

|  |
| --- |
| OpenCL을 사용한 CNN 모델 GPU 가속 |
|  |



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 과목명 | 멀티코어 프로그래밍 |  | 조 | 9조 |
| 담당교수 | 박기호 |  | 이름 | 권세준,고병욱,정우인 |

목차

**1. 프로젝트 개요3**

1.1 배경3

1.2 최적화전 성능 분석3

**2. 최적화 방법 소개5**

2.1 Max pooling , Fully connected 병렬처리 방법5

2.2 Conv 병렬처리 방법 1 : 모든 이미지 데이터 로컬화6

2.3 Conv 병렬처리 방법 2 : 워크 아이템 차원 수정 , 이미지 데이터 로컬화8

2.4 Conv 병렬처리 방법 3 : Img2Col , 타일링 , 각 PE 에 더 많은 작업 할당10

**3. Layer 별 적용 및 성능 분석12**

3.1 Max pooling , Fully Connected12

3.2 Convolution13

3.3 추가 최적화 및 최종결과15

**4. 역할분담 및 느낀점17**

4.1 역할분담17

4.2 느낀점18

# 프로젝트 개요

## 배경

본 프로젝트는 OpenCL 을 사용하여 CNN 모델(VGG 16)의 GPU 가속을 구현하는 것을 목표로 한다. CIFAR – 10 데이터셋의 3,000개 이미지를 입력 데이터로 사용하며, 딥러닝 알고리즘의 구조를 변경하지 않는 제한 조건 내에서 정답과 동일한 결과를 출력하면서 수행 시간을 단축하는 것이 주요 목표이다.

프로젝트는 2024년 11월 4일부터 12월 19일까지 수행되었으며, NVIDIA RTX 3060 GPU 기준으로 OpenCL 의 병렬 처리 기능을 최대한 활용하는데 중점을 두었다.

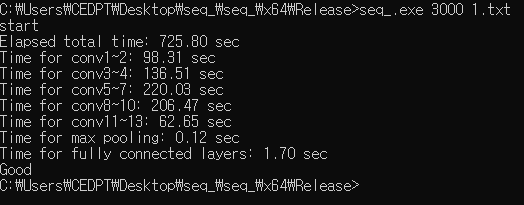
추가적으로, 본 보고서에서 참고해야 할 사항은 다음과 같다.

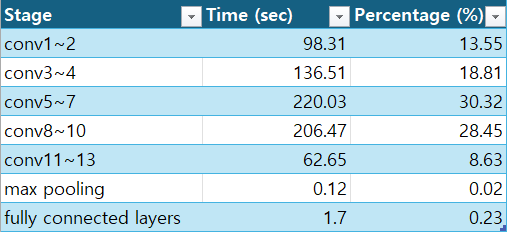
1. 모든 성능 측정치는 실습실 컴퓨터를 기준으로 한다.

2. Conv1 , Conv2 , … , Conv13 는 VGG 16 모델에서 Convolution layer 의 순서를 나타낸다.

## 최적화 전 성능 분석

최적화 전, 3000장의 이미지에 대해 CPU 를 사용하여 VGG16 모델의 성능을 측정하였고 측정은 Conv1~Conv2 , Conv3~Conv4 , Conv5~Conv7 , Conv8~Conv10 , Conv11~Conv13 , Max Pooling , Fully Connected 로 연산 단계를 구분하여 수행되었다.





측정결과 , 전체 연산 중 Convolution layer 가 99.75% 의 비중을 차지하는 것을 확인할 수 있었다. 이를 통해 Convolution layer 가 모델의 성능에 가장 큰 영향을 미친다는 사실을 파악하였으며 , 최적화의 핵심이 Convolution layer 의 연산 효율을 높이는 데 있다는 결론을 도출하였다. Convolution layer 중에서는 layer 가 3개이고 입력채널이 많은 conv5~7 , conv8~10 이 가장 많은 비중을 차지했다.

# 최적화 방법 소개

본 팀은 다양한 병렬처리 방법을 시도하였고 시행착오를 통해서 프로젝트 기간동안 점진적으로 성능을 개선 시킬 수 있었다. 이후 챕터에서 네가지 병렬처리 방법들을 소개할 것이며 각 방법 마다 어떤 문제를 발견하고 어떠한 이유로 방법을 바꾸게 되었는지 성능개선과 함께 보일 것이다.

## 2.1 Max pooling , Fully connected 병렬처리텍스트, 스크린샷, 폰트, 그래픽이(가) 표시된 사진 자동 생성된 설명 방법

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명Max pooling 의 경우 각 워크 아이템이 독립적으로 하나의 출력 픽셀을 처리하도록 병렬화 한다. 이를 통해 GPU 의 수많은 코어를 병렬로 활용할 수 있다. 또한 출력 픽셀은 인덱스가 고유하므로 메모리 쓰기 충돌이 발생하지 않고 워크 아이템간 의존성이 없기 때문에 동기화 비용 또한 필요하지 않다.

Fully connected layer에서는 로컬 메모리를 활용한 리덕션을 통해 성능을 향상했다. 지정된 워크 아이템만 전역 출력 버퍼 메모리(output)에 접근하면 되기에 메모리 접근 비용을 크게 줄일 수 있었다.

## 2.2 Convolution 병렬처리 방법 1 : 모든 이미지 데이터 로컬화

본 팀이 가장 먼저 시도한 방법은 입력으로 들어온 이미지 데이터를 최대한 로컬 메모리에 올려서 합성곱 연산을 수행하도록 하는 것이다. 워크 아이템의 Global Size 는 3차원으로 설정하여 1차원 , 2차원에는 입력으로 들어오는 이미지의 가로 , 세로크기와 같도록 하였고 3차원 축은 (이미지 하나의 채널수) \* (이미지의 개수)가 되도록 하여 이미지 여러 개를 병렬로 처리하고 직관적으로 이해하기 쉽도록 설정하였다.

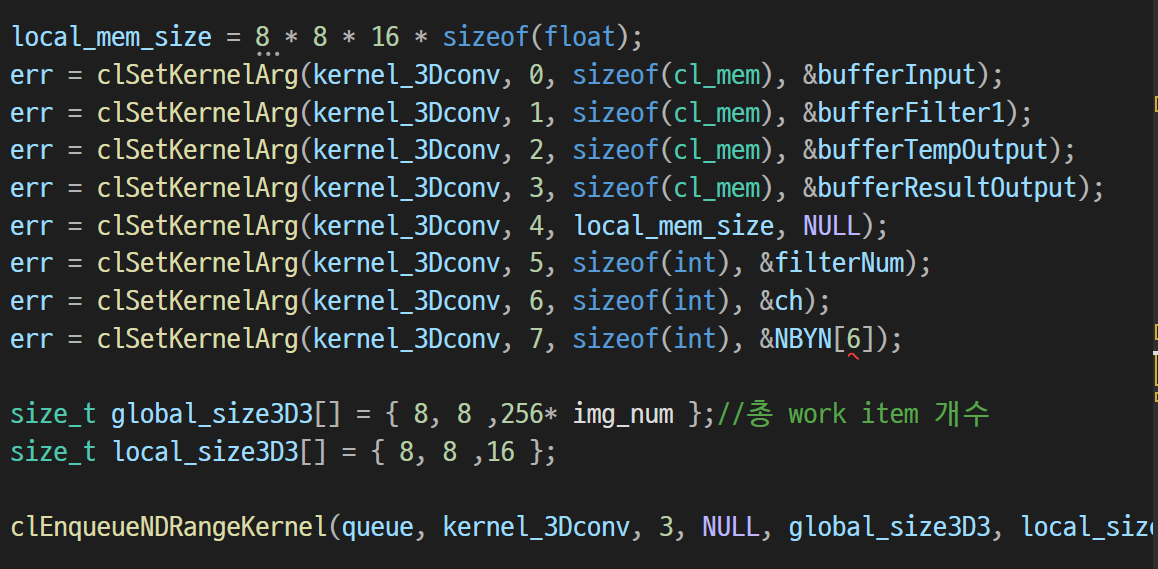
워크 그룹의 모양 또한 3차원이며 Global Size와 같이 1차원 , 2차원은 이미지의 가로 세로와 같도록 하고 3차원 축은 워크그룹의 크기 한도 내에서 가장 크게 설정하였다.

로컬메모리를 워크그룹의 크기와 동일하게 맞추어서 커널코드에서 입력으로 들어온 이미지를 모두 로컬메모리로 이동시켰고 입력 이미지를 필터의 각 요소와 곱하는 과정에서 이미지 데이터에 접근하는 연산을 모두 로컬메모리에서 수행할 수 있었다.

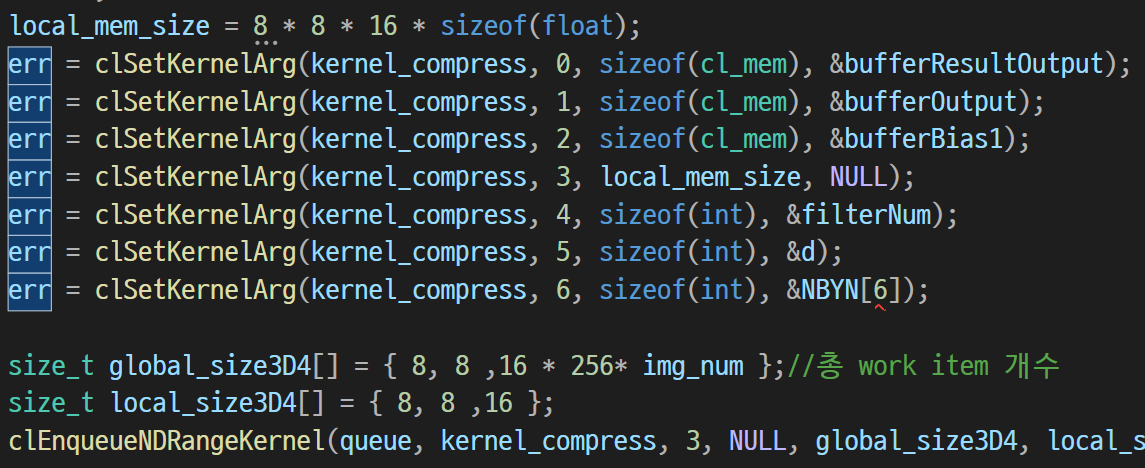
라인, 직사각형, 스크린샷, 잭이(가) 표시된 사진

자동 생성된 설명

합성곱 연산은 하나의 이미지 데이터를 필터의 각 요소와 곱해서 나온 결과를 다시 한 장으로 더해서 압축하는 과정이 필요하고 이 과정을 리덕션을 통해서 수행하고자 하였다. 그러나 위에서 설정한 워크그룹 모양을 그대로 사용한다면 더하는 과정에서 동기화 문제로 인해 한번에 리덕션이 완료되지 않는다. 로컬 메모리의 3차원 축이 이미지의 채널수 만큼 늘어날 수 없기 때문이다.따라서 Convolution을 수행하는 커널함수를 두개로 분리하여 두번째 커널코드는 첫번째 커널코드의 결과를 이어받아서 리덕션을 통해 최종결과 Output을 만들도록 하였다.



<합성곱 연산을 위해 첫번째 커널코드를 호출하는 호스트 코드>

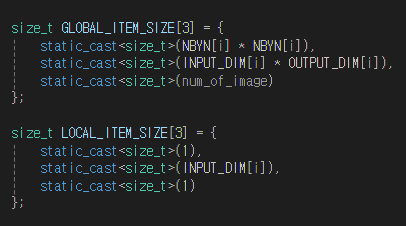


<합성곱 연산을 위해 두번째 커널코드를 호출하는 호스트 코드(global size의 3차원 값이 256\*256\*img\_Num 이 아닌 이유는 첫번째 커널에서도 리덕션을 수행해서 3차원 축이 줄어들었기 때문임)>

이 방법으로 배치사이즈(Img\_num)을 128,256으로 설정하여 3000장 실행시간을 측정하였고 약 15초 정도 소요 되었다. 사후 검토를 통해 문제점을 발견 하였다.

이 방법은 Convolution을 위한 첫번째 커널 코드의 워크아이템 개수를 설정할 때 Z축의 길이가 X,Y축의 길이보다 지나치게 길어지는 것을 방지하기 위해서 global\_size를 N\*N\*inDim\*outDim\*Img\_Num으로 설정하지 않고 N\*N\*inDim\*Img\_Num으로 설정하였기 때문에 각 이미지에 대해서 필터의 개수만큼 반복문을 통해서 연산한다.또한 워크그룹의 z축 최대 크기가 64인 문제도 있었기 때문에 워크그룹을 통해 로컬메모리를 최대한 활용하기 힘들었다.이러한 문제점을 개선하기 위해서 워크아이템의 모양 자체를 다시 설정할 필요가 있었다.

## 2.3 Convolution 병렬처리 방법 2 : 워크 아이템 차원 수정 , 이미지 데이터 로컬화



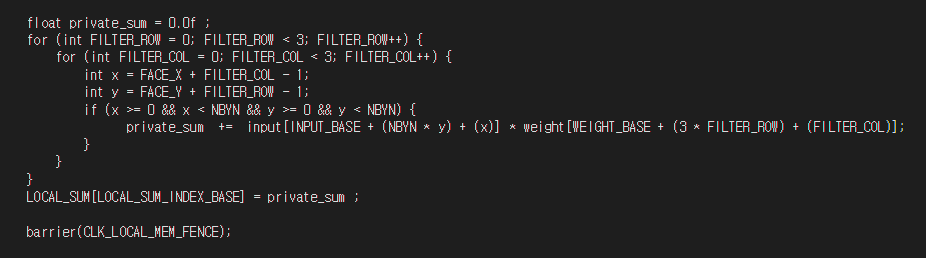
본 병렬처리 커널에서 GLOBAL\_ITEM\_SIZE 와 LOCAL\_ITEM\_SIZE 위와 같다.

X 축 NBYN[i] \* NBYN[i] 는 이미지의 각 픽셀을 병렬로 처리하기 위해 설정된 축이고 NBYN[i] 는 이미지 한 변의 길이를 나타내며 , NBYN[i] \* NBYN[i] 는 해당 이미지의 총 픽셀 수를 의미한다. 이를 통해 각 워크 아이템이 하나의 픽셀에 대한 연산을 담당하도록 설계했다.

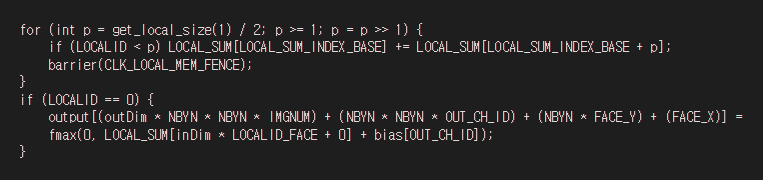
Y 축 INPUT\_DIM[i] \* OUTPUT\_DIM[i] 는 각각의 워크 그룹에서 리덕션을 진행한 후 하나의 출력 픽셀을 나타내기 위해 설정된 축이다. 로컬 메모리의 모양을 위 그림과 같이 설정하여 한번에 리덕션이 가능하도록 하였다.

Z 축은 여러 이미지를 한번에 처리하는 배치 연산을 위해 설정하였다.

이 방법은 Convolution을 위한 커널코드 하나만 사용했다. 때문에 이미지 데이터를 필터의 각 요소와 곱하고 더하는 연산은 글로벌에서 수행하되 이 결과값들을 로컬메모리에 저장해서 같은 커널코드에서 바로 리덕션을 한번에 수행하는 것이다.



커널 코드를 보면 각 워크 아이템은 3X3 필터를 적용해 입력데이터를 곱하고 합산하는 연산을 수행한다. 계산된 중간 결과는 로컬 메모리(LOCAL\_SUM)에 저장되는데 이는 추후 리덕션을 수행하기 위해 사용된다.

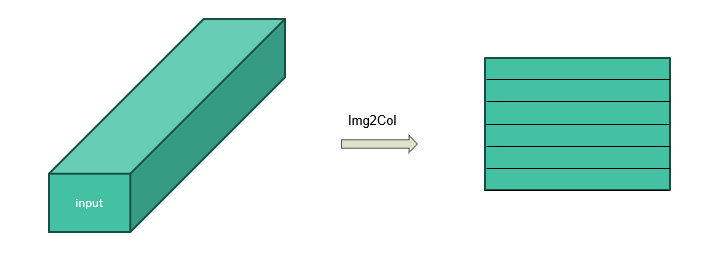


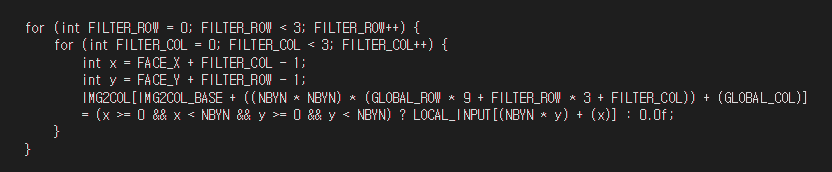
이후, 로컬 메모리(LOCAL\_SUM)에 저장된 값은 리덕션 단계를 통해 워크 그룹 내 워크 아이템들이 협력하여 병합되며 , 최종 결과는 전역 출력 버퍼 메모리(output)에 저장된다.

주요 이점으로는 첫번째로 고수준의 병렬처리가 가능하다. 반복문을 최대한 제거함으로써 입력 이미지의 모든 픽셀을 동시에 계산할 수 있어 처리 속도를 향상시켰다.

최종적으로 3000장 처리속도는 약 9초로 측정 되었다. 이 방법도 첫번째 방법과 같이 필터를 적용하는 커널코드를 두가지로 분리하여 합성곱과 리덕션 연산에서 이미지 데이터를 모두 로컬에서 수행하도록 개선할 수 있다. 하지만 첫번째 방법과 두번째 방법은 결국 입력으로 들어오는 이미지 데이터에 대해서만 로컬 메모리를 이용할 수 있는 만큼 성능개선에 한계를 느껴 다른 최적화 방법을 사용하게 되었다.

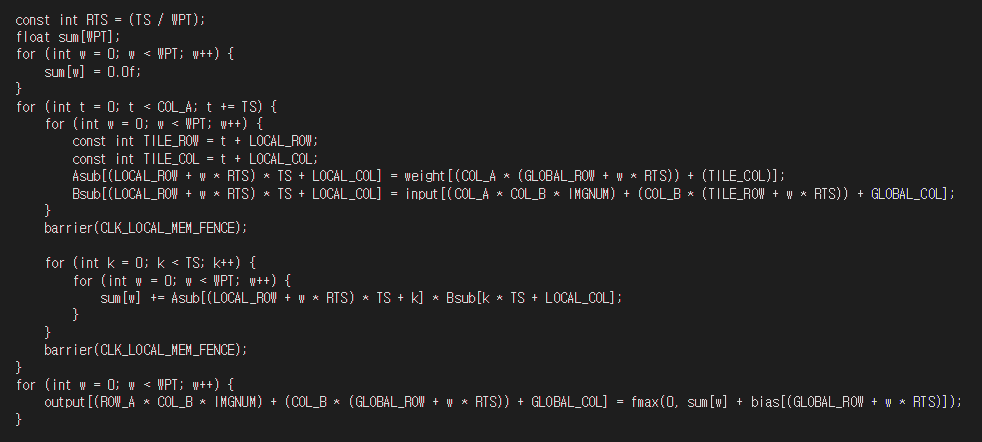
## 2.4 Convolution 병렬처리 방법 3 : Img2Col , 타일링 , 각 PE에 더 많은 작업 할당



Img2Col 은 입력 이미지(input)를 3X3 필터에 맞게 형렬 행태로 변환하는 기법이다. 즉 , 입력이미지의 모든 필터 영역을 행으로 펼쳐서 행렬로 변환한다. 이 변환된 행렬은 Convolution layer 에서의 연산을 행렬 곱셈으로 처리할 수 있게 해주기 때문에 그동안 지속적으로 겪었던 리덕션에 대한 문제를 해결할 수 있다. 기본적으로 GPU 는 행렬 곱셈에 최적화되어 있고 연속적인 메모리 접근이 가능하게 되기 때문에 이는 연산의 성능을 크게 개선할 수 있다.

위의 커널 코드는 이미지를 필터와 행렬곱을 하기 위한 모양으로 펼치는 부분이다. 해당 반복문을 돌기 전 입력 데이터는 모두 로컬메모리(LOCAL\_INPUT)에 저장하였다.

행렬곱을 할 때 [N x K] \* [K \* M] 형식으로 가정하면 왼쪽은 weight, 오른쪽은 input image로 상정하여 글로벌 워크 사이즈와 그룹 워크 사이즈를 정하였다. 필터의 크기가 3 x 3 이기 때문에 다차원의 feature맵을 2d행렬로 펼치고 나면 [(9 \* inDim) x (NBYN \* NBYN)] 의 shape 이 된다. 그리고 행렬곱 항 왼쪽의 weight는 [outDim x (9 \* inDim)] 로 일반화 할 수 있다. 이 행렬곱을 거치고 나면 한 컨볼루션 레이어를 통과한 후의 feature map이 그대로 나오고 따로 전치를 하는 등의 처리를 해 줄 필요가 없이 그대로 Row major 형식으로 버퍼에 저장되어 다음 레이어의 입력으로 사용할 수 있게 된다. 이를 바탕으로 global work item 의 shape은 행렬곱을 진행한 후의 픽셀 shape으로 정하여 원활하게 행렬곱 타일링 연산을 진행할 수 있도록 하였다.



Img2Col 을 통해 행렬로 변환된 이미지는 로컬 메모리(Asub,Bsub)에 필터와 함께 타일 단위(TS)로 로드한후 행렬 곱셈을 진행한다. 기본적인 타일링에서 더욱 성능을 높이기 위해서 각 워크아이템에 더 많은 작업을 할당하도록 하였다. 즉, 로컬메모리의 크기보다 워크그룹을 더 작게 설정하여 워크아이템의 레지스터 활용을 최대화 하는 것이다. 이때 하나의 워크 아이템이 WPT 개의 연산을 처리하도록 해 메모리 병목을 완화한다. 이후 각각의 결과에 대해 ReLU 함수를 적용한 후 전역 출력 버퍼 메모리(output)에 저장한다.

주요 이점을 정리하면 첫번째로 Img2Col 을 통해 Convolution layer 에서의 연산을 행렬 곱셈으로 변환해 GPU 에서 연산처리가 용이하도록 하였다.

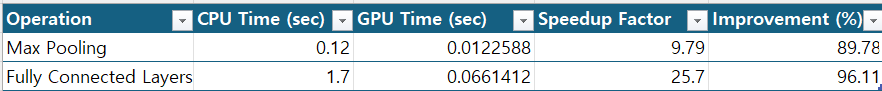
두번째로 타일링 기법으로 로컬메모리를 활용해 메모리 접근 비용을 줄일 수 있고 WPT(Work Per Thread)로 계산량 대비 메모리 접근 비용의 비율을 개선시켜 메모리 대역폭을 절약하고 계산 성능을 극대화 하였다.

특히 기존의 기본적인 타일링 기법에서는 타일크기가 32 x 32 가 한계였으나(글로벌 워크 아이템의 모든 축 곱이 1024 이하여야 하는 GPU 사양상의 제한이 존재하였음) more work per thread 기법을 적용하게 되면서 행방향의 로컬 워크 아이템 개수를 2의 지수로 나눌 수 있게 되어 WPT 16으로 설정했을 때 타일크기를 64 x 64로 늘릴수 있었다.

결론적으로 해당 병렬처리 방법으로 3000장 수행시간을 약 2.6초로 단축시킬 수 있었다.

# Layer 별 적용 및 성능 분석

## Max pooling , Fully Connected

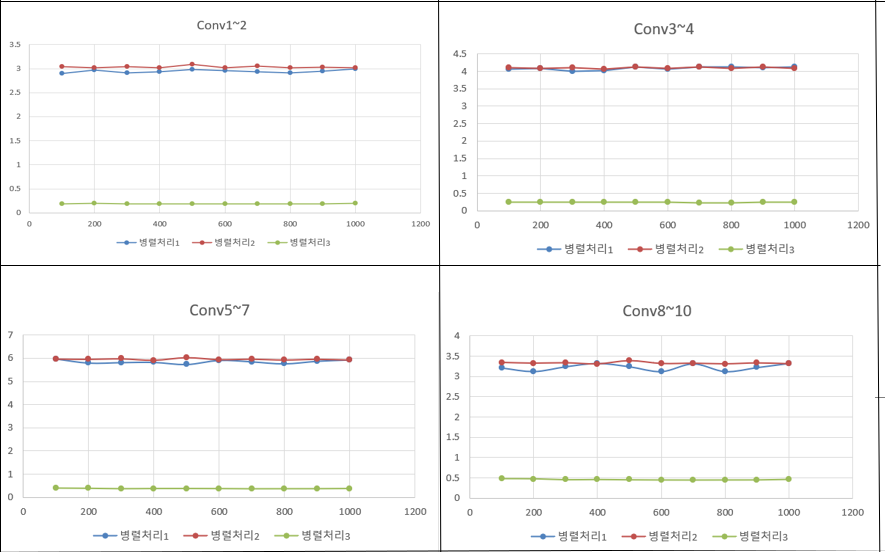


Max pooling , Fully connected layer 에서 병렬처리를 적용했을 때의 성능은 위와 같다.

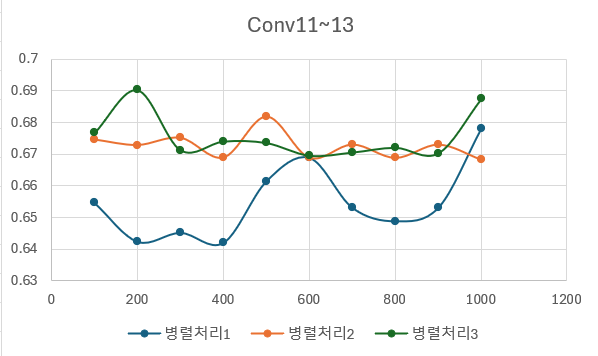
Max pooling 에서 병렬처리를 적용 했을 때 , 약 0.0122588 초로 CPU 에서 실행했을 때 보다 9.79배 빨라졌다.

Fully connected layer 에서 병렬처리를 적용 했을 때 , 약 0.0661412 초로 CPU 에서 실행했을 때 보다 25.7배 빨라졌다.

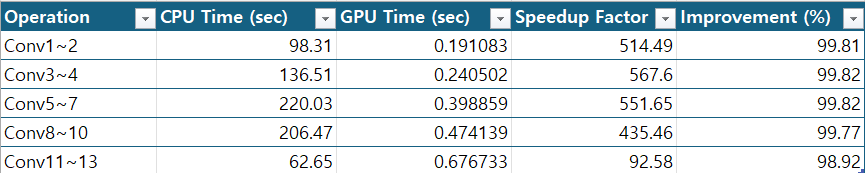
## Convolution

앞에서 소개한 3가지 병렬처리 기법에 대해 다양한 배치 사이즈(100 ~1000)에 대해 적용해 보고 실행시간을 측정하였다. 적용 대상은 5개의 Covolution layer 그룹이다.

먼저 Conv1~2 , Conv3~4 , Conv5~7 , Conv8~10 은 모두 병렬처리3이 최적의 성능을 내는 것으로 측정되었다.



그러나 Conv11~13 에서는 각 병렬처리 기법 1~3 모두 비슷한 성능을 보였고 오히려 첫번째 병렬처리 기법이 가장 빠르게 측정되었다.



위의 표는 병렬처리 기법3으로 배치 사이즈가 100일 때, 각 Convolution layer 그룹별로 실행시간을 CPU 에서의 실행시간과 비교한 결과다. Conv11~13은 CPU와는 반대로 GPU에서 다른 Convolution layer 그룹들보다 더 많은 시간이 걸리는 것으로 측정되었다.

따라서 마지막 3개의 Convolution layer 는 이미지의 가로 , 세로 크기가 아주 작아지기 때문에 기존의 타일 크기가 작아질 수 밖에 없었고 타일링의 효율이 심각하게 저하되는 것으로 판단하였다.이 부분에 대한 개선 방안은 다음 챕터에서 자세히 다룬다.

## 추가 최적화 및 최종결과

위에서 언급한 것과 같이 마지막 convolution layer에서 기존 타일링 방법의 효율이 매우 떨어지는 것을 확인했기 때문에 추가적인 가속 방법을 고안하였다.

마지막 convolution layer는 이미지의 가로,세로 크기가 2이기 때문에 기존 방법대로 타일링을 수행하려면 타일의 크기가 2\*2 로 작아질 수 밖에 없었다. 하지만 모든 이미지는 동일한 필터를 사용하고 있기 때문에 타일링이 수행되는 공간 안에 여러 장의 이미지가 들어갈 수 있다면 이미지의 크기에 구애받지 않고 타일사이즈를 늘릴 수 있다. 즉 기존에 타일링하는 코드에서 이미지 데이터가 들어가는 로컬메모리(타일)에 여러 장의 이미지가 들어가도록 하는 것이다.

디자인, 라인, 스크린샷, 도표이(가) 표시된 사진

자동 생성된 설명  
마지막 convolution layer에 입력으로 들어오는 이미지를 시각적으로 표현하면 위 그림의 직사각형과 같다. 호스트에서 워크아이템 크기를 설정할 때 빨간 선과 같이 1차원 축 크기를 N\*N 보다 크게 설정한 다음 타일 사이즈를 원하는 만큼 늘린다.

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

<커널코드 안에서 하나의 채널에 여러장의 이미지 데이터를 가져오기 위해서 워크아이템의 x축 크기를 늘림(대신 3차원 축 크기는 비례하게 줄임) >

디자인, 라인, 스크린샷, 도표이(가) 표시된 사진

자동 생성된 설명



이후 커널코드에서 뒤에 있는 이지미 데이터까지 타일로 가져온다.이후 계산과정은 모두 전과 동일하다.연산을 모두 마치면 이미지 여러장의 계산 결과가 하나의 워크그룹 안에 저장되고 output 모양에 맞도록 적절히 인덱싱을 수행해서 outputbuffer에 저장한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

<이미지 데이터를 위한 로컬메모리 Bsub에 뒷 채널에 있는 이미지의 데이터도 넣고 있다(추가로 work per thread까지 적용된 모습))>

이 방법을 통해서 Con11~13의 실행시간을 70%이상 줄일 수 있었다.

또한 타일링 수행에 있어서 이미지 크기에 제약을 받고있는 Conv8~10에도 추가 적용하였고 기존 3번째 병렬처리 방법의 실행결과를 2.6초에서 약 1.46초로 단축시킬 수 있었다.

텍스트, 라인, 그래프, 도표이(가) 표시된 사진

자동 생성된 설명

[최종결과]

# 역할분담 및 느낀점

## 4.1 역할분담

본 프로젝트는 OpenCL 기반 GPU 가속화를 통해 CNN으로 3000장의 이미지를 최소의 시간에 처리하는 목표를 달성하기 위해 다양한 역할로 나누어 진행되었다. 각 역할은 프로젝트의 성과를 극대화하기 위해 명확하게 정의되었으며, 협업을 통해 최적의 결과를 도출하였다. 역할 분담 목록은 다음과 같다.

1. 성능 분석

기존 CPU 기반 처리 시간과 GPU 가속화 이후의 처리 시간을 측정 및 비교하고 배치 사이즈와 레이어 그룹별로 처리시간 데이터를 수집하고 시각화 한다. 또한 성능 병목이 발생하는 구간을 분석하고 개선방향을 도출한다.

2. 보고서 작성 및 발표 자료 준비

프로젝트의 목표 , 최적화 기법 , 성능 분석 결과를 포함한 보고서를 작성한다. 그래프 , 표 , 이미지를 활용해 결과를 시각적으로 표현한 발표자료를 준비한다.

3. 병렬 처리 기법 설계 및 코딩

다양한 최적화 기법을 바탕으로 OpenCL 의 커널을 설계하고 CNN 의 각 연산을 실습실 GPU 에 최적화된 방식으로 구현하는 역할을 수행한다.

4. 추가 최적화

성능 분석 측정치를 기반으로 초기 설계 및 구현된 커널에 대해 성능 병목을 해결하고 더 높은 성능을 확보하기 위해 추가적인 최적화를 수행한다.

**권세준** :병렬 처리 기법 설계 및 코딩 , 추가 최적화

**고병욱** : 병렬 처리 기법 설계 및 코딩 , 추가 최적화

**정우인** : 성능 분석 , 보고서 작성 및 발표 자료 준비

## 4.2 느낀점

GPU에서 연산을 가속하기 위해서 모든 연산을 ‘동시에’ 실행하도록 하는 것이 꼭 정답이 아닐 수 있다라는 것을 느꼈다. 기본적으로 병렬성을 극대화하는 것이 중요하지만 반복문을 사용하더라도 연산의 흐름을 체계적으로 분석해서 하드웨어의 대역폭이나 연산 유닛의 개수와 같은 요구사항을 최대한 만족시킬 수 있도록 하는 것이 중요함을 느꼈다.

본 팀은 여러가지병렬기법을 적용해 봄으로서 타 팀들에 비해 성능을 개선시키는 점에 있어서 속도가 느렸다. 하지만 다양한 시도와 경험을 통해 병렬처리 프로그래밍에 익숙해질 수 있었고 이를 바탕으로 지속적인 고민과 개선을 통해 만족할 만한 결과를 얻을 수 있었다.

3명이서 팀으로 시간을 줄이려 하지 않았다면 절대로 1초대로 진입할 수 없었을 것이라는 것도 느꼈다. 괄목할 만한 성능향상은 한 명이서 코드를 붙잡고 있을 때 나온 것이 아니라 각 팀원이 의견을 개진하고 받아들일 때 나왔기 때문이다. 결국 현대문명의 고도화된 문제들을 해결하기 위해서는 다른 생각을 하는 사람들이 모여야 가능하다는 생각이 들었다. 유연한 사고를 하기 위해서는 타인의 생각이 필요하다.