멀티코어 프로그래밍(001) [~2020/12/7]

프로젝트 최종 보고서 16011105 이재성

OPENCL을 이용한 CNN 추론 가속화 프로젝트

1. **프로젝트에 대한 분석**

저희가 직접적으로 작성해야하는 부분은 **cnn\_opencl.c** 와 cnn\_opencl.c에서 GPU를 이용하여 가속화 시킬 커널 코드를 작성할 **kernel.cl** 부분입니다.

저희는 network.bin에서 읽어온 Weight와 Bias에 해당하는 값들을 가져와서 num\_images만큼의 cifar\_10\_image.bin에서 읽어온 이미지에 대해 정해진 네트워크 구조에 따라 값을 계산한후 맨 마지막에 나오는 label과 그에 해당하는 confidence를 계산하여 labels와 confidences 메모리에 저장합니다.

이를 result.out으로 포맷을 맞게 저장한 후에 주어진 answer.out와 제가 만든 result.out과 비교하면서 0번값부터 num\_images까지 각 이미지의 Label, Confidence 결과에 따라

**1. 10개의 클래스에 대해 같게 분류 [Label]**

**2. 오차 0.01 이내의 같은 confidence값을 지님 [Confidence]**

을 확인하는 과정을 겪게 됩니다.

Main 함수에서 부르게 될cnn(images, network\_sliced, labels, confidences, num\_images)을 통해 주어진 이미지를 적절한 네트워크 구조에 따라 연산을 하여 내놓은 결과 값과 그 연산을 하는 데에 걸린 시간 및 정확성이 이 프로젝트의 중점이라고 생각합니다.

이미 다른 부분은 다 주어진 상태로 적절하게 구현되어 있으므로 cnn\_init함수 안에서 OPENCL을 사용할 수 있도록 설정을 하고 cnn함수 안에서 적절한 네트워크 구조에 따른 Conv, Pooling, Fully Connected, SoftMax부분을 적절하게 GPU를 사용하여 시간을 단축하는 것이 최종 목표입니다.

일단 이번 프로젝트에서 쓰는 cnn 함수의 구조를 큰 틀에서 살펴보게 되면

각각의 이미지에 대해

**Image**

**->Conv->Conv->Pooling**

**->Conv->Conv->Pooling**

**->Conv->Conv->Conv->Pooling**

**->Conv->Conv->Conv->Pooling**

**->Conv->Conv->Conv->Pooling**

**->FC->FC->FC->SoftMax->Label & Confidence Save**

식의 네트워크구조를 진행하게 됩니다. Conv과정과 FC과정에서는 network.bin값으로 주어진 네트워크 Weight, Bias값을 사용합니다. 이에 반해 Pooling, SoftMax과정에서는 주어진 Input에 대해서만 가공하여 Output을 생성하게 됩니다.

1. **CNN에 대해 이해한 내용**

CNN(Convolutional Neural Network)는 Convolution 연산을 이용하여 가공하는 과정입니다.

일반적으로 학습과정에서는 Filter(Weight)을 훈련시키기 위해 적절한 오차 함수를 두고 BackPropagation 을 통해 필터의 가중치들을 학습시키게 됩니다.

이번 프로젝트에서는 훈련시키는 과정이 아닌, 이미 훈련되어서 정해진 Weight와 Bias를 이용하여 CNN의 구조(Conv Layer, Pooling Layer, FC Layer) 에 따라 계산을 진행합니다. 즉 추론과정을 GPU를 사용하여 가속화하는 것을 구현하는 것입니다.

Convolution 과정은 입력된 Input에 zero padding을 한후 Filter와 Convolution연산을 한후 나온 값에 대해 Bias를 더해주는 과정입니다. 이 과정에서 Filter의 개수 만큼 Featuremap이 나오게 되고 이것이 다음 Layer의 입력으로 들어가게 됩니다.

Pooling 과정은 입력된 Input에서 2x2크기의 구역에서 값 하나를 Output해당 위치에 저장하는 과정입니다. 즉 Output의 크기는 Input크기의 1/4가 됩니다. Pooling에는 여러 알고리즘(Max, Average, Min)등이 있는데 프로젝트에서 사용하는 것은 MaxPooling으로써 앞에서 말한 2x2크기의 구역에서 가장 큰값하나를 뽑아 Output해당 위치에 넣어주는 것입니다. [물론 2x2풀링이 아닌 다른 풀링도 존재합니다]

FC 과정은 들어온 Input에 대해서 행렬 연산처럼 단순히 Weight를 적절하게 곱하고 Bias를 더해주어 Output으로 반환하는 과정입니다.

이러한 과정들은 크게 2단계로 나눌수 있게 됩니다.

1. **Feature Extration [Conv Layer & Pool Layer]**

이부분은 주어진 데이터(이미지)로부터 특성을 추출해 내는 과정입니다. 주로 Conv Layer의 Filter(Weight&Bias)를 이용하여 특징을 추출하고 Pooling Layer를 통해 특징을 강화시키고 Feature Map의 크기를 줄입니다.

1. **Classification [Fully Connected]**

1차원의 Fully Connected형태를 이용하여 분류작업을 하고 맨 마지막에 Output 크기 10짜리로 결과를 내놓은후 SoftMax작업을 거쳐서 Label과 Confidence를 저장합니다.

1. **CNN의 전체적인 구조**

이프로젝트에서 CNN함수의 전체적인 구조는 총 크게 6단계 입니다.

**Image**

**->Conv->Conv->Pooling (A)**

**->Conv->Conv->Pooling (B)**

**->Conv->Conv->Conv->Pooling (C)**

**->Conv->Conv->Conv->Pooling (D)**

**->Conv->Conv->Conv->Pooling (E)**

**->FC->FC->FC->SoftMax->Label & Confidence Save (F)**

전체적인 구조는 위와 같고, (A)~(E)단계는 Feature Extraction, (F)단계는 Classification이라고 할 수 있습니다.

(A)(B)단계에서는 Convolution 연산을 2번한후 Pooling을 진행하고

(C)(D)(E)단계에서는 Convolution 연산을 3번한후 Pooling을 진행하면서

(F)단계에서는 FC Layer를 3번 거친 후 SoftMax연산을 통해 결과를 계산하게 됩니다.

1. **Convolution Layer**

Convolution연산을 진행하는 레이어 입니다. 저희는 주어진 Weight(filter)와 Bias를 이용하여 Input에 Convolution 연산을 진행하는데 논리적인 zeropadding을 이용하여 Conv연산을 진행하였고 ReLU activation을 이용하여 Output을 내놓게 됩니다. 가장 많은 시간을 투자해야 할 부분으로 생각합니다. 이부분이 대부분 연산의 시간을 차지하기 때문입니다. 필터 개수만큼 Output이 나오게 됩니다.

1. **Pooling Layer**

이번 프로젝트에서는 MaxPooling 방식을 사용합니다. 단순하게 들어온 Input혹은 Activation Map(Feature Map을 Relu로 activation한후의 Map)에 대해서 너비를 1/4로 줄입니다. Input에 2x2window를 stride 2로 주어서 훓으면서 가장 큰값을 찾아서 Output에 해당위치에 쓰는 작업으로 노이즈가 감소하고 Image Classification 작업에 좋은 영향을 준다고 합니다. 즉 크기를 감소시키고 feature를 부각시킵니다.

c) **Fully Connected Layer**

3차원 Output을 1차원으로 바꿔서 계산을 진행하게 해주는 레이어입니다. 이 부분을 통해서 맨마지막 Output의 크기 10(클래스 개수) 의 확률을 SoftMax를 통해 구할수 있게 됩니다. Classification을 위한 레이어입니다.

1. **병렬 처리 전략**
2. **Convolutional Layer**

프로젝트를 진행하면서 가장 많은 시간을 소비하는 곳이 Convolution 연산을 진행하는 이 Conv Layer인 것을 알았습니다. 그렇기에 초반에는 Conv Layer의 출력인 D2xNxN의 연산을 D2의 globalsize를 두고 각각의 PE내에서 D1xNxN의 연산을 하게 하였지만 좀더 좋은 성능향상을 위해 (NxN,D2,Batch)식으로 globalsize를 주어서 각각의 PE가 단순히 D1x9(필터루프)+1(bias)번의 연산을 하도록 배분해주었습니다.

다시 말해, 각각의 PE가 담당하는 일을 최대한 배분을 잘해주어서 하나가 일할 때 다른 것들이 놀지 않도록 적절하게 일을 배정해 주는 전략으로 프로젝트를 진행했습니다. 가능하면 local memory또한 사용하여 좀더 성능향상을 이루고 싶었으나, 시간의 부족으로 인해 구현하지 못했습니다.

1. **Pooling Layer**

프로젝트에서는 MaxPooling기능을 하는 layer에 대한 병렬화 전략을 짤 때 병렬화를 진행한 것이 좀더 좋은 성능을 내었기 때문에 Pooling Layer또한 병렬화를 진행하기로 하였습니다. 추가적으로 batch개념을 도입하여 좀더 적은 오버헤드 발생을 위해서도 병렬화가 필수적이였습니다.

그렇기 때문에 (DxNxN,BATCH)를 globalworkspace로 준후 안에서 Output에 해당하는 인덱스마다 2x2크기의 구역을 비교하면서 가장 최대 값을 가져와서 저장하도록 하였습니다.

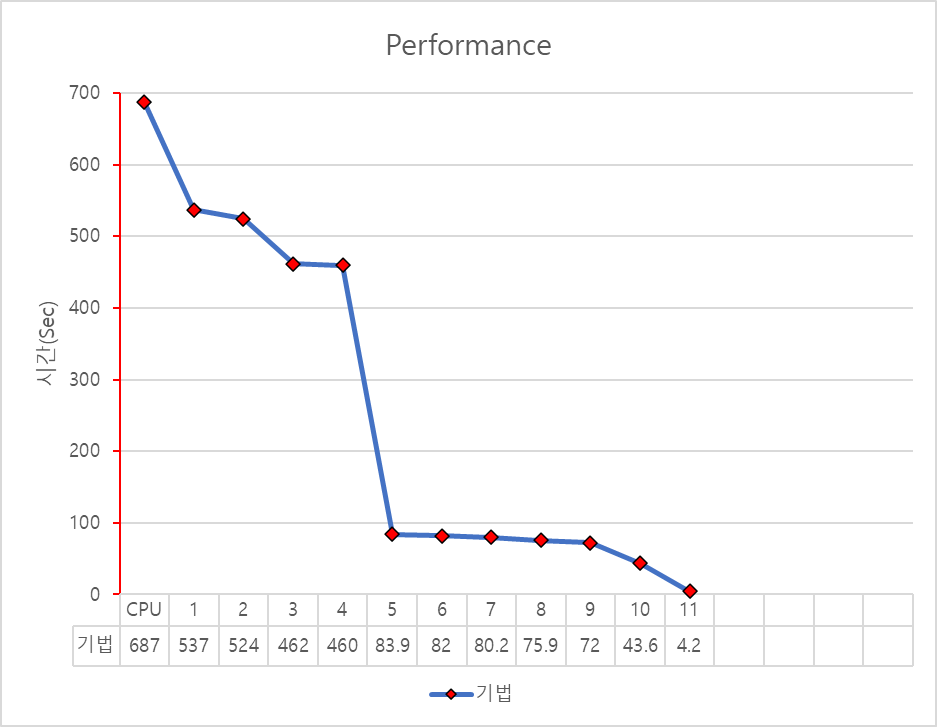
1. **Fully Connected Layer**

초기에 병렬화를 시켰더니 오히려 좋지않은 결과가 나왔기에 병렬화를 시키지 않으려고 하였으나, BATCH개념을 생각해서 오히려 병렬화를 시키는 것이 좋은 결과가 나올 것 같았고, 예상대로 배치를 사용하면서 메모리 복사 오버헤드를 줄이면서 좀더 좋은 결과가 나오게 되었습니다.

1. **전체적인 구조, 데이터 흐름에 대해 전략**

이번 프로젝트에서 데이터 흐름에 매우 많은 시간을 들였습니다. 원래 제공된 cnn\_seq.c의 구조를 차용하여 그대로 가져와서 각각의 함수내에서 계속 쓸모없는 데이터 복사를 줄이는 것에 대해 많은 고민을 하였습니다. 주로 num\_image에 해당하는 루프안에서 계속 함수로 보내는 Weight,bias등에 대한 메모리 복사 오버헤드, input\_neuron, output\_neuron에 대한 메모리 복사 오버헤드, kernel 함수내에서 선언 오버헤드등, 함수 내에는 오직 cl\_mem를 받아서 GPU로 보내서 계산하게끔만 설정하여 이러한 오버헤드등을 줄이는 것에 대한 노력을 많이 했습니다.

1. **병렬 처리 결과**



5: Conv연산 LocalWorkSize 배정

3: 네트워크 메모리 복사 개편

4: 3x3필터 루프 간편화

2: Private 메모리 활용

1: Conv 병렬화

6: Pool & FC 병렬화

7: Conv worksize 재배정

8: 입출력 데이터흐름 개편

9: Kernel 선언 개편&

Conv worksize재배정

10. BATCH 적용 및 CONV, POOL, FC Layer 수정

11. Conv 알고리즘 수정

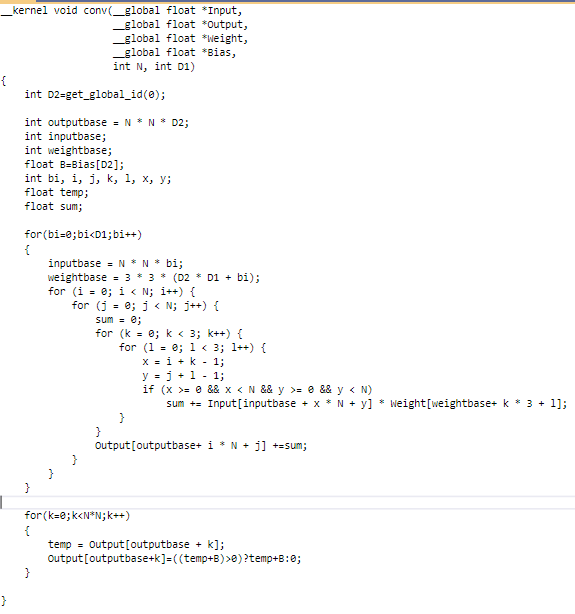
CPU: i5-8500 3.00GHz

GPU: Nvidia GeForce RTX 2060

1. **적용한 기법**
2. **CPU단계**: 주어진 cnn\_seq.c를 기준으로 돌렸을 때 나온 결과입니다. 이 단계는 단순히 비교를 하기위한 기준점을 만들기 위해 제공된 코드 그대로 실행했으므로 적용한 기법은 없습니다. **(약 687초)**

이 단계에서는 시행착오가 따로 없었고 주로 전체적인 코드의 흐름을 보면서 어떤 식으로 병렬처리를 해야 좀더 좋은 성능을 낼지에 대해 고민하던 상황이었습니다. 주로 conv layer에 많은 시간이 걸린다는 것을 알았고 이러한 고민 속에 cnn\_seq의 구조를 그대로 차용하면서 GPU를 이용해 각각의 함수 내에서 가속화를 하는 방향으로 진행하는 것을 생각했습니다.

1. **1단계**: 단순히 Convolution Layer에 해당하는 부분에서 OpenCL을 이용하여 GPU를 이용하게 만든 단계입니다.

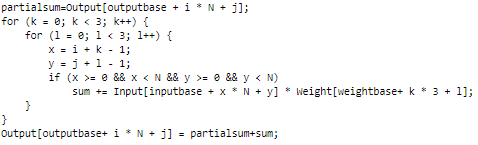
즉, GlobalWorkSize를 저희가 원하는 Output 크기인 D2로 잡고 각각의 PE가 get\_global\_id(0)로 해당 Output에 해당하는 값 마다 Input으로 들어온 NxN의 D1개의 dimension에 대해서 inputbase, weightbase를 계산하여 Convolution 연산을 진행한후 PE가 담당하고 있던 Output 위치에 값을 누적해 주는 것입니다. 이후, Bias까지 더해주고 ReLU 계산을 하여 넣어줍니다.

이단계에서는 딱히 시행착오가 없이 한 번에 되었고 **(약 687🡪 537초)**의 성능향상이 있었습니다.

이단계를 거치면서 대략적인 방향성을 잡게 되었습니다. 하지만 대략적인 병렬화를 진행했음에도 불구하고 성능이 좋게 나오지 않아서 실망했었습니다.

요약: 주어진 코드의 맨 바깥부분을 병렬화를 시켰습니다.

1. **2단계**: 1단계에서 루프안에서 사용했던 +=로 Global Memory에 필요없이 2번 접속해야 했기 때문에 이러한 오버헤드를 줄이기 위해 1단계 루프 내부에서 partialsum이라는 private 변수에 값을 먼저 불러와서 저장한후 sum(Conv연산 누적합)과 partialsum과 합쳐서 넣어줍니다.



이단계에서도 global memory를 여러 번 접근하는 것을 단순히 private memory에 저장한후 값을 넣는 것 이기에 시행착오가 없었고 성능향상은 **(약 537🡪524초)** 정도 되었습니다. 이단계에서 또한, 생각보다 적은 성능향상을 보였기에 실망했습니다.

요약: conv연산 내의 커널 코드에서 sum 및 partialsum에 대한 private 메모리사용

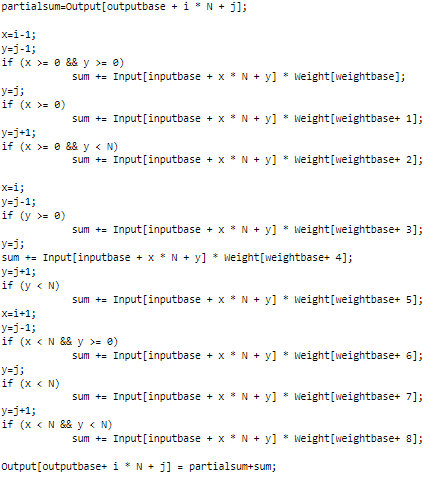
1. **3단계**: cnn\_seq와 같은 구조를 가져오면서 함수에서 네트워크에 대해서 계속 float\*로 입력을 받고 clCreateBuffer를 하여 메모리 복사에 의한 시간 지연이 발생하였기 때문에 네트워크를 전역변수로 clCreateBuffer를 통해 opencl에서 사용하는 버퍼로 만든 후 각각의 함수 안에서 사용할 수 있도록 하였습니다. 이러한 방법으로 중복된 일(고정된 네트워크값을 계속 만들어서 사용한 행위)를 줄임으로써 불필요한 데이터 복사를 여러 번 일어나지 않도록 하였습니다.

이단계에서는 네트워크를 적절하게 clCreateBuffer를 통해서 cl\_mem에 매핑했어야 했고 실수도 여러 번 하면서 진행하였지만, 꼭 이것을 해내야 성능에 유의미한 영향을 끼친다는 생각 아래 열심히 진행하여 마무리할 수 있었습니다. 이에 따라 성능향상은 약 **524🡪 462**초로 60초정도 단축하였습니다.

이단계부터 메모리 복사의 오버헤드에 대해서 생각을 많이 하게 되었습니다. 생각보다 많은 부분 시간을 차지하였고, 아무리 병렬화를 진행한다고 해도 Host에서 발생하는 지연에 대해서는 GPU와는 상관없었기에 반복 작업을 줄이는 것에 초점을 두었습니다.

요약: 네트워크(고정값)을 계속 clCreateBuffer해서 복사하였던 오버헤드를 줄임.

1. **4단계**: kernel Code내에서 Conv연산시에 존재하던 루프를(3x3) 푸는 방법으로 진행하였습니다.

이는 2단계에서 말씀드렸던 부분사이에 존재하는 루프를 단순히 인덱스 상으로 계산할 수 있도록 풀은 것이며 중간에 필요 없는 비교를 줄임으로써 **(462🡪460)**초의 약 2초의 성능향상을 일으켰습니다.

이를 진행할 때에 단순히 3x3루프를 코드로 풀어서 진행하면 성능향상이 존재할까 의문이 많았지만, 실제적으로 성능향상이 일어난 것에 대해 많이 신기한 경험을 체감했습니다. 이부분에서는 시행착오가 따로 없었고 조건 생각을 하면서 코드를 단순화했습니다.

이단계를 진행하기 위해서 좀더 강도높은 CNN에 대한 이해가 필요하였고 생각보다 높은 성능향상이 일어나지는 않았지만 Convolution 연산하는 곳의 구조를 깊이 이해하였다는 것에 만족했습니다.

요약: Conv커널 코드내에 있는 필터의 3x3루프를 제거함.

1. **5단계**: Conv연산시에 전까지는 GlobalWorkSize만 할당하여 진행하였지만, LocalWorkSize를 4정도로 준 후 실행하였더니 많은 성능 향상 효과가 있었습니다. 이때부터 단순히 GPU에 데이터를 넣고 GlobalWorkSize만 명시하는 것보다 얼마나 적절히 LocalWorkSize를 배정하여 진행하는 것이 성능향상의 지름길임을 알게 되었습니다.



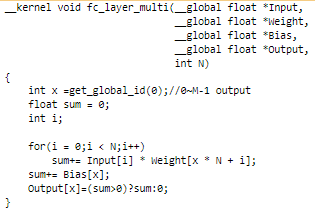
이 단계에서 많은 LocalSize들에 대해서 실험해 보았고 그 중에서도 좋은 성능을 내는 것을 찾아서 얻은 성능향상이 **460🡪84초**입니다.

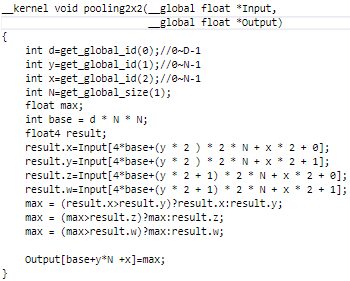
이때부터 적절한 localWorkSize를 찾는 것을 목표로 하였습니다. 비록 kernel코드 딴에서 알고리즘이 바뀐다고 하더라도 적절한 local사이즈를 주는 것이 성능향상에 많은 영향을 끼칠 것이라고 생각하게 되었습니다.

요약: 여러가지로 실험을 해보다가 localWorkSize를 적당히 줌으로써 좀더 좋은 성능을 내는 것을 확인함. 여러 실험을 진행하다가 가장 좋은 결과를 냈던 값으로 설정함.

1. **6단계**: Pooling Layer와 Fully Connected Layer를 병렬화를 하여 얻은 결과입니다. 생각 보다 많은 향상이 있지는 않았지만 순차적인 코드보다 추후 좀더 많은 성능향상이 기대되었기 때문에 CNN의 주요 연산인 Convolution, Pooling, Fully Connected 등을 모두 병렬화 시킨 결과입니다.

두개의 병렬화 전략 모두 단순하게 seq코드를 opencl용으로 바꾼 것입니다.

FC layer같은 경우에는 GlobalWorkSize를 저희가 원하는 Output크기, 즉 M으로 두고 각각의 Output에 Input과 Weight를 적절히 곱해서 누적한후 맨 마지막에 ReLU연산을 진행합니다. 각각의 PE마다 N번의 곱셈을 실행하고 N+1번의 합연산을 진행합니다.

Pooling에서는 Input으로 DxNxNx4를 받아 Output으로 DxNxN으로 내놓기 위해 clEnqueueNDRangeKernel 시에 입력받은 D와 N을 3차원으로 (D, N, N)으로 globalWorkSize를 주어서 GPU를 통해 계산하였습니다. 이를 통해 Output에 MaxPooling한 값을 저장할 수 있게 되었습니다.

이 과정을 하기위해 여러가지 실패를 거쳐가면서 결과적으로 병렬화를 성공하게 되었지만 생각보다 좋은 성능향상이 일어나지는 않았습니다. **84🡪82**초로 성능향상이 일어났습니다. 이때부터 중복 작업을 최소화하기 위한 BATCH도입을 생각하기 시작했습니다.

요약: Conv레이어 말고도 Pooling과 FC 레이어 또한 병렬화를 진행함. 조금의 성능향상이 있었고 특히 추후에 배치 작업을 진행할 때 좋은 성능을 낼 것 같아서 진행함.

1. **7단계**: Conv연산에서 LocalWorkSize를 Input의 크기에 따라 적절히 재배정한 결과 얻은 성능 향상입니다. 이는 5단계에서 진행하였던 LocalWorkSize를 함수에 들어오는 원하는 Output의 크기에 따라 다르게 설정하였더니 좀더 좋은 성능을 낼 수 있었습니다. **82🡪80초**

제 GPU에 맞는 Conv연산에 local사이즈를 찾는 과정이었습니다. 단순히 많은 실험 끝에 얻은 결과입니다.

요약: Conv레이어 내에서 LocalWorkSize 조정

1. **8단계**: 이전까지는 항상 input neuron 과 output neuron의 값을 구할 때 함수 내에서 float\*형으로 함수에 입력을 받은 후 함수내에서 clCreateBuffer를 통해 메모리 복사를 진행한 후 GPU로 보내고 이후에 연산이 끝나고 나서 다시 ReadBuffer를 이용해 해당 위치에 메모리를 복사하는 방식으로 진행하였습니다. 이러한 과정이 메모리 복사 오버헤드를 일으키고 있다는 것을 알고 있었기에, input과 output모두 cl\_mem으로 선언하여 해당 주소를 넘겨주어서 함수내에서 다른 메모리 복사과정이 필요 없이 clSetKernelArg 이후 clEnqueueNDRangeKernel를 통해 GPU로 바로 넘겨줘서 연산을 할수 있도록 하였습니다. 이를 통해 **80🡪76초**의 성능향상을 이뤄낼 수 있었습니다.

목표는 명확하였고 일은 단순하였기 때문에 시행 착오는 없었습니다만, 생각보다 이러한 메모리 복사에 의해 낭비되는 시간이 많았다는 것을 다시 한번 느끼고 좀더 반복되는 작업을 최소화해야 성능향상에 도움이 된다는 것을 다시 한번 체감하였습니다.

요약: cnn\_seq구조를 가져옴으로써 레이어 함수내에서 계속 clCreateBuffer를 통해 cl\_mem으로 만들고 Write하여 GPU로보내고 Read로 읽어오는 부분을 cl\_mem 데이터를 처음에 main에서 cnn함수를 부를 때 필요한 만큼 다 할당한후 진행하도록 하여서 메모리 복사 오버헤드를 줄임.

1. **9단계**: Conv연산 함수 내에서 LocalWorkSize를 다시 여러 실험을 통해 사이즈를 재배정 하였고 각각의 함수내에서 kernel을 새로 만드는 과정을 cnn\_init안에서 만든 후 각각의 함수내에서 계속 사용하는 방식으로 바꾸었더니 좀더 좋은 성능을 내게 되었습니다.

시행착오 또한 없었고 성능향상은 **76 🡪 72초**로 일어났습니다. 최대한 루프내에서의 중복작업을 줄이기 위한 노력을 했습니다.

요약: 함수(conv layer, Pool layer, fc layer)를 호출할 때 마다 안에서 새롭게 설정했던 kernel을 cnn\_init안에서 설정한후 cnn 함수 끝날 때 해제함으로써 필요 없는 중복 작업을 감소.

1. **10단계**: 배치시스템을 적용시켰습니다. 모든 Layer, 즉 Conv, Pool, FC에 대해서 배치크기 만큼을 한 번에 GPU에서 수행하도록 하였습니다. 이를 통해, 좀더 효율적인 메모리 복사와 중복작업 제거가 가능하였고 그에 따라 **72🡪43**초의 성능향상을 이끌어 낼 수 있었습니다.

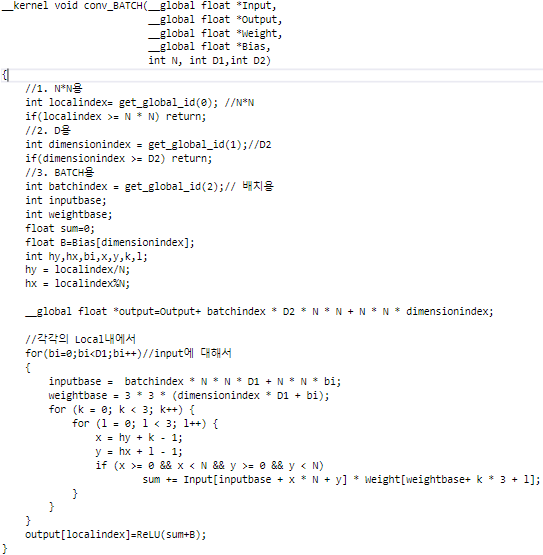
이 과정에서 여러 시행착오가 있었습니다. 주로 모든 Layer가 배치단위로 연산을 수행하도록 만들지 않았기 때문에 커널 코드부분부터 모두 다 배치크기만큼 연산을 할 수 있도록 바꿔야만 했습니다.

그래도 알고리즘은 앞에서 구현하였던 것 그대로 배치관련 하여 조금의 변화만 필요하였고 이를 통해 생각보다 좋은 성능향상이 있었습니다. 특히 배치의 가장 효율적인 크기를 찾기 위해 여러 번 실험을 진행하였는데 250정도의 크기가 그래도 가장 좋은 성능을 내는 것을 확인하여 차용하였습니다.

요약: 전까지는 하나의 이미지마다 주어진 네트워크를 흘려보내면서 결과를 내었지만, 좀더 효율적인 방안을 모색하던 도중 배치화를 시켜보자는 생각이 들어서 단순히 전에 커널로 구현한 모든 함수들을 배치화를 가능한 구조로 바꿈.

1. **11단계**: Conv연산에 들어가는 메인 알고리즘을 바꿨습니다. 지금까지 Conv함수를 부를 때 입력으로 들어온 Input: D1xNxN, Output: D2xNxN에 대해 GlobalWorkSize를 D2만큼 설정한후 한 PE에서 D1xNxN연산을 진행했는데, 조금 더 효율적인 일 배분을 위해서 GlobalWorkSize를 (NxN, D2, BATCH)로 주어서 각각의 Output에 해당하는 위치 마다 Conv연산을 진행하는 방향을 사용하여 매우 좋은 성능을 낼 수 있게 되었습니다. [즉 각각의 PE는 D1x9+1(bias)의 작업을 하게 됩니다]

이 과정에서 여러가지 우여곡절들을 겪으면서 제가 현재 실험을 진행하고 있는 GPU에서 가장 좋은 성능을 내는 localworksize를 찾는 것에 많은 시간을 들였고 여러 실험 끝에 제 GPU에 적절한 localworksize를 찾는데 성공하였습니다.

특히 전체적인 메인 알고리즘을 바꿀 때 Weight계산할 때 batchindex 또한 포함시켜서 계산해서 몇시간 동안 고민했는데, 생각해보니 weight는 batch와 관계없다는 것을 생각하고 고치자 마자 잘 돌아가는 것을 보고 허탈했습니다.

시간향상은 **43🡪4초**입니다.

요약: 전에는 D2개의 일을 각각의 PE가 맡아서 D1xNxNx9 + NxN번의 연산을 한 구조를 각각의 PE가 D1x9 + 1번의 연산만 진행하도록 구조를 바꿈

1. **느낀점**

너무나도 재밌게 진행한 프로젝트였습니다. 특히 항상 관심이 많았던 CNN에 대해서 OpenCL을 이용하여 추론과정에 대해서 이렇게 가속화 해본 것이 처음이었고 GPU를 사용하는 것만으로도 680초의 Sequential 일이 4초만에 가능한 것이 놀라웠습니다.

또한 이번 프로젝트를 진행하면서 제 한계를 많이 느꼈습니다. OPENCL과 관련 내용들에 대해서도 인터넷을 많이 찾아보면서 Local Memory를 사용하여 좀더 좋은 결과를 내고 싶었지만 시간의 부족과 능력의 부족으로 제가 현재 할 수 있는 범위안에서 최대의 결과를 낸 것 같습니다.

학교에서 진행한 프로젝트 중 초반에 방향성을 가장 잡기 힘든 프로젝트였던 것 같습니다. 전체적으로 멀티코어 프로그래밍에 대한 내용뿐만 아니라 OPENCL에 대한 예시 코드들도 구하기 어려워서 기초 과정에서 가장 어려운 길을 지나가게 된 것 같습니다.

교수님께서 항상 하시던 말씀인 어려운 만큼 남들도 이 경지에 도달하려면 노력을 해야 한다는 것을 마음에 안고 여러가지 방향성을 추구하면서 공부한 결과 비록 최상의 결과는 아니지만 제가 만족할 만한 결과가 나온 것 같습니다. 하지만 타일링 기법을 시도하였지만 안된 것이 너무 아쉽습니다. 만약 잘 진행이 되었으면 좀더 만족할 만한 결과가 나왔을 것 같아서 슬픕니다. 그럼에도 불구하고, 많은 절망과 많은 시도를 하였고 노력한 만큼 결과가 나와서 다행인 프로젝트였습니다. 한학기동안 멀티코어를 배우면서 너무 재밌었습니다.