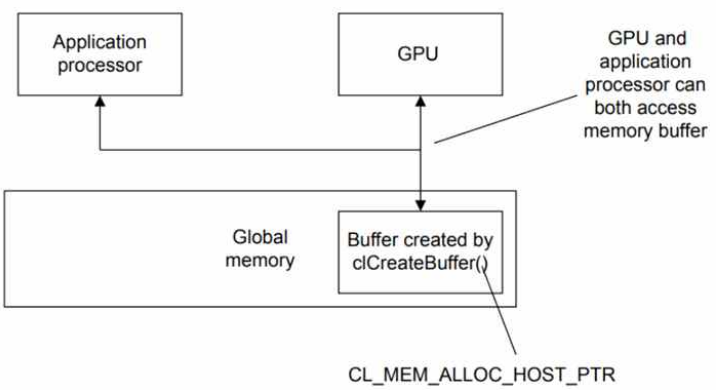


[그림 5-5] 레벨 1에서 메모리 스와핑이 필요한 지점

제 5 절 Zero copy

CPU와 GPU 사이 버퍼를 이동할 때 사용되는 OpenCL API인 clEnqueueReadBuffer()와 clEnqeueWriteBuffer()는 내부적으로 버퍼를

옮기는 과정에서 memory copy가 발생하게 된다. 하지만, 모바일은 CPU와 GPU가 메모리를 공유하는 공유 메모리 시스템이기 때문에, 아래 그림[13]과 같이 GPU에서 할당된 메모리를 memory copy 없이 호스트에서도 접근할 수 있다. GPU에서 clCreateBuffer() API를 사용하여 버퍼를 생성할 때 CL_MEM_ALLOC_HOST_PTR 옵션을 주면 호스트에서도 접근할 수 있는 포인터를 얻을 수 있다.



[그림 5-6] Memory buffer created by clCreateBuffer() (CL_MEM_ALLOC_HOST_PTR)

제 6 절 Half-precision floating-point

모델의 입력 이미지는 각 픽셀이 10 bit의 integer형 데이터를 담고 있다. 이를 0~1사이 값인 Single-precision floating-point로 형 변환한 후, 추론 과정을 거친 후, 최종적으로는 각 픽셀은 0~255 값을 갖는 8-bit integer 형의 png 파일로 저장된다. 모델 경량화를 위해 추론 시, Single-precision 대신 Half-precision을 사용하였다. Half-precision을 사용할 경우 계산 시간이 줄어들고 메모리 입출력 크기가 절반으로 줄어들기 때문에 성능 향상과 메모리 사용량을 절약할 수 있다.

제 6 장 Convolution 최적화

PyNET 모델은 CNN의 구조의 모델이기 때문에, 연산의 77% 이상이 convolution 연산으로 이루어져 있다. 따라서, convolution 연산의 성능이 모델의 성능에 큰 영향을 미친다. 어떠한 최적화도 적용하지 않은 Direct Convolution을 적용할 경우, 모바일에서는 많은 지연 시간이 발생하기 때문에 모바일에서 사용하기에 적합하지 않다. 따라서 OpenCL 기반의 convolution 성능 최적화를 위해 Implicit GEMM Convolution과 Winograd Convolution을 구현하였다. PyNET에 등장하는 filter 크기 3x3, 5x5 convolution에는 Winograd Convolution을 적용하고, Winograd Convolution의 경우 filter 크기가 커질수록 accuracy에 영향이 있고 메모리 사용량이 커지기 때문에 filter 크기 7x7, 9x9 convolution에는 Implicit GEMM Convolution을 적용하였다.

제 1 절 Direct convolution

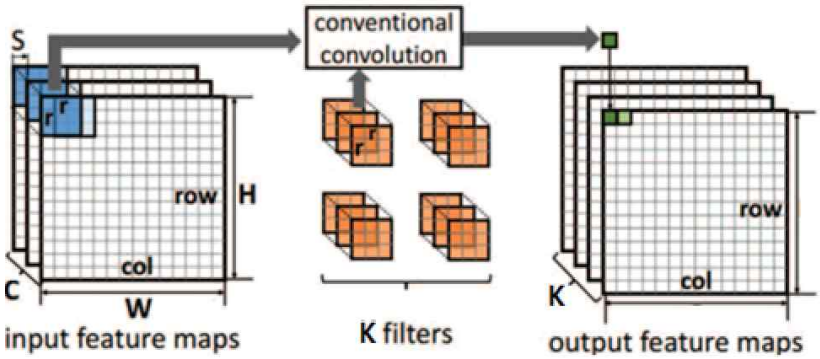
어떠한 최적화도 적용하지 않은 Direct convolution이다. [그림 6-1]와 같이[8], $r \times r$ 크기의 filter가 input channel인 C개만큼 존재하고, 이러한 filter 묶음이 output channel인 K개만큼 존재한다. 이 filter를 가지고 크기 (C, H, W)인 i번째 이미지에 convolution을 적용한다. (u, v)에 위치한 filter element는 $G_{k,c,u,v}$, (x, y)에 위치한 image element는 $D_{i,c,x,y}$로 표기할 수 있고, convolution 연산의 하나의 output element인 $Y_{i,k,x,y}$는 아래와 같이 구할 수 있다.

$$Y_{i,k,x,y} = \sum_{c=1}^{C} \sum_{v=1}^{R} \sum_{u=1}^{S} D_{i,c,x+u,y+v} G_{k,c,u,v}$$

$r \times r$ 크기의 filter tile(G)과 filter가 적용되는 input tile(D)의 쌍으로

표현하면 아래와 같고[8], *는 합성곱을 의미한다.

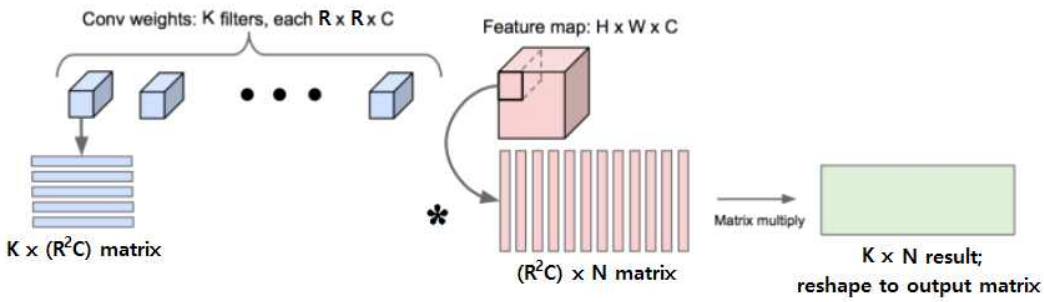
$$Y_{i,k} = \sum_{c=1}^{C} D_{i,c} * G_{k,c}$$



[그림 6-1] Direct Convolution

제 2 절 Implicit GEMM Convolution

1. Implicit GEMM Convolution 알고리즘



[그림 6-2] im2col 방식의 convolution

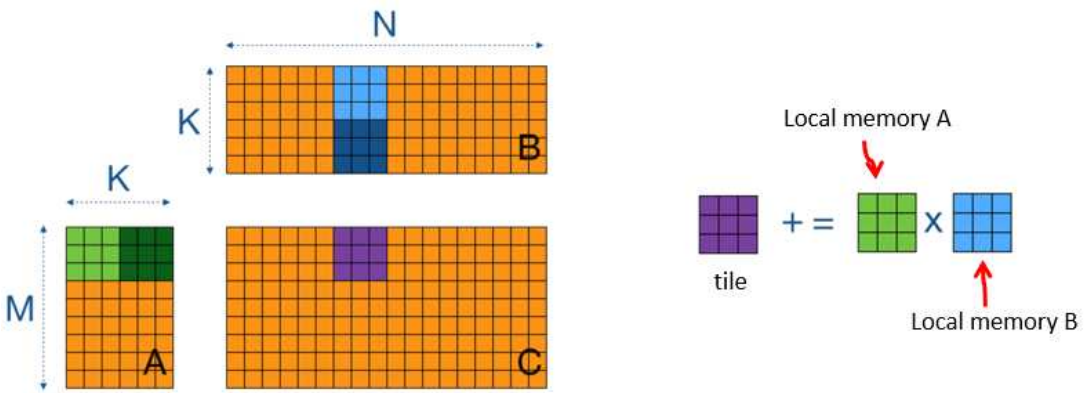
Implicit GEMM Convolution은 Direct Convolution의 변형으로, 위 그림과 같이[9] im2col 방식으로 이미지를 변환하여 GEMM 형태로 convolution을 수행한다. 다만, 이미지를 변환하여 별도의 메모리 공간에

저장하지 않고 on-the-fly 방식으로 index를 계산하여 convolution을 수행하기 때문에, 추가적인 메모리가 필요하지 않다는 이점이 있다.

2. Implicit GEMM Convolution 커널 최적화

① Tiled matrix multiplication 적용

Matrix multiplication 수행 시, 아래 그림[10]처럼 Tiled matrix multiplication을 적용하여 Global memory에서 tile에 해당하는 데이터를 Local memory로 이동하여 matrix multiplication을 처리하게 된다. Local memory 내부 데이터 재활용성을 높임으로써 성능이 향상된다.



[그림 6-3] Tiled matrix multiplication

② Memory Coalescing

Tiled matrix multiplication 수행 시, Global memory에서 tile에 해당하는 데이터를 load하거나 결과 output tile을 Global Memory에 store할 때, coalesced access를 수행한다.

③ 최적의 tile 크기와 워크-그룹 크기 선정

Tiled matrix multiplication 수행 시, 최적의 tile 크기(TILE_M, TILE_N, TILE_K)는 TILE_M은 32, TILE_N은 32, TILE_K는 16이다. 한 tile을 한 워크-그룹이 처리하게 되는데 최적의 워크-그룹 크기는 64

이다.

3. Implicit GEMM Convolution 커널

한 스레드가 $(K, R^2 \times C)$ matrix와 $(R^2 \times C, OH \times OW)$ matrix의 Tiled matrix multiplication을 수행한다($\widetilde{M} = K$, $\widetilde{N} = OH \times OW$, $\widetilde{K} = R^2 \times C$). 총 output tile 개수는 $\widetilde{M}$/TILE_M×$\widetilde{N}$/TILE_N이며, 총 스레드 개수(GWS)는 총 output tile의 개수×워크-그룹 크기(LWS)이다.

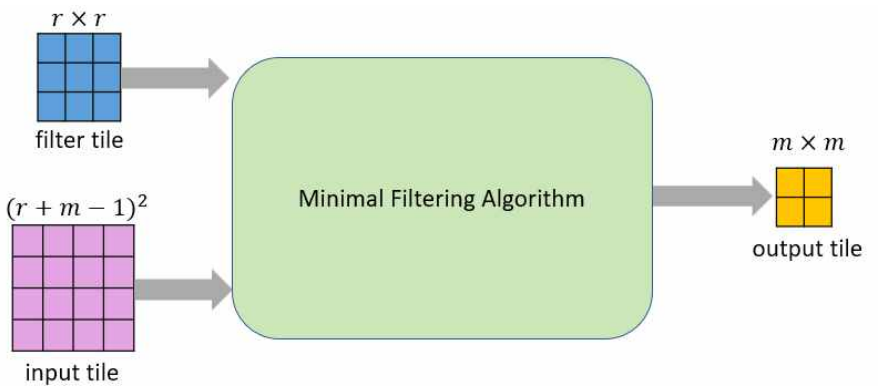
Implicit GEMM Convolution 커널	
커널이 하는 일	Tiled matrix multiplication $(\widetilde{M}, \widetilde{N}, \widetilde{K})$
워크-그룹이 하는 일	한 워크그룹이 하나의 output tile처리
GWS	($\widetilde{M}$/TILE_M× $\widetilde{N}$/TILE_N×LWS 1, 1)
LWS	(64, 1, 1)
TILE_M	32
TILE_N	32
TILE_K	16

[표 6-1] Implicit GEMM Convolution 커널

제 3 절 Winograd convolution

1. Minimal Filtering 알고리즘

Winograd Convolution은 Minimal Filtering 알고리즘을 활용하여 convolution을 수행한다. 2차원 Minimal Filtering 알고리즘[6]은 $F(m \times m, r \times r)$로 나타낼 수 있으며, 크기 $r \times r$인 2차원 filter tile을 가지고 크기 $(m+r-1) \times (m+r-1)$인 input tile에 적용하여 크기 $m \times m$인 output을 계산하게 된다.



[그림 6-4] Minimal filtering 알고리즘인 $F(m \times m, r \times r)$의 입력과 출력

이를 계산하는데 필요한 최소 곱셈수($\mu(F(m \times m, r \times r))$는 $(m+r-1)^2$으로 Minimum Filtering 알고리즘을 사용하지 않은 기존 방식의 필요한 최소 곱셈수 $(m \times m \times r \times r)$에 비해 필요한 곱셈 수를 최소한으로 줄여 성능 이득을 볼 수 있게 된다.

예를 들어, 2차원 Minimum Filtering 알고리즘인 $F(2 \times 2, 3 \times 3)$은 다음과 같이 계산된다. 3x3 크기의 filter tile을 가지고 2x2 크기의 output tile을 계산하기 위해 필요한 곱셈 수는 $\mu(F(m \times m, r \times r))$ = $(m+r-1)^2$ = $(2+3-1)^2$ = 16으로, 기존 곱셈 수인 36 대비 2.25배의 성능 향상이 있다.

2차원 minimum filtering 알고리즘 $F(m \times m, r \times r)$은 matrix 형태로 아래와 같은 수식으로 구할 수 있다.

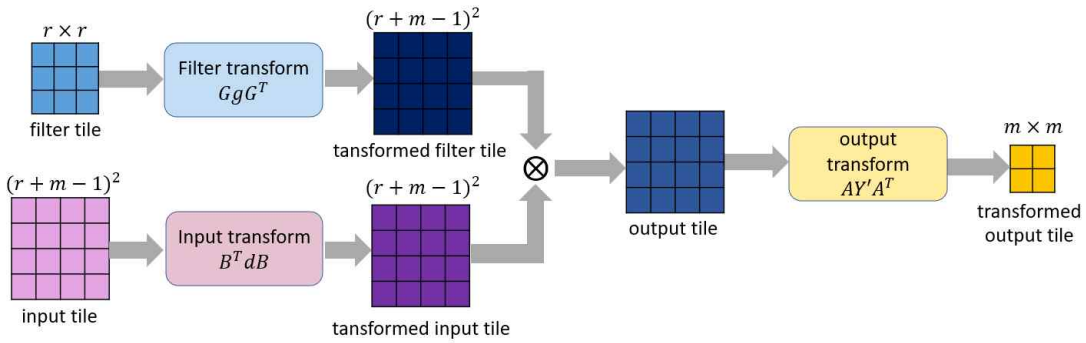
$$Y = A^T \left[[GgG^T] \odot [B^T dB] \right] A$$

g는 $r \times r$ 크기의 filter이고, d는 크기 $(m+r-1)^2$의 input tile, Y는 크기 $m \times m$인 output tile, G, B, A matrix는 상수행렬로 구성된 transformation matrix에 해당한다.

수식은 4단계의 변환 과정(transform)으로 구성된다(① input

transform, ② filter transform, ③ matrix multiplication, ④ output transform).

① input tile인 d를 주어진 input transformation matrix(B, B^T)를 통해 변환하는 과정(input transform:$B^T dB$), ② filter tile인 g를 filter transformation matrix(G, G^T)를 통해 변환하는 과정(filter transform: GgG^T), ③ 변환된 input tile과 filter tile을 element-wise multiplication 하여 output tile을 구하는 과정, ④ output tile을 output transformation matrix(A, A^T)를 통해 변환하는 과정(output transform:$AY'A^T$)으로 구성된다.



[그림 6-5] Minimal filtering 알고리즘 $F(m \times m, r \times r)$ 과정

2. Winograd convolution 알고리즘

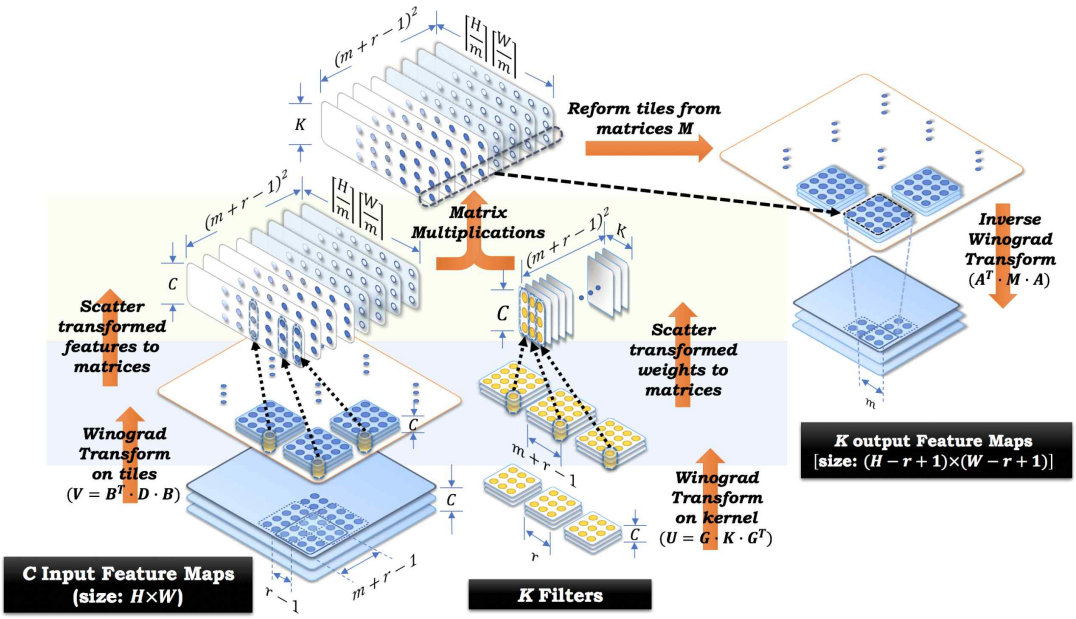
Minimal filtering 알고리즘은 하나의 filter tile과 input tile에 대하여 output tile을 생성하므로, 이를 활용하여 크기 (C, H, W)인 이미지와 크기 (K, C, r, r)인 filter에 대하여 크기 (K, H', W')인 output을 생성하는 Winograd Convolution을 수행할 수 있다. 이를 그림으로 나타내면 아래 그림과 같다[7].

우선, filter는 tile 원소가 $X(=(r+m-1)^2)$개인 $(K \times C)$개의 filter tile들로 이루어져 있으며, filter tile들에 filter transform(GgG^T)을 수행하여 변환된 filter tile들을 scatter하여 filter tiles에 저장한다.

input은 input tile끼리 가로 방향 또는 세로 방향으로 $(r-1)$개의 원소

가 중첩되어, 원소 m씩 떨어진 input tile들로 구성된다. input tile은 원소가 $X(=(r+m-1)^2)$개인 $(C\times P\times Q)$개의 input tile들로 이루어져 있으며(P=세로 tile 개수=⌈H/m⌉, Q=가로 tile 개수=⌈W/m⌉), input transform(B^TdB)을 수행하여 변환된 input tile들을 scatter하여 input tiles에 저장한다. 이때, scatter하여 저장하는 작업은 element-wise 연산을 matrix multiplication 연산으로 대체하여 GPU에서 더 효율적인 구현을 하기 위함이다. filter tiles에 저장된 X개의 (K, C) matrix와 input tiles에 저장된 X개의 $(C, P\times Q)$ matrix에 대해서 각 X번의 (K, C) matrix와 $(C, P\times Q)$ matrix의 matrix-multiplication을 수행한다. 이렇게 되면 최종적으로 X개의 $(K, P\times Q)$ matrix가 생성되는데 이를 output tiles에 저장한 후, 각 output tile들을 output transform($A^TY'A$)을 통해 최종 변환하여 output(K, H', W')에 저장한다.

PyNET에서는, Filter 크기 3x3 convolution에는 $F(4\times4, 3\times3)$를 Filter 크기 5x5 convolution에는 $F(4\times4, 5\times5)$ 활용하였다.



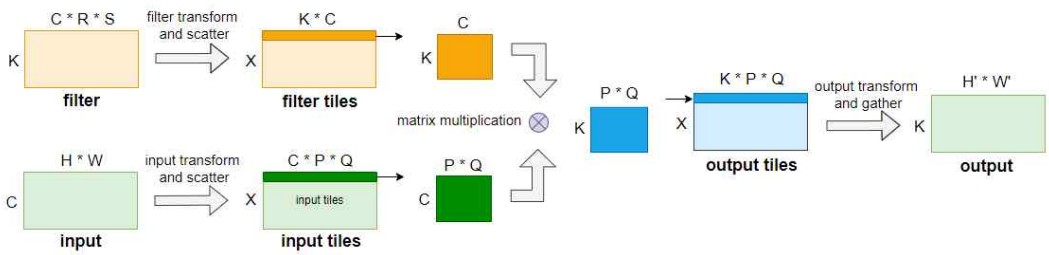
[그림 6-6] Winograd convolution

3. Winograd Convolution 메모리 사용량 최적화

[그림 6-8]에서 보듯이, Winograd Convolution의 각 중간 결과물(input tiles, filter tiles, output tiles)를 저장할 메모리 오브젝트는 아래와 같다(X = $(m+r-1)^2$. P = ⌈ H/m ⌉, Q = ⌈ W/m ⌉).

- filter tile들을 저장할 filter tiles 형태 : (X, K, C)
- input tile들을 저장할 input tiles 형태 : (X, C, P, Q)
- output tile들을 저장할 output tiles 형태 : (X, K, P, Q)

filter tiles에 저장된 X개의 (K, C) matrix와 input tiles에 저장된 X개의 $(C, P\times Q)$ matrix에 대해서 각 X번의 (K, C) matrix와 $(C, P\times Q)$ matrix의 matrix-multiplication을 수행하게 된다. 이렇게 되면 최종적으로 X개의 $(K, P\times Q)$ matrix가 나오게 되는데 이를 output tiles에 저장한 후, 최종적으로 output transform을 통해 변환하여 output에 저장한다.



[그림 6-7] Winograd - 기본 처리

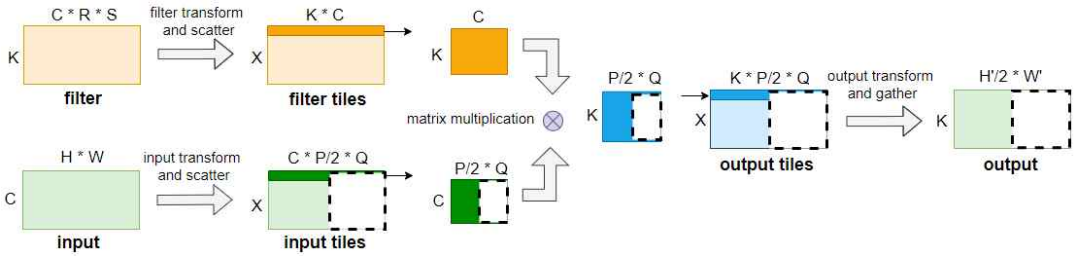
문제는 Winograd Convolution을 수행하기 위해 필요한 추가 메모리 공간인 input tiles, output tiles들이 메모리 오브젝트 크기 제한을 초과하는 경우가 발생하여 모바일 디바이스의 제약사항을 위반하게 된다. 따라서 버퍼를 분할하여 Winograd Convolution을 실행해야 하는데, P(세로 tile 개수)를 기준으로 분할 처리하는 방법과 C(channel)를 기준으로 분할 처리하는 방법 2가지가 있다.

① P(세로 tile 개수)를 기준으로 분할 처리하는 방법

P를 기준으로 반으로 분할 처리하면, 필요한 메모리 오브젝트는 다음과 같이 변경된다.

- filter tile들을 저장할 filter tiles 형태 : (X, K, C)
- input tile들을 저장할 input tiles 형태 : $(X, C, P/2, Q)$
- output tile들을 저장할 output tiles 형태 : $(X, K, P/2, Q)$

filter tiles에 저장된 X개의 (K, C) matrix와 input tiles에 저장된 X개의 $(C, (P/2) \times Q)$ matrix에 대해서 각 X번의 (K, C) matrix와 $(C, (P/2) \times Q)$ matrix의 matrix-multiplication을 수행한다. 이렇게 되면 최종적으로 X개의 $(K, (P/2) \times Q)$ matrix가 생성되는데, 이를 output tiles에 저장한 후, output tiles에 저장된 각각의 output tile들을 output transform을 통해 변환하여 output에 해당하는 위치에 저장한다. 이는 inpu tiles의 절반에 해당하는 부분만 계산된 결과이므로, 나머지 부분들에 대해서도 같은 방식으로 반복하여 처리하여 최종 output을 계산한다. 이로써, 기존 input tiles와 output tiles에 해당하는 메모리 오브젝트 크기의 절반만으로도 Winograd Convolution 연산이 가능해진다.



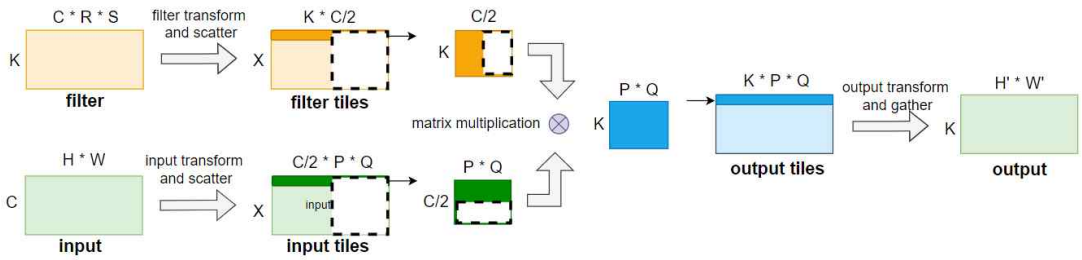
[그림 6-8] Winograd - 세로 tile 개수(P) 기준으로 분할 처리

② C(channel)을 분할하는 방법

C를 기준으로 반으로 분할 처리하면, 필요한 메모리 오브젝트는 다음과 같이 변경된다.

- filter tile들을 저장할 filter tiles 형태 : $(X, K, C/2)$
- input tile들을 저장할 input tiles 형태 : $(X, C/2, P, Q)$
- output tile들을 저장할 output tiles 형태 : (X, K, P, Q)

filter tiles에 저장된 X개의 $(K, C/2)$ matrix와 input tiles에 저장된 X개의 $(C/2, P\times Q)$ matrix에 대해서 각 X번의 $(K, C/2)$ matrix와 $(C/2, P\times Q)$ matrix의 matrix-multiplication을 수행하게 된다. 이렇게 되면 최종적으로 X개의 $(K, P\times Q)$ matrix가 나오게 되는데, 이를 output tiles에 저장한다. 이는 input tiles와 filter tiles의 절반에 해당하는 부분만 계산된 결과이므로, 나머지 부분들에 대해서도 동일한 처리를 하여 구해지는 X개의 $(K, P\times Q)$를 output tiles에 누적합 한다. 최종적으로 output tiles에 저장된 각각의 output tile들을 output transform을 통해 변환하여 output에 저장한다. 이로써, 기존 input tiles와 filter tiles에 해당하는 메모리 오브젝트 크기의 절반만으로도 Winograd Convolution 연산이 가능해진다.



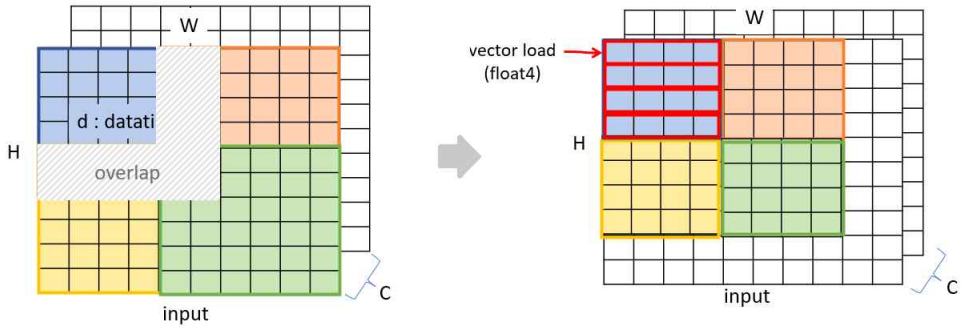
[그림 6-9] Winograd - input channel(C) 기준으로 분할 처리

4. Winograd convolution 커널 최적화

Matrix 4단계의 변환 과정(transform)에 따라, input transform 커널, filter transform 커널, matrix multiplication 커널, output transform 커널로 구현하고 최적화하였다.

① Global Memory 중복 접근 제거 및 Local Memory 활용

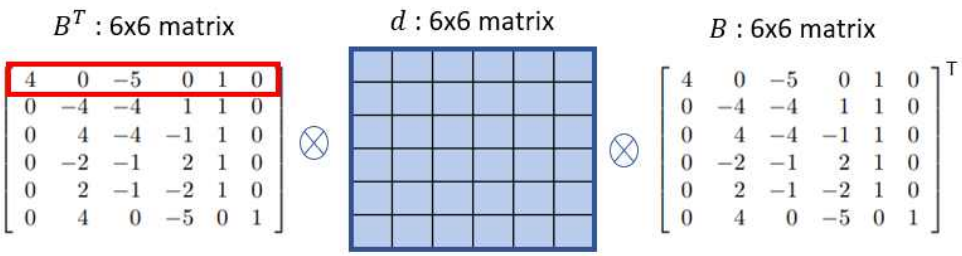
input transform 커널의 경우, 한 스레드가 한 input tile에 해당하는 부분을 Global Memory에서 읽어오게 되는데, input tile 간에 중첩되는 부분 때문에, 스레드 간에 중복해서 읽어오는 부분이 존재하게 된다. 이를 제거하기 위하여, 한 스레드는 중복되는 부분을 제외한 부분만 vector I/O를 이용하여 Global Memory에서 읽어 Local Memory에 저장하여 활용하도록 하였다.



[그림 6-10] Winograd - Global Memory 중복 접근 제거

② Register 사용량 감소

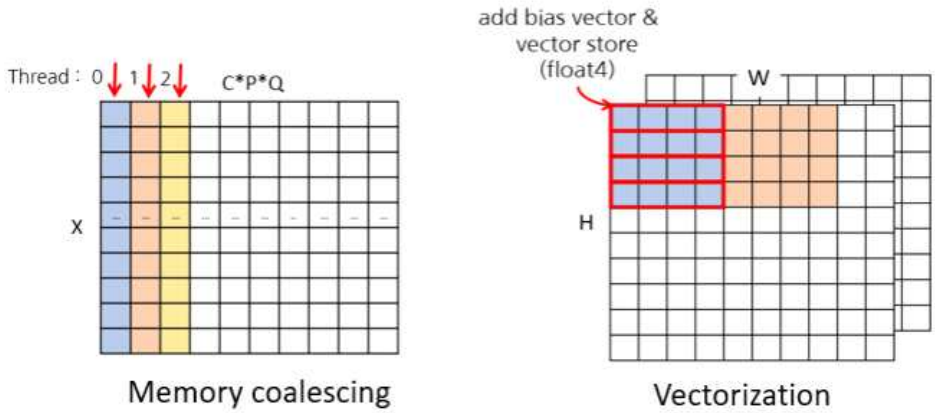
Tile을 변환할 때, 상수 행렬을 앞, 뒤로 곱해주는 과정을 거치게 되는데($B^T dB$, GgG^T, $A^T Y' A$), 이때, 앞의 상수행렬을 곱한 결과와 뒤의 상수행렬을 곱한 중간 결과들을 저장할 private register가 다수 필요하게 되어 register spilling이 발생하게 된다. 따라서 행렬 곱을 할 때, 한 행씩 분할하여 처리하여 register 사용량을 감소하였다.



[그림 6-11] Winograd - Register 사용량 감소

③ Memory coalescing, Vectorization 적용

Global memory에서 input tile과 filter tile을 load하거나 Global memory로 output tile을 store할 때, Memory coalescing 또는 Vector I/O를 적용하였다. 또한 bias를 더할 때 vector computation을 적용하였다.



[그림 6-12] Winograd - Memory coalescing, Vectorization

④ Tiled matrix multiplication 적용

Matrix multiplication 수행 시, Tiled matrix multiplication을 적용하였다.

⑤ 최적의 tile 크기와 워크-그룹 크기 선정

Tiled matrix multiplication 수행 시, 최적의 tile 크기(TILE_M, TILE_N, TILE_K)를 선정하였다(TILE_M=32, TILE_N=32, TILE_K=16). 한 tile을 한 워크-그룹이 처리하게 되는데 최적의 워크-그룹 크기는 64이다.

5. Winograd convolution 커널

① Input transform 커널

한 스레드가 하나의 input tile을 담당하여 input transformation($B^T d B$)을 수행한 후, scatter하여 input tiles(output)에 저장한다.

Input transform 커널	
커널이 하는 일	input transformation ($B^T dB$)
커널 입력	Input
커널 출력	input transformation 적용한 input tiles
스레드가 하는 일	한 스레드가 하나의 tile 처리
GWS	(Q, P, C)
LWS	(16, 16, 1)

[표 6-2] Input transform 커널

② Filter transform 커널

한 스레드가 하나의 filter tile을 담당하여 filter transformation(GgG^T)을 수행한 후, scatter하여 input tiles(output)에 저장한다.

Filter transform 커널	
커널이 하는 일	filter transformation (GgG^T)
커널 입력	filter
커널 출력	filter transformation 적용한 filter tiles
스레드가 하는 일	한 스레드가 하나의 filter tile 처리
GWS	(K×C, 1, 1)
LWS	(128, 1, 1)

[표 6-3] Filter transform 커널

③ GEMM 커널

한 스레드가 (K, C) matrix와 $(C, P \times Q)$ matrix의 tiled matrix multiplication을 수행한다($\widetilde{M} = K$, $\widetilde{N} = P \times Q$, $\widetilde{K} = C$). 총 output tile 개수는 X×$\widetilde{M}$/TILE_M×$\widetilde{N}$/TILE_N이며, 총 스레드 개수(GWS)는 총 output tile의 개수 × 워크-그룹 크기(LWS)이다.

GEMM 커널	
커널이 하는 일	Tiled matrix multiplication
커널 입력	input tiles, filter tiles
커널 출력	output tiles
워크그룹이 하는 일	한 워크그룹이 하나의 output tile처리
GWS	$(X\times \widetilde{M}/TILE_M\times \widetilde{N}/TILE_N\times LWS, 1, 1)$
LWS	(64, 1, 1)
TILE_M	32
TILE_N	32
TILE_K	16

[표 6-4] GEMM 커널

④ Output transform 커널

한 스레드가 하나의 output tile을 gather한 후, output transform $(A^{T}Y'A)$한 결과를 output에 저장한다.

Output transform 커널	
커널이 하는 일	output transformation $(AY'A^{T})$
커널 입력	output tiles
커널 출력	output
스레드가 하는 일	한 스레드가 하나의 output tile처리
GWS	$(K\times P\times Q, 1, 1)$
LWS	(128, 1, 1)

[표 6-5] Output transform 커널

제 7 장 실험 결과 및 분석

본 장에서는 모바일 실험 환경과 convolution의 성능 테스트를 위해 정의한 Convolution Configuration Set을 소개하고 모바일 환경에서 실제 실험한 결과를 분석한다. 실험에 사용한 입력 이미지로는 47 MB(가로 3968, 세로 2976)인 Raw 이미지를 사용하였다.

제 1 절 모바일 실험 환경

Device	Qualcomm Snapdragon 888 Mobile Hardware Development Kit
CPU	Qualcomm Kyro 680 CPU (8 cores)
Memory (shared)	12 GB
GPU	Qualcomm Adreno 660 GPU @ 840 MHz
OS	Android

[표 7-1] 모바일 실험 환경

제 2 절 Convolution Configurations Set

PyNET에서 추론 시 실제 사용되는 convolution 레이어의 모든 configuration(C=input channel, H=input height, W=input width, K=output channel)들을 리스트 업하여 Convolution Configuration Set을 정리하였다. 총 35가지 configuration이 등장하며 convolution 알고리즘별 성능 테스트에 사용하였다.

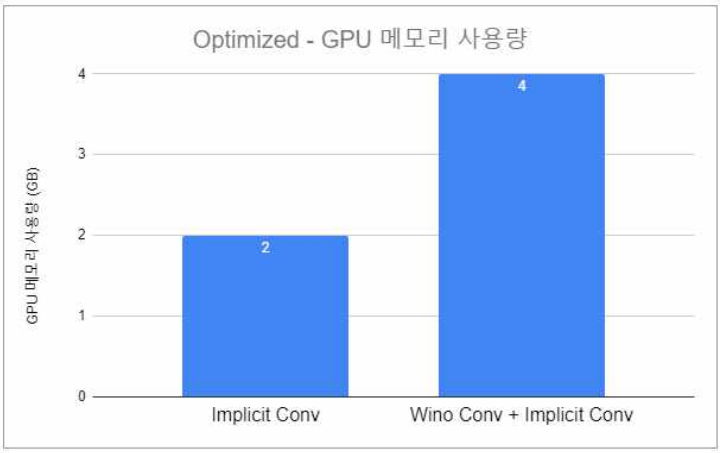
no	C	H	W	Filter size	K
0	4	1440	1984	3x3	32
1	32	1440	1984	3x3	32
2	32	720	992	3x3	64
3	64	720	992	3x3	64
4	64	360	496	3x3	128
5	128	360	496	3x3	128
6	128	180	248	3x3	256
7	256	180	248	3x3	256
8	256	90	124	3x3	512
9	512	90	124	3x3	512
10	512	180	248	3x3	256
11	512	90	124	3x3	3
12	256	360	496	3x3	128
13	256	180	248	3x3	3
14	256	360	496	5x5	128
15	128	360	496	5x5	128
16	512	360	496	3x3	128
17	128	720	992	3x3	64
18	128	360	496	3x3	3
19	128	720	992	5x5	64
20	64	720	992	5x5	64
21	192	720	992	3x3	64
22	192	720	992	5x5	64
23	192	720	992	7x7	64
24	64	720	992	7x7	64
25	256	720	992	3x3	64
26	256	720	992	5x5	64
27	64	1440	1984	3x3	32
28	64	720	992	3x3	3
29	32	1440	1984	5x5	32
30	32	1440	1984	7x7	32
31	32	1440	1984	9x9	32
32	16	2880	3968	3x3	16
33	32	1440	1984	3x3	3
34	16	2880	3968	3x3	3

[표 7-2] Convolution Configuration Set

제 3 절 실험 결과

1. 최적화 모델의 GPU 메모리 사용량 측정

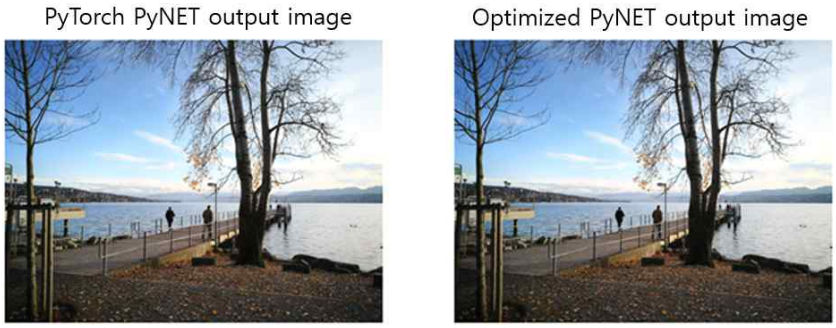
최적화 모델의 GPU 메모리 사용량은 Implicit GEMM convolution 사용 시 2 GB, Winorad Convolution과 Implicit GEMM convolution를 함께 사용 시 4 GB(Winorad Convolution에 2 GB 소요)로 측정되어 모바일에서 실행 가능하도록 최적화하였다.



[그림 7-1] 최적화 모델의 GPU 메모리 사용량

2. 결과 이미지 PSNR 값 측정

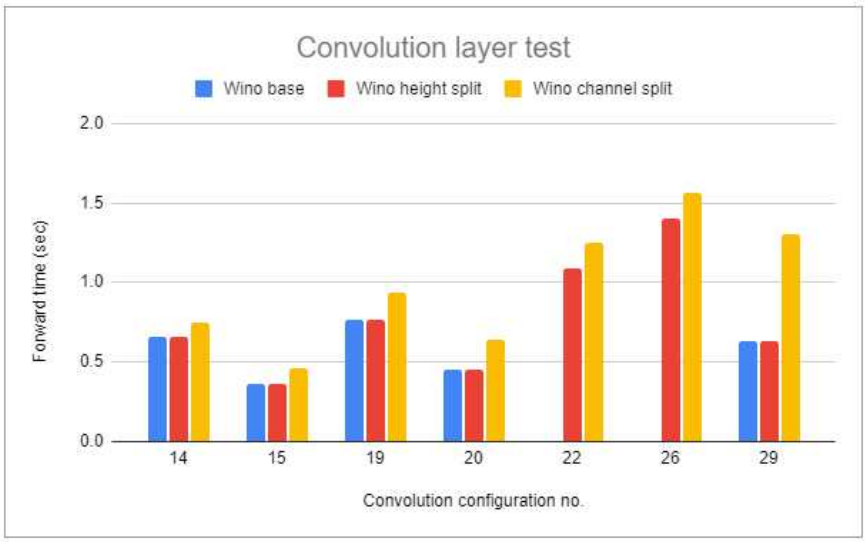
결과 이미지 검증을 위해, PyTorch 기반의 PyNET 애플리케이션 결과 이미지와 최적화 모델의 결과 이미지의 PSNR 값을 비교하였다. 비교 결과, PSNR 값은 36 데시벨로 통상 30 데시벨이 넘으면 육안으로 비교가 불가하기 때문에, 오차 범위 내의 유의미한 결과 이미지를 생성하는 것을 검증하였다.



[그림 7-2] PyTorch 결과 이미지 vs Optimized 결과 이미지

3. Winograd Convolution 분할 방식에 따른 성능 측정

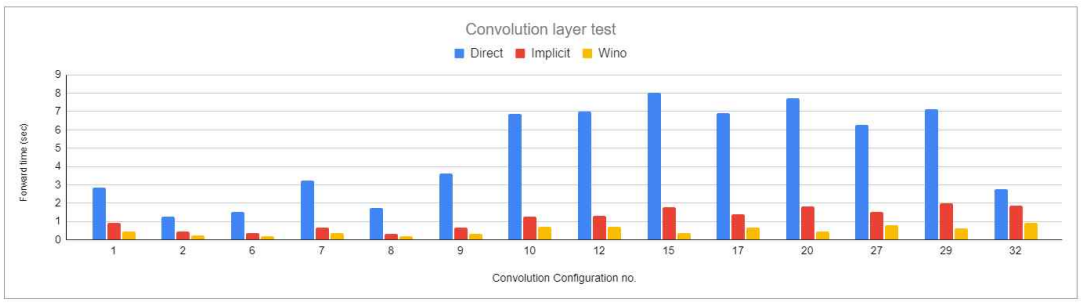
P(세로 tile 개수) 기준으로 분할하여 Winograd Convolution를 수행하는 경우(wino height split)과, C(input channel)을 기준으로 분할하여 수행하는 경우(wino channel split)에 대해 convolution 레이어의 forward 시간을 비교하였다. P 기준으로 분할하는 것이 분할하기 전과 비교했을 때 속도 저하가 거의 없이 Winograd Convolution을 수행하였다. 따라서, 메모리 제약사항을 위반하는 경우에는 P 기준으로 분할하여 처리하도록 구현하였다.



[그림 7-3] Winograd Convolution 분할 방식에 따른 성능 측정

4. Convolution 알고리즘 별 성능 테스트

Direct Convolution, Implicit GEMM Convolution, Winograd Convolution의 성능 측정을 진행하였다. Winograd Convolution은 Direct Convolution에 비해 평균 9.62배, Implicit GEMM Convolution은 Direct Convolution에 비해 평균 3.62배의 속도 향상이 있었다. Winograd Convolution이 가장 좋은 성능을 보였다.



[그림 7-4] Convolution 알고리즘 별 성능 테스트

5. 최적화 기법 적용에 따른 PyNET 추론 시간

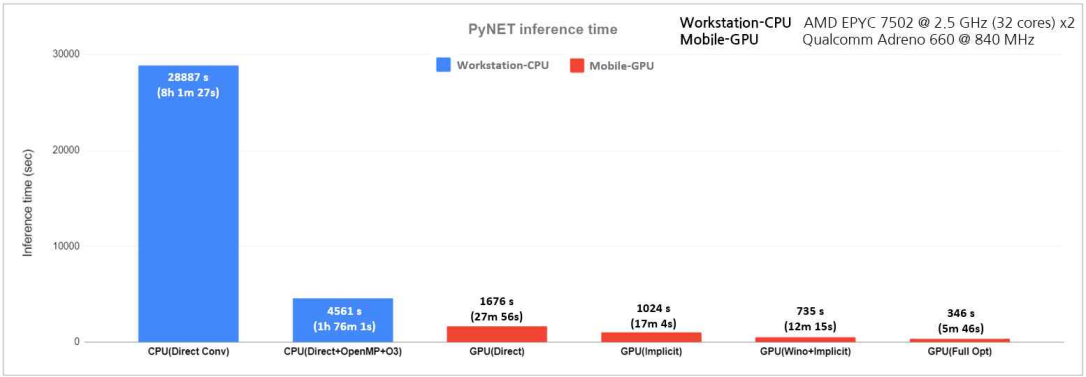
최적화 모델의 GPU 추론 시간은 모바일에서 측정하였고, CPU 추론 시간은 지연 시간 때문에 워크스테이션에서 측정하였다. 워크스테이션 CPU로는 AMD EPYC 7502가 2개 탑재되어 있는 환경이다.

워크스테이션 CPU에서 Direct Convolution을 사용한 추론 시간은 28887s (8h 1m 28s), 여기에 추가적으로 OpenMP를 사용한 추론 시간은 4561s (1h 76m 1s)로 측정되었다.

모바일 GPU에서는 Direct Convolution을 사용한 추론 시간은 1676s (27m 256s), Implicit GEMM Convolution을 사용한 추론 시간은 1024s (17m 4s), Implicit GEMM Convolution와 Winograd Convolution를 함께 사용한 추론 시간은 735s (12m 15s), 여기에 Memory Coalescing, Vectorization 등 모든 최적화 적용한 최종 추론 시간(Full Opt)은 346s

(5m 46s)로 측정되었다.

결과적으로, 최종적인 모바일 GPU 추론 시간은 워크스테이션 CPU 추론 시간 대비 83.4배의 성능이 향상되었다.



[그림 7-5] 최적화에 따른 PyNET 추론 시간

제 8 장 결론

본 논문에서는 모바일에서 핵심 기능을 담당하는 ISP의 기능을 대체하는 딥 러닝 모델인 PyNET 모델을 모바일에서 실행하기 위한 메모리 관리 기법 및 OpenCL 기반의 convolution 최적화 기법을 제시하였다.

모바일의 제약조건을 위반하지 않기 위해서 모델 구조를 변경하거나 입력을 분할하고, GPU 메모리를 초과하는 것을 방지하기 위해 메모리 풀, 메모리 스와핑, Half-precision floating-point 등을 적용해 모델을 경량화 하였다. 또한, 추론 시간을 줄이기 위해 다양한 OpenCL 기반의 convolution 최적화 기법을 적용하였다. 이를 통해, 딥 러닝 프레임워크로는 모바일에서 추론이 불가능한 모델을 모바일 디바이스에서 실험함으로써 이를 검증하였다.

또한 Winograd Convolution의 경우, convolution을 수행하기 위한 추가적인 메모리 공간이 많이 필요하게 되는데, 이를 분할하여 처리할 수 있는 기법을 제시하여 자원이 제한된 모바일 환경에서 Winograd Convolution을 적용하였다. Convolution 성능 실험에서 보시다시피, 다른 알고리즘에 비해 가장 좋은 성능을 보이는 Winograd Convolution을 성능 저하 없이 분할 처리하여 모바일에서 실행 가능하게 함으로써 모델의 성능 향상에 기여하였다.

본 연구에서 소개하는 메모리 관리 기법 및 convolution 최적화 기법은 다른 딥 러닝 모델에서도 폭넓게 활용될 수 있다.

참 고 문 헌

[1] Andrey Ignatov, Luc Van Gool, Radu Timofte. "Replacing Mobile Camera ISP with a Single Deep Learning Model". IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), 2020.

[2] H. N. Loc, Youngki Lee, R. Balan. "DeepMon: Mobile GPU-based Deep Learning Framework for Continuous Vision Applications", MobiSys'17: Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, June 2017.

[3] Shiqi Jiang, Lihao Ran, Ting Cao, Yusen Xu, and Yunxin Liu. "Profiling and optimizing deep learning inference on mobile GPUs", APSys'20: Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems, pp. 75 - 81, 2020.

[4] Olivier Valery, Pangfeng Liu, Jan-Jan Wu, "CPU/GPU Collaboration Techniques for Transfer Learning on Mobile Devices", 2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS), pp. 477-484, 2017.

[5] Han, S., Shen, H., Philipose, M., Agarwal, S., Wolman, A., and Krishnamurthy, A., "Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints", Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '16, pages 123 - 136, New York, NY, USA, 2016.

[6] Andrew Lavin, Scott Gray, "Fast Algorithms for Convolutional Neural Networks", Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 4013-4021, 2016.

[7] Feng Shi, Haochen Li, Yuhe Gao, Benjamin Kuschner, Song-Chun Zhu, "Sparse Winograd Convolutional neural networks on small-scale systolic arrays", FPGA, 2019.

[8] L Lu, Y Liang, Q Xiao, S Yan, "Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs", IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2017.

[9] Artificial Inteligence, GitBook, https://leonardoaraujosantos.gitbook.io/artificial-inteligence/chapter1

[10] OpenCL SGEMM Tutorial https://cnugteren.github.io/tutorial/pages/page4.html

[11] Qualcomm® SnapdragonTM Mobile Platform OpenCL General Programming and Optimization, file:///C:/Users/Administrator/Desktop/80-nb295-11_a.pdf

[12] OpenCL ML SDK, Qualcomm® developer network, https://developer.qualcomm.com/blog/accelerate-your-models-our-opencl-ml-sdk

[13] OpenCL with Mali GPU https://wiki.loliot.net/docs/lang/cpp/libraries/opencl/opencl-with-mali-gpu/

Abstract

Memory Management and Optimization for Deep Learning Model Inference on Mobile GPU

Sooyeon Kang
Department of Computer Science & Engineering
The Graduate School
Seoul National University

Image processing acceleration technology on mobile is becoming more important day by day. Mobile is equipped with ISP(image signal processing) and DSP(digital signal processing) for image processing as well as CPU and GPU, so each processor processes images according to the image processing algorithm in charge. However, as the image processing process on mobile becomes increasingly complex, it costs a lot of labor and money to build an ISP or DSP. Nevertheless, there is a lack of development of image processing technology using mobile GPUs that can replace ISPs. Although the PyNET, a deep learning model that replaces ISPs, exists, the model developed by a deep learning framework is impossible to infer due to

a memory shortage problem on mobile. In this work, we proceed with optimization to infer the PyNET on mobile GPU. In this process, we present various memory management techniques and convolution optimization techniques based on OpenCL to solve the memory shortage and latency problems that arise when inferring deep learning models. Through optimization, we validate this by experimenting with a model that cannot be inferred on mobile by a deep learning framework on real mobile devices.

keywords : Mobile GPU, Deep Learning, Memory Management Techniques, Convolution Optimization, OpenCL.
Student Number : 2012-20726