병렬 프로그래밍

Final Report

Team\_B

20135159 이준범

20125189 양형모

20145101 고동환

20143209 김혜민

**연구 목표**

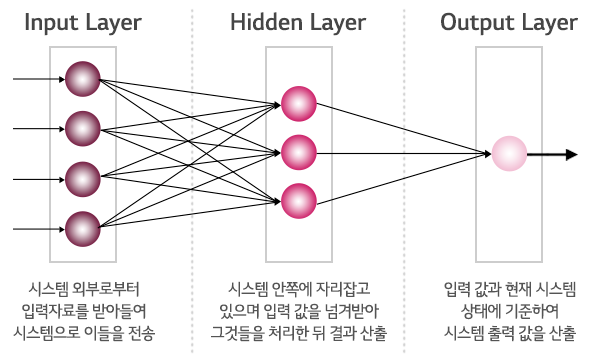
**CUDA를 활용한 Convolutional Neural Network (CNN) 성능 향상 기법 분석 및 성능평가**

**문서 개요**

1. **GPU 코어에 대한 이해와 CUDA 동작 방식**
2. **인공 신경망(Neural Network)**
3. **합성곱 신경망(Convolution Neural Network)과 구조**
4. **C 로 구성된 CNN 코드 분석**
5. **C 코드 -> Cuda 코드로 변환 (병렬 처리)**
6. **성능 비교**
7. **인공 신경망(Neural Network)이란 무엇인가?**

**시냅스의 결합으로 네트워크를 형성한 인공 뉴런( 각각의 노드 )이 학습을 통해서 시냅스의 결합 세기를 변화시켜 (더 적합한 node쪽으로 가중치가 커짐) 문제 해결능력을 가지는 비선형 모델(각각의 가중치가 다름)**

**1)Input 과정에서 시스템 외부로부터 입력자료를 받아들여서 시스템으로 전송하면 2)시스템이 입력 값을 넘겨받아서 처리 후 결과를 산출 3)Output 과정에서는 현재 시스템 상태에 기준, 출력 값을 산출한다.( 결과 내는 과정 -> 데이터 분류로 나오는 결과 값)**



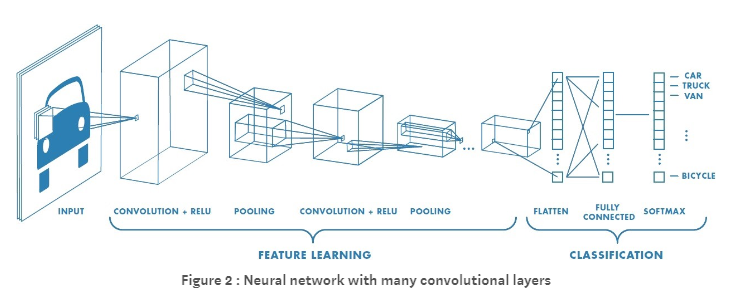
1. **합성곱 신경망(Convolution Neural Network, CNN)**

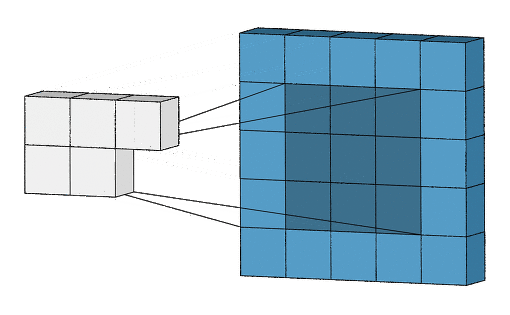
**합성곱 신경망은 합성곱+신경망의 합성어로 딥 러닝에서 이미지 인식에 많이 사용되는 기법.**

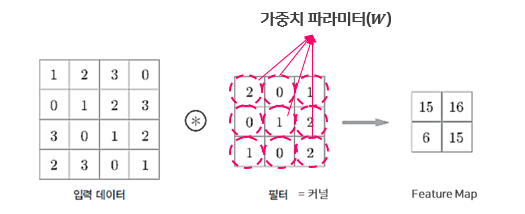
**원본 데이터를 받아서**

* **합성곱 층(Convolution Layer)**

**합성곱층(Convolutinal Layer)에서 기존의 특징 filter를 적용, subSampling(일반적으로 CNN에서는 Max pooling을 사용)과정을 반복적으로 수행해서 마지막으로 활성 함수를 적용해 특징(Feature)을 뽑아내는 일련의 과정을 거친다.**







**합성곱 과정에서 전체 입력데이터의 모든 픽셀에 연결하는 것이 아니라 합성곱 뉴런 수용영역안에 있는 픽셀에만 연결하기 때문에 저 수준 특징에 집중할 수 있고 다음 합성곱층으로 넘어갈수록 고수준의 특성 조합으로 나아가게 된다. 또한 3차원 이미지를 그대로 입력 데이터로 받기 때문에 제대로 학습할 가능성이 높다고 할 수 있다.**

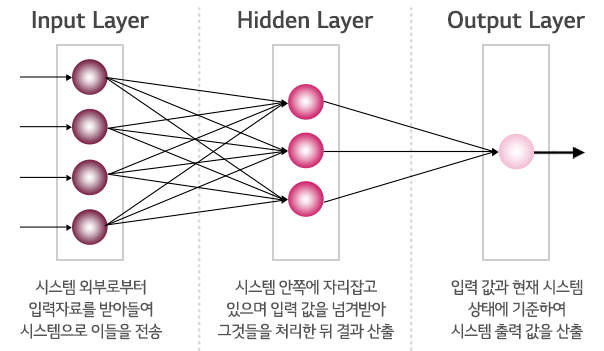
* **특징 데이터 (Feature)**

**Convolution Layer를 거친 이미지 데이터들은 특징 데이터로 분류 되며 이를 다시 Neural Network(인공 신경망)의 입력 값으로서 사용이 되어 분류 작업을 거쳐 최종 데이터를 산출하게 된다.**

* **신경망 층(Neural Network)**

**Convolutional Layer를 거치고 활성 함수 적용을 통해서 추출한 특징 데이터들을 분류하는 과정 복잡한 신경망 구조로 처음에는 랜덤한 가중치를 갖고 데이터를 분류하지만 결과에 따라 가중치가 달라진다.**

**(활성 함수가 적용되면 Convolutinal Layer에서 결과로 나온 Matrix들이 실수 값으로 변경된다. 이 데이터를 가지고 가중치를 주면서 분류하는 과정)**

****

**이러한 일련의 과정을 CNN이라고 하며 마지막 신경망 층까지 거치고 나온 데이터들을 통해서 입력 데이터(원본 이미지)의 특징을 추출하는 것이다.**

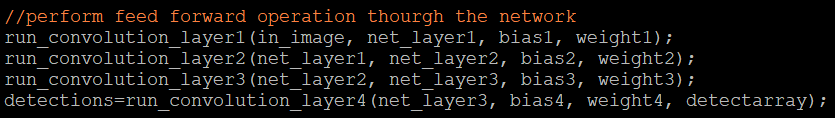
1. **C로 구성된 CNN 코드 분석**

C로 구성된 코드 내의 함수 헤더이다.

각각의 함수에는 함성 곱 연산이 진행이 된다.

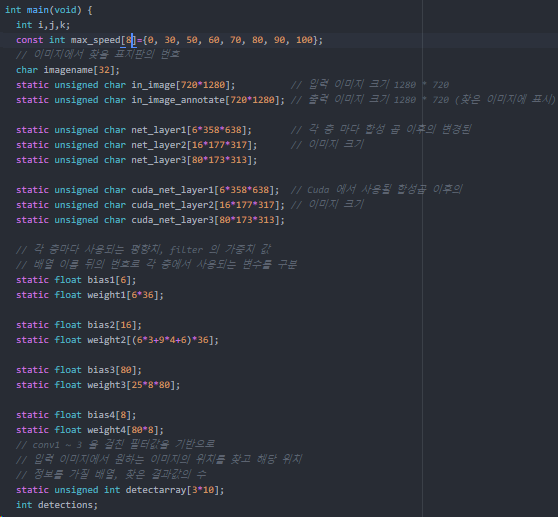
Conv 1 ~ Conv3 에서는 Input\_image(찾고자 하는 이미지 원본 OR Conv함수를 통과한

이미지) filter(찾고자 하는 이미지의 특성 값)의 합성 곱이 진행이 된다.

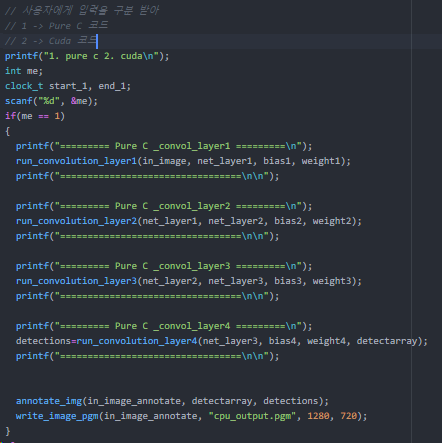


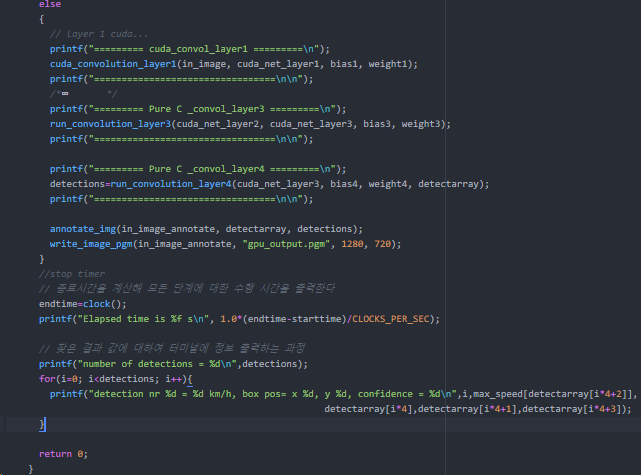
본 코드에는 완성된 모델(머신러닝)이 사용 되므로 함수 연산 과정에서 사용되는 상수 값은 모델의 인자 값이다.

CNN – Pure C 코드 전체









구현되는 main 코드의 모습이다.

각각의 영역마다 주석으로 내용 설명을 첨부 하였으며

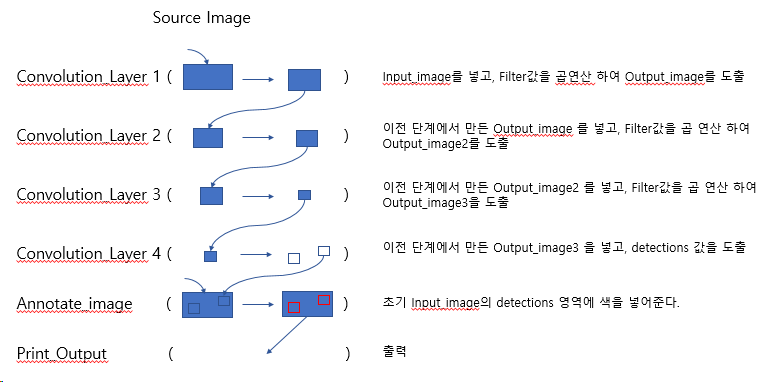
실제로 작업(CNN) 진행 되는

convolution layer 1 ~ 4 의 함수에 대해서 자세히 설명 하겠다.

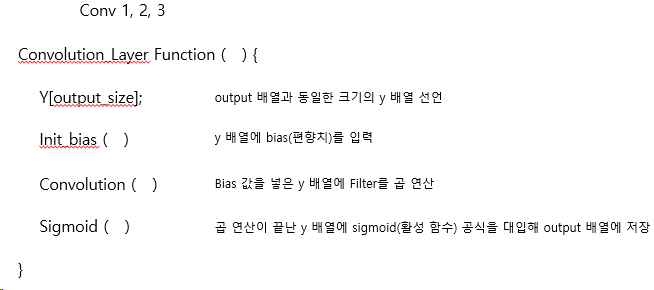
설명 하기에 앞서 main 코드가 구현 되는 과정

Convolution 함수가 내부의 구조를 도식화한 이미지를 확인해 보자.

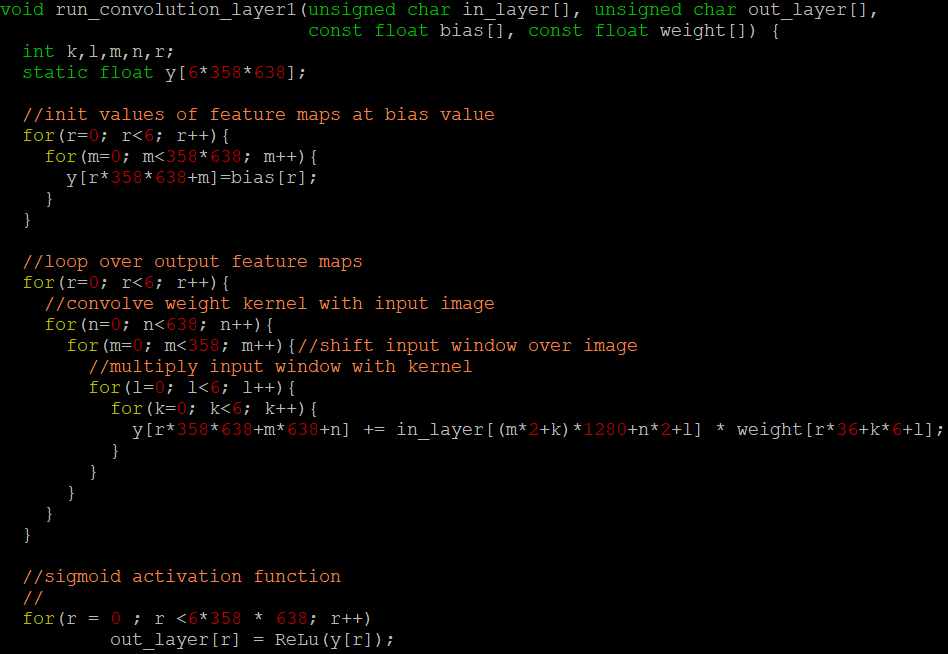
Main 구성도



Conv 1, 2, 3 함수 내부 (요약 내용이며, 세부 내용은 조금씩 차이가 있다.)



Using Pure C – Layer1



위에서 순서 대로

initBias () -> convolution () -> Sigmoid() 과정이 진행 됨을 볼 수 있다.

하나하나 세부적으로 알아보자.

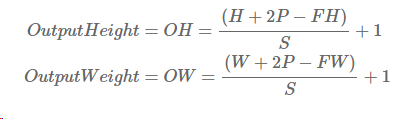
InitBias()

기존에 완성된 머신러닝 모델의 편향치를 연산에 사용하기 위하여 초기화 작업



초기화에 사용되는 Bias 의 값이다.

Y 의 배열의 크기는 공식에서 도출된 크기이다.



위의 사진에서 OH | OW 는 출력되는 이미지의 너비와 높이이며

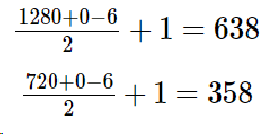
H | W 는 입력 이미지의 높이와 너비 이다.

FH | FW 는 필터 이미지의 높이와 너비 이다.

P 는 패딩의 크기 이며 패딩에 대한 설명은 문서 초반을 참조하길 바란다.

S 는 스트라이드의 값이며 패딩과 마찬가지로 설명은 위와 같다.

위의 공식을 적용 해보자.

패딩은 사용 되지 않기 때문에 0이 대입 된다.

스트라이드는 2로 진행

필터의 크기는 6 x 6이다.

이렇게 공식에 대입하여 638 X 358이라는 출력 이미지의 크기가 도출 된다.

하지만 (638 x 358) 이라는 이미지 크기 이외에

X 6 이 수행됨을 볼 수 있는데

이 이유는 연산에 사용되는 필터의 개수가 6개 이기 때문에 출력 이미지 개수도 6개가 되기 때문이다.

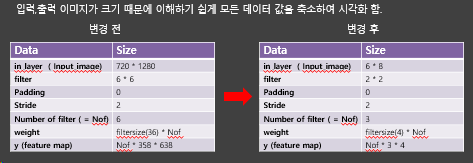
Convolution()

실제로 중요한 연산이 진행 되는 과정이다.

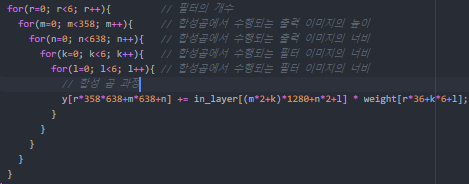
위의 함수는 1280 x 720 (입력 이미지) 와 6 x 6 (필터 이미지) 의 합성 곱으로 진행이 된다.

이는 수가 너무 커지기 때문에 구동 과정에서의 직관성이 떨어진다 확인하여

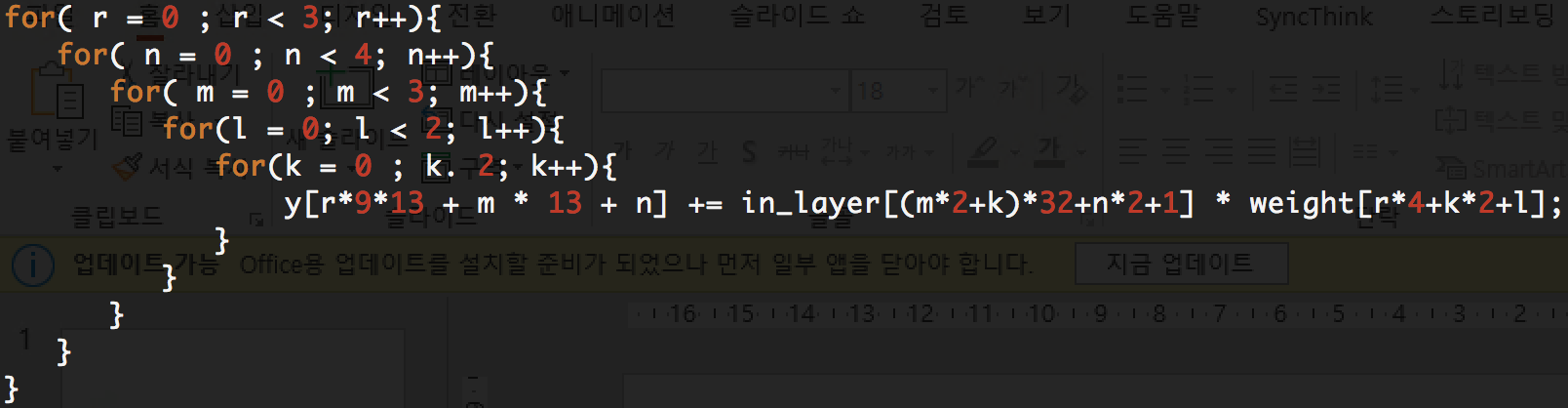
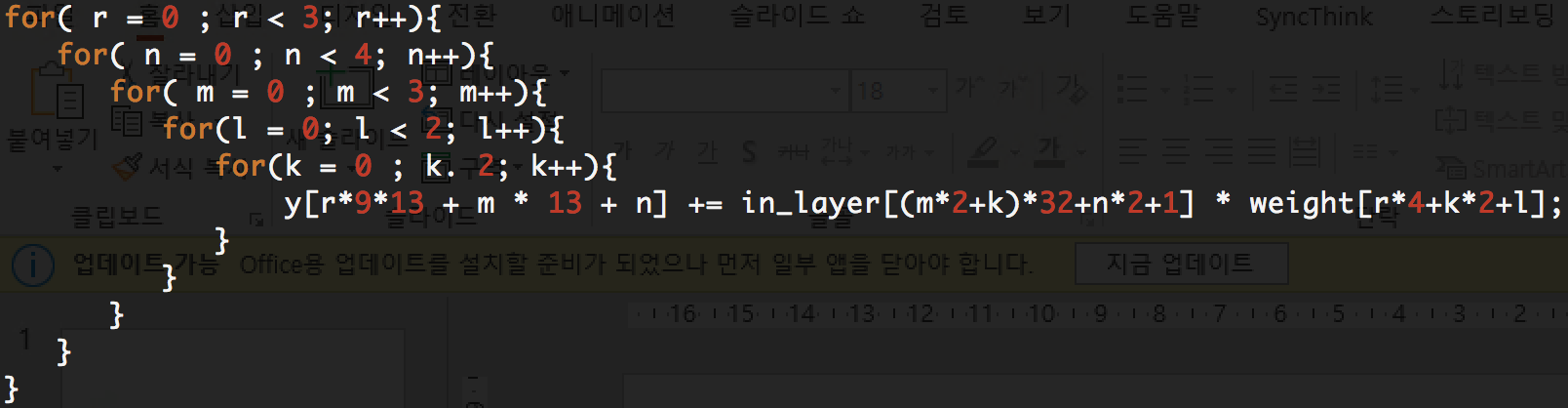
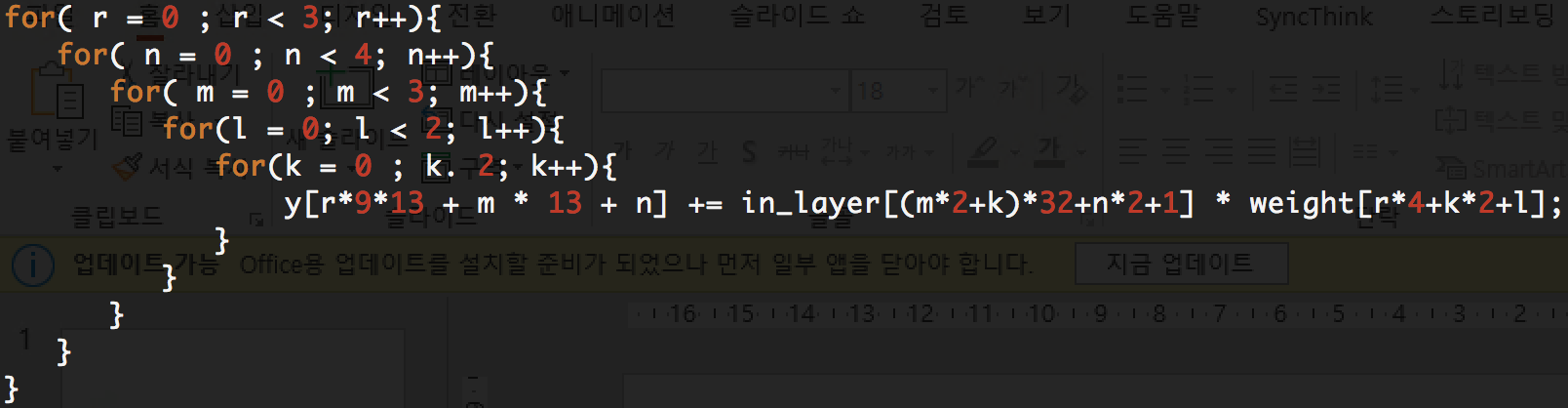
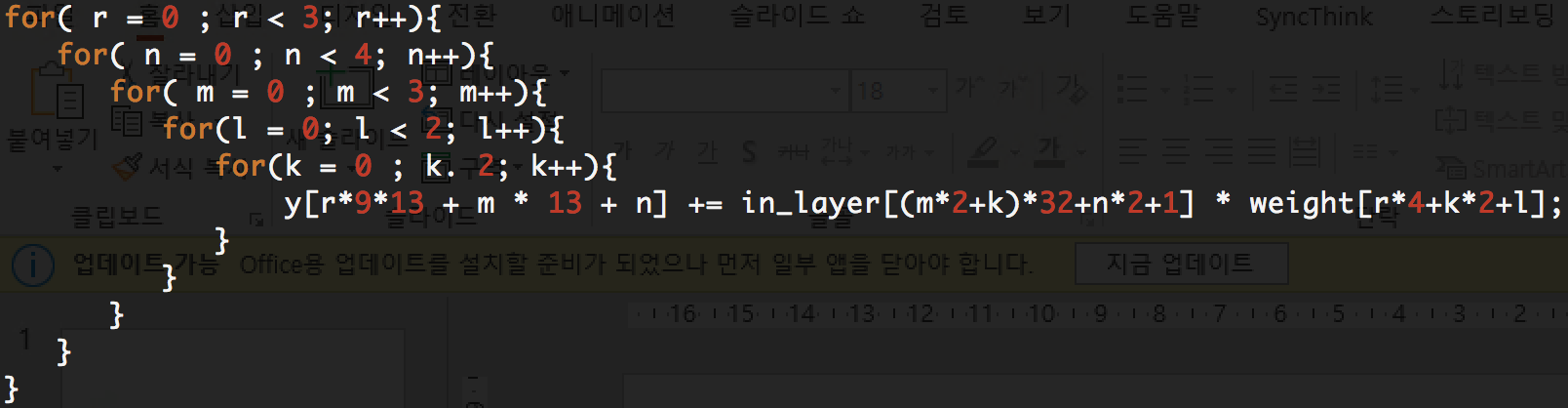
위의 과정을 분석 하기 위하여 입력 이미지와 필터 이미지를 축소 시켜 보았다.



기존 Convlution



이미지 축소에 따른 변경된 Convlution



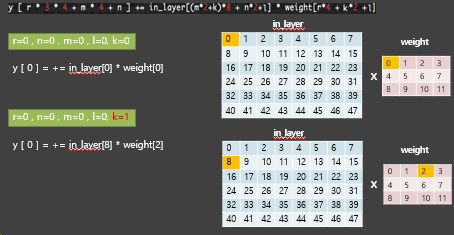
For 문에 사용되는 변수를 변경하며 과정을 확인해 보자

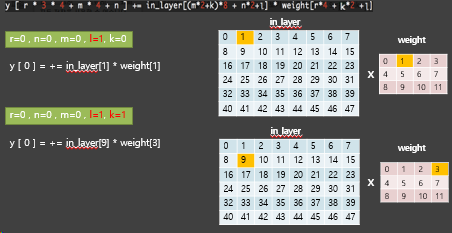
가장 안의 for문 변수인 k를 수정 했을 때이다.

여기서 이미지의 weight 은 0123 -> filter 1 | 4567 -> filter2 8,9,10,11 -> filter3이며

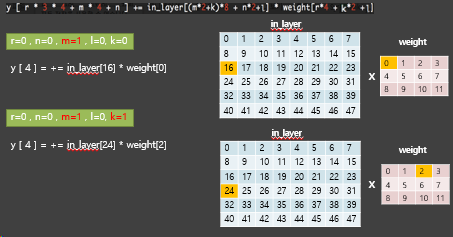
각각의 filter는 2 x 2 로 0 1 으로 구성 된다.

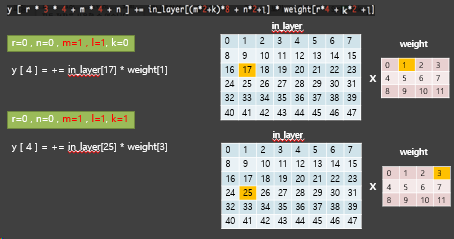
2 3

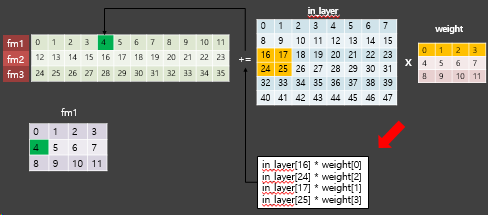


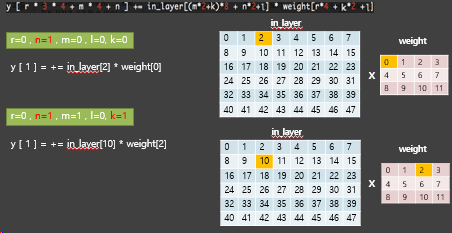


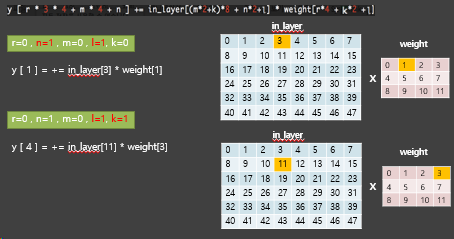


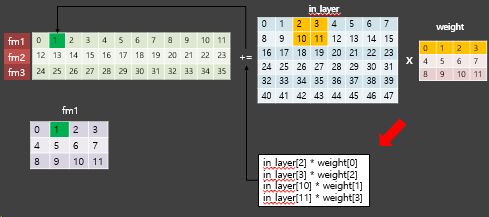












위의 이미지 진행 되로 연산은 수행된다.

해당 과정으로 연산이 모두 수행된 y[6 x 638 x 358] 의 배열은 이후

Sigmoid() 비선형 데이터화 과정을 거치게 된다.

Sigmoid()



해당 부분은 단순하게 배열 안의 모든 값을 함수(sigmoid)를 거친 값(비선형 데이터) 로 변환 하는 작업으로 적중률을 높이는데 사용이 된다.

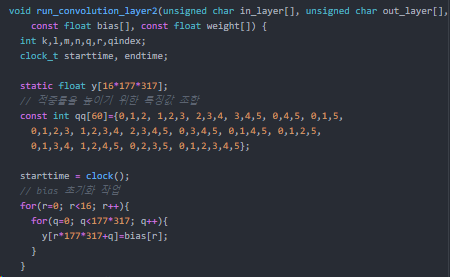
이렇게 convolution1() 함수는

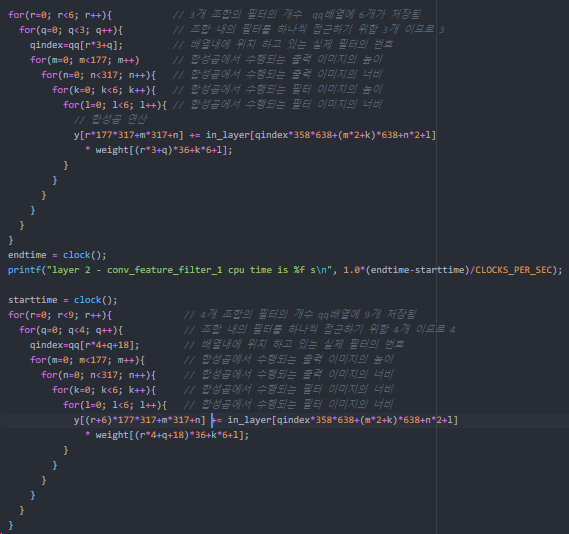
InitBias() -> convolution() -> sigmoid() 이 3가지가 기본적으로 수행이 된다.

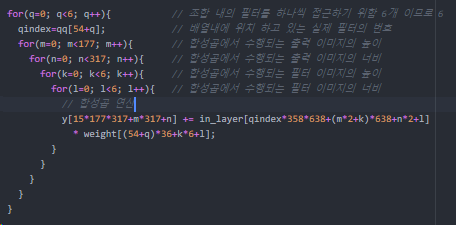
본 층은 단순하게 입력 이미지에 필터 이미지를 합성 곱 만을 진행하는 층이기 때문이다.

하지만 2, 3 층의 경우 모델의 적중률을 높이기 위해 convolution() 단계가 조금 독특하게 진행이 된다. Initbias()와 sigmoid()는 동일한 과정이 수행. 다른 부분인 convolution() 부분만 알아보자.

Using Pure C – Layer2



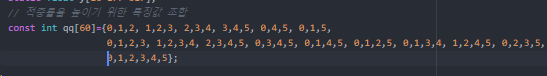






Convolution2 의 경우 3개의 연산으로 나뉘어 진다.

이유는



여기에 있다.

Convloution1 을 통과한 우리는 현재 6개의 결과 이미지가 존재한다

이는 0 ~ 5의 인덱스를 가지고 있으며 conv2 에서는 높은 분석률을 위해서 특성 조합을 사용하여 연산에 적용 한다.

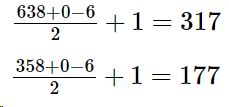
이 조합은 필터 3개, 4개 6개의 조합을 이용하여 합성 곱을 진행한다.

조합을 위해서 qq 배열을 선언하여 내부에는 실제 인덱스 값 을 저장 해 놓은 모습이다.

배열 저장 모양은 직관성을 위해 각 층마다 3조합, 4조합 6조합으로 나눈 모습이다.

조합을 통해서 완성되는 convolution 2 층의 결과 사이즈는 아래와 같은데

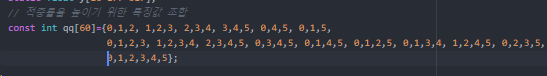
이는 1층과 마찬가지로 공식을 통해 도출된 크기이다.



사이즈는 공식을 통해 이유를 확인 했다.

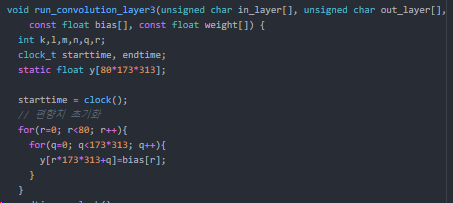
그러나 왜 결과 이미지의 개수는 16개 인가???

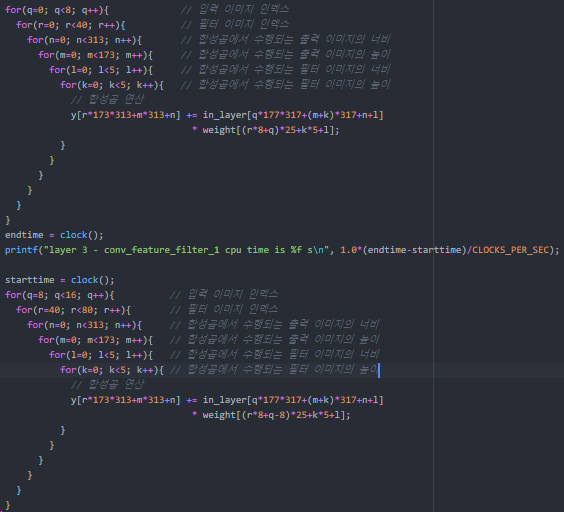
이는 조합을 통해 완성되는 개수가 16 개 이기 때문이다.



해당 이미지의 그룹을 확인해보면 16개이며 위의 그룹에 따라 합성 곱이 진행 되었기 때문에 결과 이미지가 16개가 생성되는 것이다.

Using Pure C – Layer3

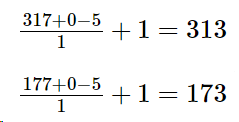






Convolution 3 층은 합성 곱 과정이 최종적으로 마무리 되는 과정이다.

기존의 필터 이미지가 6 x 6 크기 였다면 여기서는 5 x 5 로 변경되어 진행 된다. 또한 스트라이드 도 2 -> 1 로 변경되며 그렇기 때문에 결과 이미지 크기를 계산하는 과정에서 큰 변화가 생긴다.

 분모가 되는 스트라이드가 2 에서 1로 변경 됨에 따라

입력 이미지의 절반이 되던 출력 이미지가 큰 변동이

생기지 않음을 확인 할 수 있다.



위에서는 출력 이미지가 173 x 313 이 되는 근거를 알아 보았다.

그렇다면 왜 80은 곱한 것일 까???



이는 합성 곱이 진행되는 filter (weight)의 크기에 있다.

상수에 대하여 하나씩 분석 해보자.

먼저 25는 변경된 필터의 크기인 5 x 5를 의미한다.

8은 분석해야 할 표본의 개수이다.

결론적으로 해당 시스템은 표지판의 이미지를 출력하는 것이기 때문에 표지판으로 존재하는 위의 숫자의 정보를 출력하는 것 이기 때문에 main에서 선언된 8개의 표본을 기준으로 한다.

마지막으로 80은 각각 표본(각 번호판) 에 대한 예시 데이터 이다.



위의 이미지에서 1 ~ 9 까지의 필기체를 표현 하는 이미지가 여러 개 있듯이

필터 이미지로 사용 되는 각 표지판의 다른 이미지들이 80개씩 존재 한다는 의미이다.

For 문을 두 층으로 나눈 이유는 0 ~ 8 (표본 개수) 에 비해 입력 이미지로 들어온

(16 x (638 x 358)) 에서 배열 인덱스가 벗어나지 않게 하기 위함에 2층으로 구분 지어 연산을 진행 한다.

마지막으로 convoution4 () 이다.

앞서 설명 했듯이 합성 곱은 convolution3 에서 연산은 마무리가 된다.

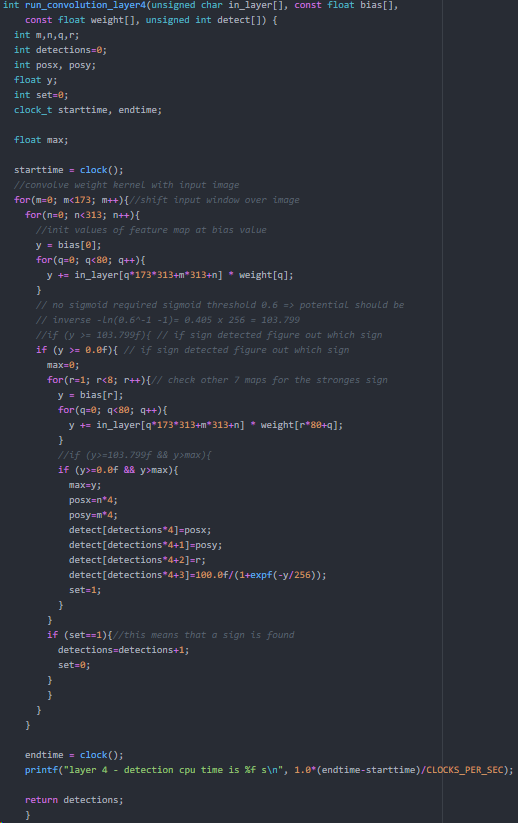
그렇다면 convoluion4 () 에서는 어떤 작업을 수행 하는 것인가.

CNN 은 convolution layer 와 neuralNetwork의 과정이 합쳐진 알고리즘이다.

1 ~ 3층의 내용이 이미지 합성곱을 통해 특징 값들을 도출 했다면

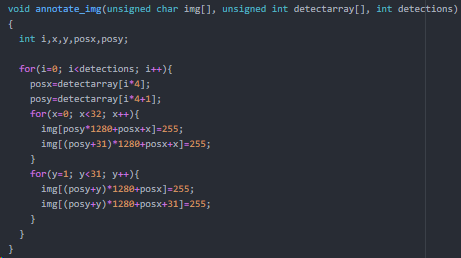
Convolution4 에서는 완성된 특징 값들을 바탕으로 여 실제 사용자가 찾고자 하는 이미지를 찾는 과정이다.

Using Pure C – Layer4



표시된 영역에서 스코어로 사용되는 y 의 값을 기준(찾고자 하는 이미지) 에 만족 할 경우 해당 영역을 detect배열에 위치를 저장 하는 과정이다.

위의 함수를 통과하면 완성된 필터 결과 값을 통해 초기 입력 이미지인 1280 x 720 의 표지판 위치(posX, posY, 표지판 내용(제한 속도), 예측한 점수(신뢰율) )의 정보가 저장된다.



Convolution 1 ~ 4 층을 모두 통과하여 detect(모든 정보가 담긴 배열) 을 기반으로 입력 이미지

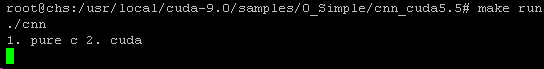
에서 위치를 찾아 테두리를 그리는 과정이다.

해당 함수(입력이미지에 표시하는 과정)까지 진행되면 모든 과정이 마무리가 된다.

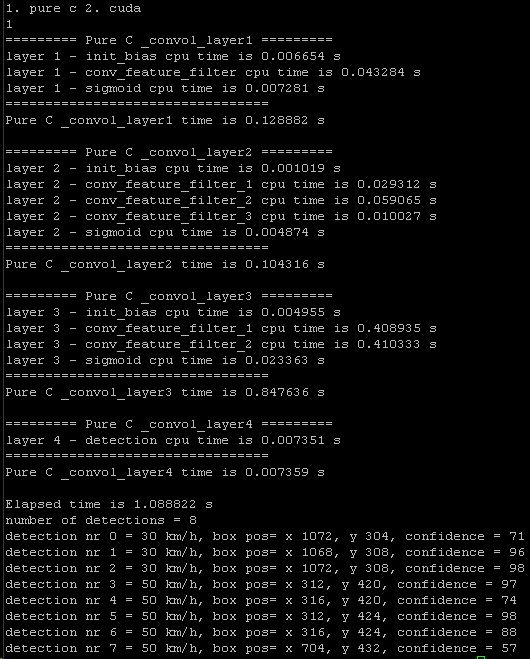
그렇다면 실제 구동 해보자.

시스템 수행 시

사용자 입력을 통해 CPU 연산 CUDA 연산을 구분 한다.



1 입력하여 C 코드의 수행 시간을 알아 보자.



각 층마다 그리고 함수 내부에서 InitBias(), Conv(), Sig() 함수 수행 시간을 측정해 보았다

이미지를 비교해 보자.

우선 입력 이미지이다.



위 이미지에서 표지판을 검출 후 이미지



찾은 임지 위치에 테두리가 그려짐을 확인 할 수 있고

터미널에서는 해당 이미지의 정보를 확인 할 수 있다. 그 정보는 터미널 캡쳐 화면의 빨간 테두리와 같다.

하나의 이미지에서 몇 개 없는(? 8개)를 검출 하는데 총 1초가 소요 됨을 확인 할 수 있고

내부 코드 또한 최대 많으면 5중 반복문이 여러 개 수행 됨을 확인 할 수 있다.

이미지 에서도 이런 소요 시간이 발생 한다면

동영상(많은 이미지가 여러 개 반복되는) 에서의 특정 이미지 검출은 더 많은 시간이 소요 될 것이다. 이런 소요 시간을 해결 하기 위해서는 CPU 연산이 아닌 GPU연산을 통한 병렬 처리가 해결 답안이 된다.

convolution에서 핵심 연산은 각각의 메트릭스를 회귀 하며 서로 연산 하는 데에 있다.

이러한 과정에서는 순서가 중요하지 않고 최종 결과 값이 중요하다.

이 말은 즉, 해당 과정이 순차적이 아닌 병렬적으로 동시 수행 되어도 문제가 없다는 의미이다.

그렇기 때문에 해당 내용은 Cuda를 통한 병렬 처리에 적합한 과제 이다.

본 연구 목표는 GPU를 활용한 CNN 가속화 이다.

위의 내용 까지는 CNN이 실제로 어떻게 구동이 되는지를 파악 하였으며

이 과정에서 수많은 반복문이 중복되는 모습을 설명을 통해 알아 볼 수 있었고

실제 수행 시간(많은 소요시간?) 또한 확인 할 수 있었다.

그렇다면 실제로 GPU를 활용한 병렬 코드로 작성 하였을 때의 소요시간 변화를 확인 해보자.

그전에 Pure C 코드 -> CUDA 코드 에서 적용된 내용을 먼저 확인 해 보자.

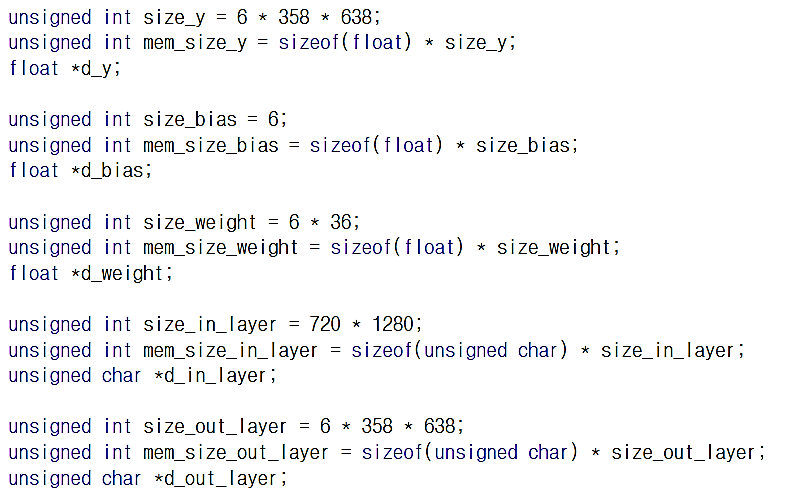
Cuda 코드

Cuda code는 1층만 구현되었다.

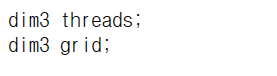
동작은 이하와 같이 진행된다.

먼저, 각 변수와 배열을 선언한다.

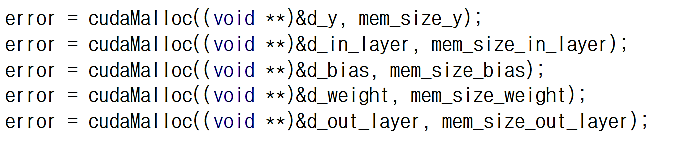
사용할 배열들의 사이즈와 배열을 선언 하고,



3차원의 쓰레드와, 그리드를 선언한다.



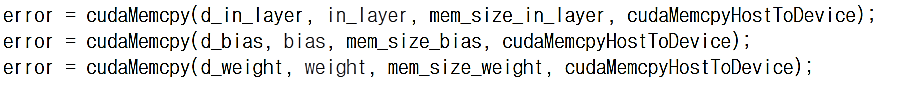
그리고 선언한 배열들을 동적할당 해주는데, cuda에서 동적 할당을 할 때는 c와달리 cudaMalloc 이라는 cuda 고유의 함수를 사용한다.



이어서 매개변수로 넘겨 받은 in\_layer와 bias, weight 값을 동적할당한 배열에 복제하는데, 이때도 cudaMemcpy라는 cuda 고유의 함수를 사용한다.

cudaMemcpy에는 4번째 옵션으로 Host 배열의 값을 cuda 배열로 보내는 cudaMemcpyHostToDevice와 cuda 배열의 값을 Host 배열로 보내는 cudaMemcpyDeviceToHost가 있다.

여기서는 cudaMemcpyHostToDevice 옵션을 사용한다.



그리고 배열을 복제하는 동안 cudaDeviceSynchronize() 라는 동기화 함수를 사용해 복제가 끝날 때까지 진행을 멈춘다.



복제가 끝나면 다음으로 d\_y 배열에 bias 값을 입력한다.

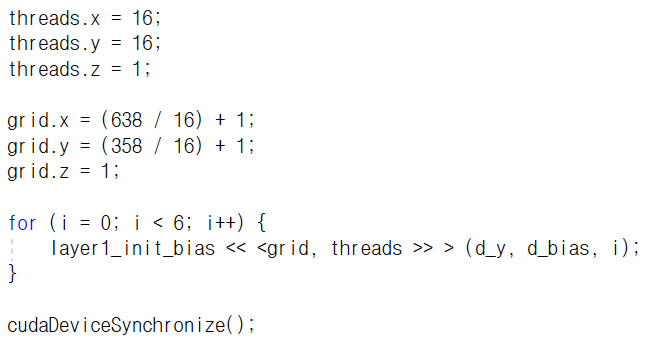
Threads와 grid는 각각 3차원으로, 임의로 x축 y축, z축을 설정한다.

여기에선 threads의 x축 y축을 16,16으로 한 블록당 256개로 했고, grid의 x축 y축은, Output image의 가로 세로 사이즈에 threads의 x축과 y축을 나눈 값을 입력한다.

이때, +1 연산을 하는데, 가로 세로 사이즈에 threads의 x축, y축 값을 나눴을 때 소수점이 무시되어 output 배열의 크기보다 쓰레드의 개수가 적어지게 되어 +1을 주어 사이즈를 늘린 것이다.

6개의 출력 이미지를 만들기 위해 6번의 반복을 하며, layer1\_init\_bias 커널을 병렬로 호출하여 사용한다.

커널이 동작되는 도중에 진행되면 다음에 이어질 동작에서 bias 값이 채워지지 않은 배열을 사용하게 되는 경우가 생길 수 있어, cudaDeviceSynchronize() 동기화 함수를 통해 동작이 수행되는 동안 진행을 멈춘다.



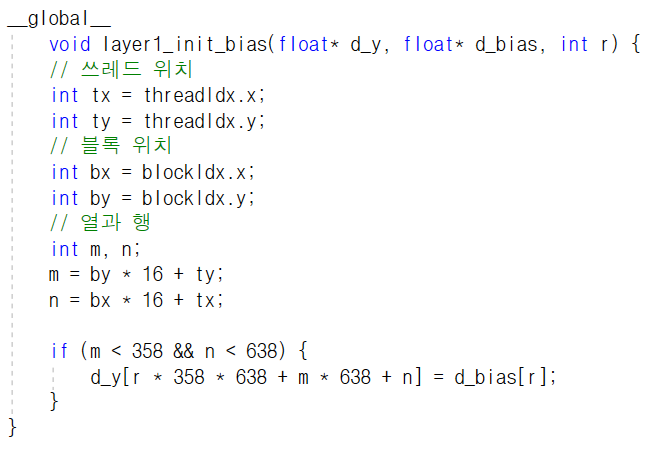
아래는 Layer1\_init\_bias 커널의 내용이다.

이 커널에서는 매개변수로 넘겨받은 d\_y 배열에 bias 값을 입력하는 동작을 수행한다.

쓰레드는 각각 16개가 넘어왔고, 블록은 bx가 40개, by가 23개가 넘어온다.

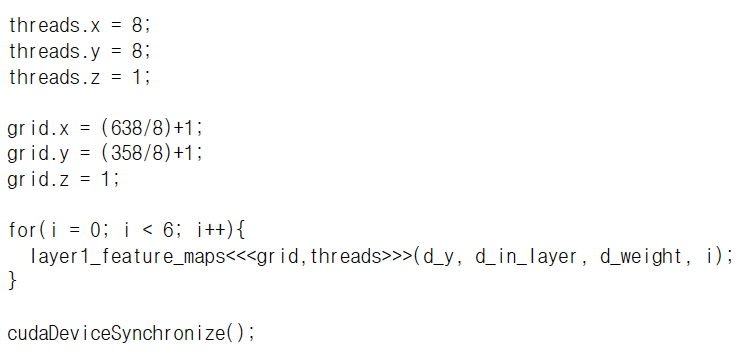
m과 n은 열과 행인데, gird를 초기화 할 때 +1을 해준만큼 최대 값이 본래 규격인 638과 358보다 크게 된다.

그래서 if (m < 358 && n < 638) 이라는 조건을 주어 d\_y 배열의 크기를 벗어나지 않게 하며, bias 값을 입력한다.



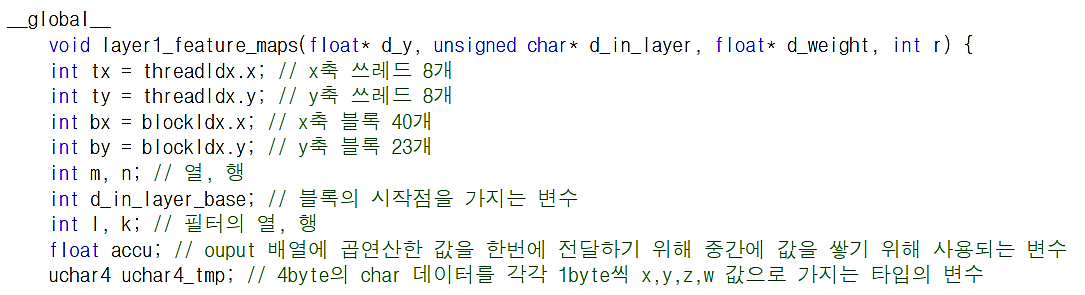
다음은 곱연산을 수행하는 영역이다.

Threads를 각각 8로, 그만큼 gird에 8을 나누며, 마찬가지로 소수점이 무시되기에 +1을 해준다.



layer1\_feature\_maps 커널을 보면,

먼저 아래와 같이 변수를 선언하며 시작한다.



C에서와 달리 이곳에서는 s\_in\_layer와 s\_weight이라는 두 공유메모리를 사용한다.

공유 메모리는 블록 내에 있는 메모리이다. 공유 메모리를 사용하면 같은 블록 내의 쓰레드들이 공유 메모리에 접근해 필요한 자원을 사용할 수 있게 되어, 매번 글로벌 메모리에 접근할 때와 현저한 속도 차이를 보인다.

layer1\_feature\_maps 커널에서 공유메모리를 사용하는 방식을 보면.

먼저,

\_\_shared\_\_ float s\_weight[6\*6];

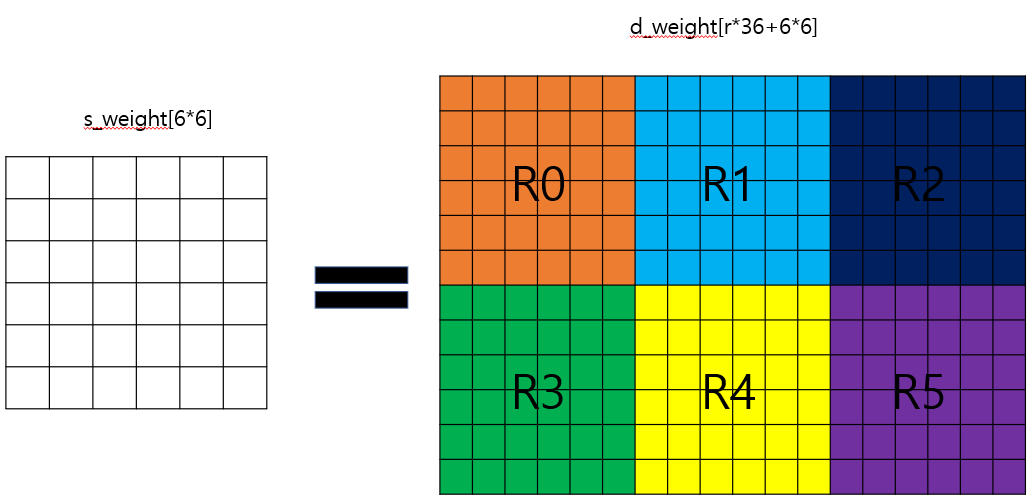
으로 6\*6의 크기를 가지는 필터값을 저장할 공유메모를 선언한다. 그리고,

if(tx<6 && ty<6){

s\_weight[ty\*6+tx] = d\_weight[r\*36+ty\*6+tx];

}

이런 식으로 매개변수로 넘겨받은 r번째 필터를 저장한다.



s\_in\_layer의 선언은

\_\_shared\_\_ unsigned int s\_in\_layer[(20) \* (20)]; 이렇게 된다.

크기 20은 계산이 완료된 값이고, 풀어서 쓰자면 이렇다.

\_\_shared\_\_ unsigned int s\_in\_layer[(8\*2-1+6-1) \* (8\*2-1+6-1)];

여기서 8은 블록 내 쓰레드의 한 축 길이이다.

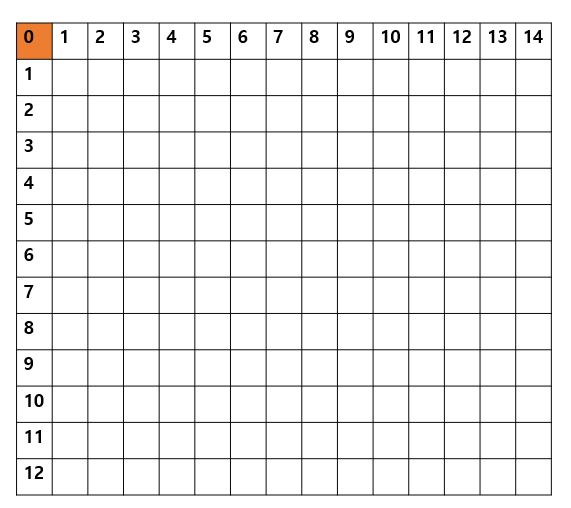
그리고 2는 stride값이고,

6은 Filter의 한 축 길이이다.

그렇다면 두 -1은 어디서 나오는 것일까?

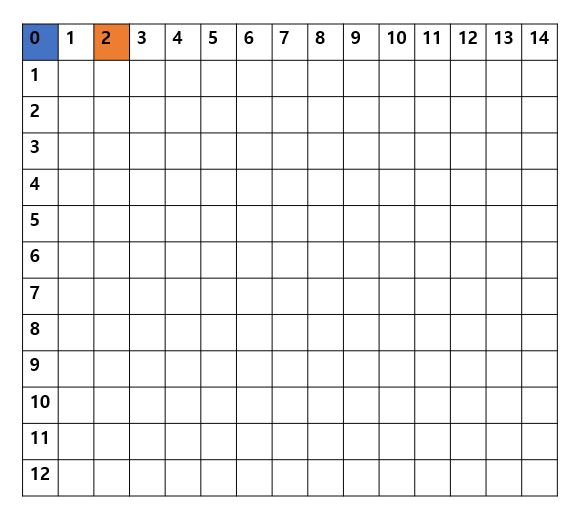
그걸 설명하자면 먼저 8\*2+6 이 어떻게 나온 숫자인지부터 알아야 한다.

8\*2를 그림으로 표현하자면 이렇다.

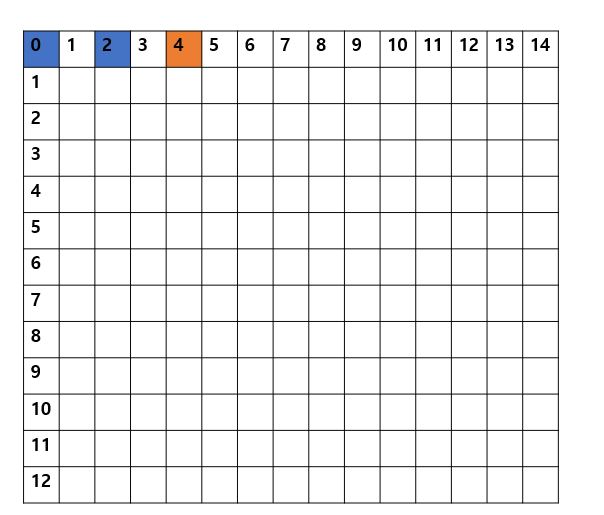


색칠 된 0은 처음 쓰레드가 가장 먼저 접근하는 영역이다.

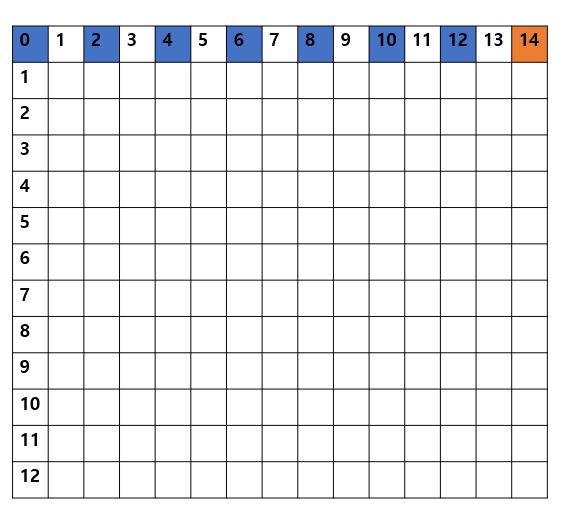
여기서 두 번째 쓰레드가 stride인 2 만큼 이동하면 이렇게 된다.



그리고 다시 세 번째 쓰레드가 2만큼 이동하면 이렇게 된다.



최종적으로 by가 0일 때 bx가 8인, 8번째 쓰레드는 이 위치에 있게 될 것이다.



그림에서 보다시피 8번째 쓰레드의 시작 지점은 (14,0)이 된다.

0번째에서 시작했으니 실제로는 15번째인 셈이다.

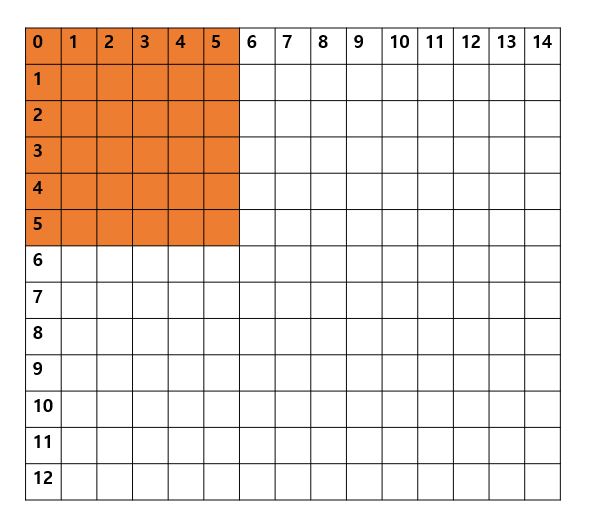
8\*2=16이지만, 애초에 0번에서 시작하는 만큼 1칸을 덜 이동하는 것과 같다.

그래서 8\*2-1이 나오는 것이다.

그렇다면 6-1은 어떻게 나온 걸까?

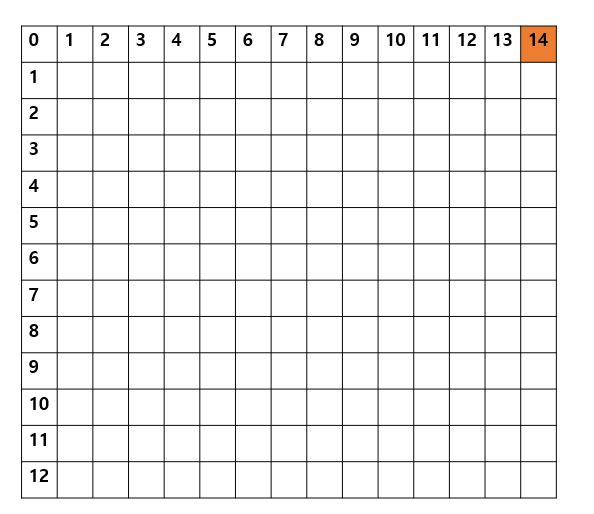
6은 아까도 설명 했다시피 Filter의 한 축 길이이다.

한 쓰레드는 Filter 크기인 6\*6만큼, 총 36번을 움직이게 된다.



그림에서 색칠된 부분이 첫 번째 쓰레드가 접근한 위치이다.

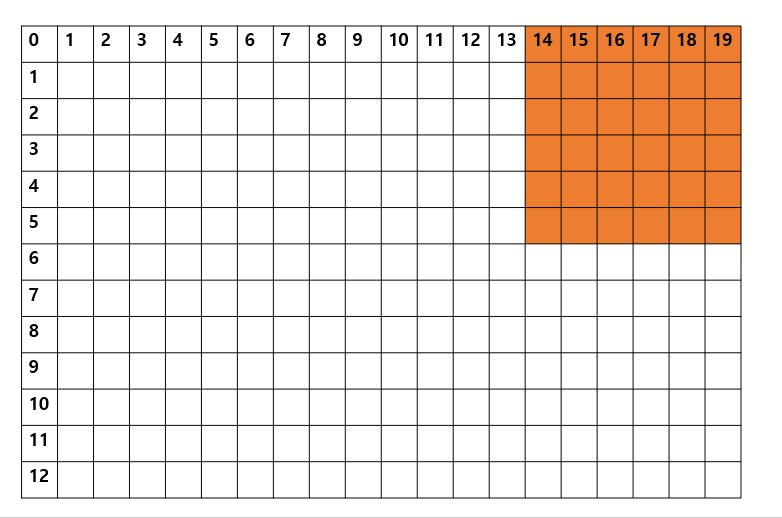
그렇다면 여덟 번째 쓰레드의 경우를 보자



여덟 번째 쓰레드의 시작지점은 이곳이다.

이 쓰레드는 이곳 에서부터 6\*6만큼 접근을 하게 된다.

즉,



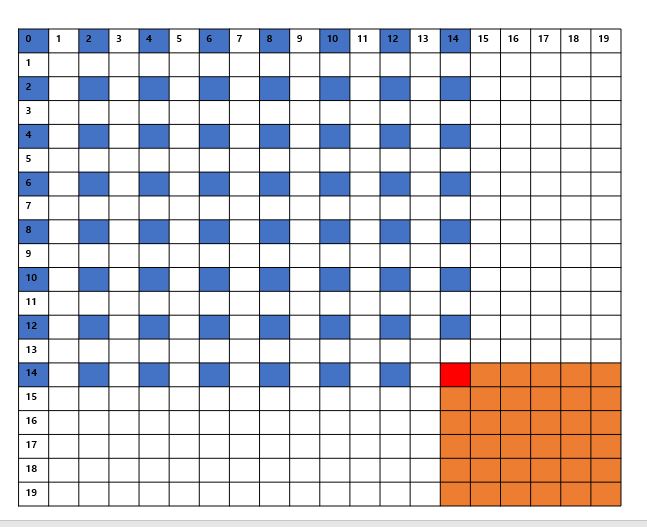
이렇게 접근을 한다는 것이다.

그런데 생각해보면 필터 6\*6인만큼 x축만 총 6번 접근을 하게 되지만, 접근하는 영역에 시작지점을 포함하게 된다.

그래서 그림 에서처럼 최대로 접근한 위치는 19번째가 된다.

6이 아니라 5칸을 추가로 접근하게 된다는 것인데, 그렇기에 6에서 1을 빼 주는 것이다.

이 공식을 적용하면, 8\*8 번째, 블록 내 마지막 쓰레드가 접근하는 영역을 이렇게 된다.



s\_in\_layer의 크기가 왜 20\*20이 되는 것인지 알았다.

그렇다면 다음으로 s\_in\_layer에 값을 채워 넣는 부분을 알 필요가 있다.

s\_in\_layer에 값을 채워 넣는 코드 부분은 이렇다.

d\_in\_layer\_base = by\*8\*2\*1280+bx\*8\*2;

if(tx<5){

uchar4\_tmp = ((uchar4\*)d\_in\_layer)[ (d\_in\_layer\_base + ty\*1280 + tx\*4) / 4];

s\_in\_layer[ty\*20+tx\*4] = uchar4\_tmp.x;

s\_in\_layer[ty\*20+tx\*4+1] = uchar4\_tmp.y;

s\_in\_layer[ty\*20+tx\*4+2] = uchar4\_tmp.z;

s\_in\_layer[ty\*20+tx\*4+3] = uchar4\_tmp.w;

uchar4\_tmp = ((uchar4\*)d\_in\_layer)[ (d\_in\_layer\_base + (8+ty)\*1280 + tx\*4) / 4];

s\_in\_layer[(8+ty)\*20+tx\*4] = uchar4\_tmp.x;

s\_in\_layer[(8+ty)\*20+tx\*4+1] = uchar4\_tmp.y;

s\_in\_layer[(8+ty)\*20+tx\*4+2] = uchar4\_tmp.z;

s\_in\_layer[(8+ty)\*20+tx\*4+3] = uchar4\_tmp.w;

if(ty<4){

uchar4\_tmp = ((uchar4\*)d\_in\_layer)[ (d\_in\_layer\_base + (16+ty)\*1280 + tx\*4) / 4];

s\_in\_layer[(16+ty)\*20+tx\*4] = uchar4\_tmp.x;

s\_in\_layer[(16+ty)\*20+tx\*4+1] = uchar4\_tmp.y;

s\_in\_layer[(16+ty)\*20+tx\*4+2] = uchar4\_tmp.z;

s\_in\_layer[(16+ty)\*20+tx\*4+3] = uchar4\_tmp.w;

}

}

먼저 d\_in\_layer\_base 선언과, tx<5라는 제한을 준 이유를 설명하기 전에,

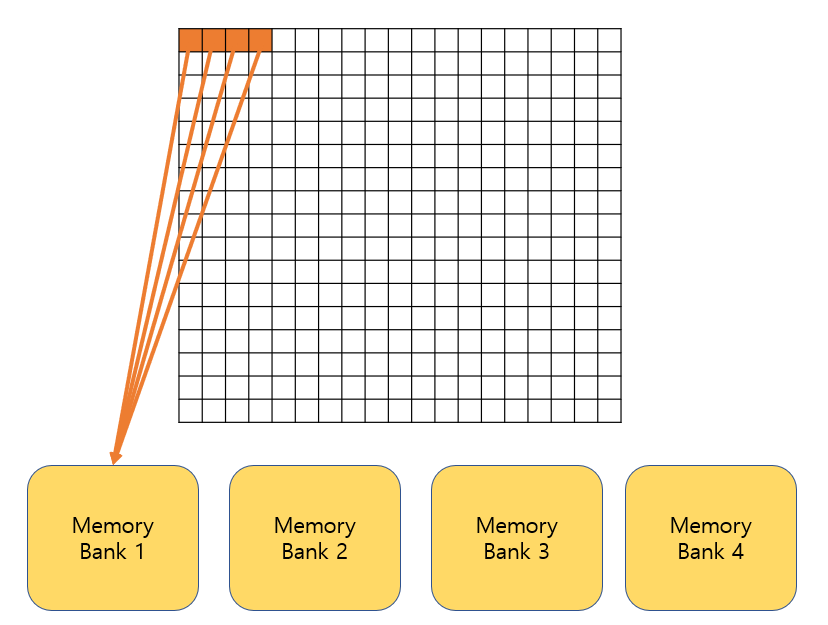
uchar4\_tmp = ((uchar4\*)d\_in\_layer)[ (d\_in\_layer\_base + ty\*1280 + tx\*4) / 4];

부분을 설명할 필요가 있다.

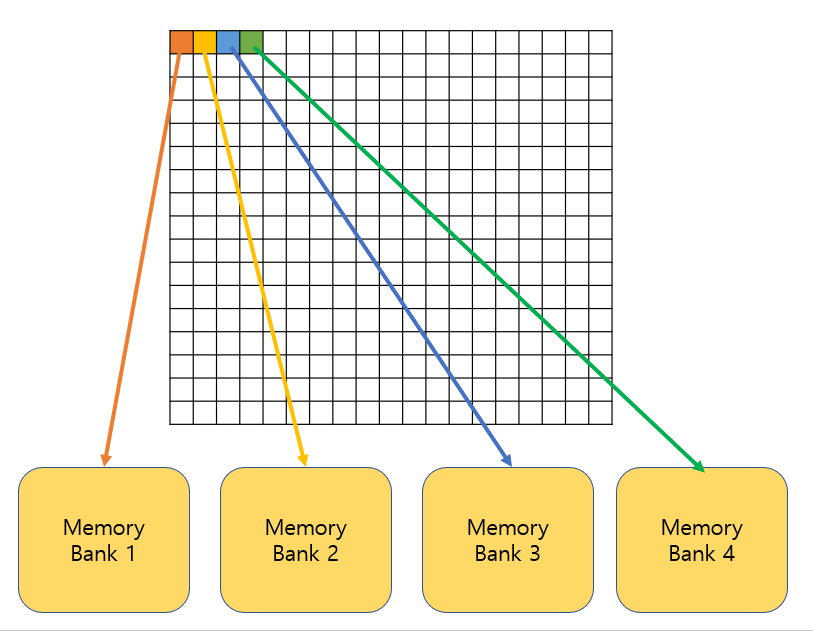
위 연산을 통해 uchar4\_tmp는 char 타입의 4byte짜리 연속된 데이터가 아니라 4개로 분할된 x, y, z, w의 값을 가지게 된다.

이렇게 하는 이유는, 데이터를 연속적으로 입력하게 되면 하나의 메모리 뱅크에만 패킹 하게 되어 뱅크 충돌이 일어나게 되는데, 4개로 쪼개면 각각의 메모리 뱅크에 패킹 하게 되어 뱅크 충돌을 줄일 수 있어 더욱 속도가 빠르고 효율적이기 때문이다.

아래 그림과 같이 unsigned int 자료형으로 선언된 s\_in\_layer 배열에 그냥 값을 저장하면 4byte가 하나의 메모리 뱅크에 패킹 하게 된다.



하지만 uchar4로 4개로 분할한다면 아래 그림과 같이 각 1byte가 각각 4개의 메모리 뱅크에 패킹 하게 되어, 뱅크 충돌을 줄일 수 있다.



d\_in\_layer의 내부에 쓰이는 d\_in\_layer\_base는

d\_in\_layer\_base = by\*8\*2\*1280+bx\*8\*2;

이런 식으로 선언과 동시에 초기화가 되어, input\_layer를 블록과 쓰레드로 나눴을 때 각 블록의 시작점을 가진다.

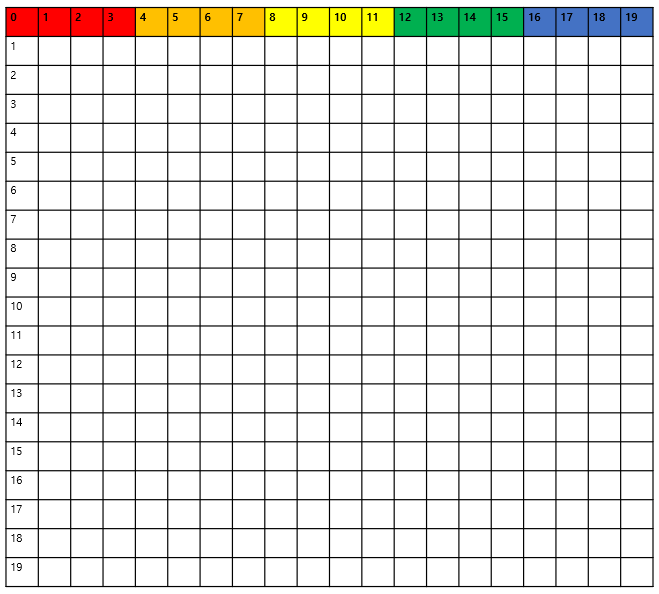
그 위치에서 쓰레드(tx, ty)가 움직이며 4byte씩 공유메모리에 넣는다.

여기서 if(tx<5) 라는 조건문을 추가해 tx가 최대 4가 되도록 한 이유는, x축 가장 끝 부분에 들어갈 s\_in\_layer[ty\*20+tx\*4+3] = uchar4\_tmp.w;

값이, tx가 4여야

s\_in\_layer[0][19] = uchar4\_tmp.w;

처럼 마지막 값이 공유 메모리 x축 최대 길이에 딱 맞기 때문이다.



if(tx<5){

영역에는

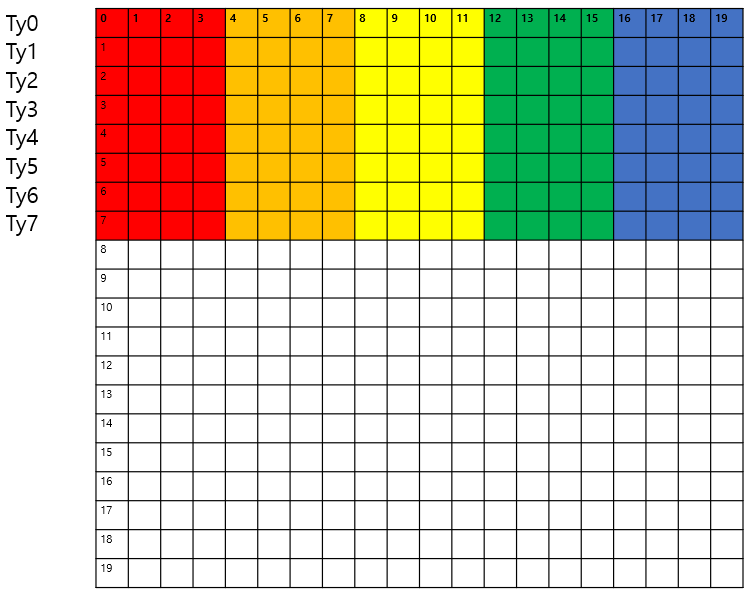
s\_in\_layer[ty\*20+tx\*4] = uchar4\_tmp.x;

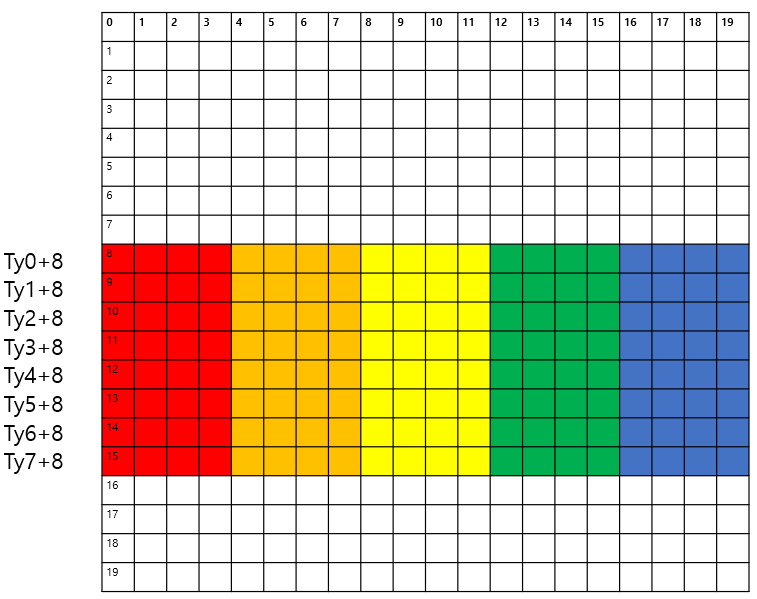
와 함께 ty에 +8을 하는 아래와 같은 코드가 있는데,

s\_in\_layer[(8+ty)\*20+tx\*4] = uchar4\_tmp.x;

이렇게 하는 이유는 쓰레드의 개수가 8개이기 때문이다.

아래 그림과 같이 ty의 최대 값은 7이 될 수밖에 없기 때문에 8번째부터는 따로 +8을 하는 코드에서 값을 저장하는 것이다.

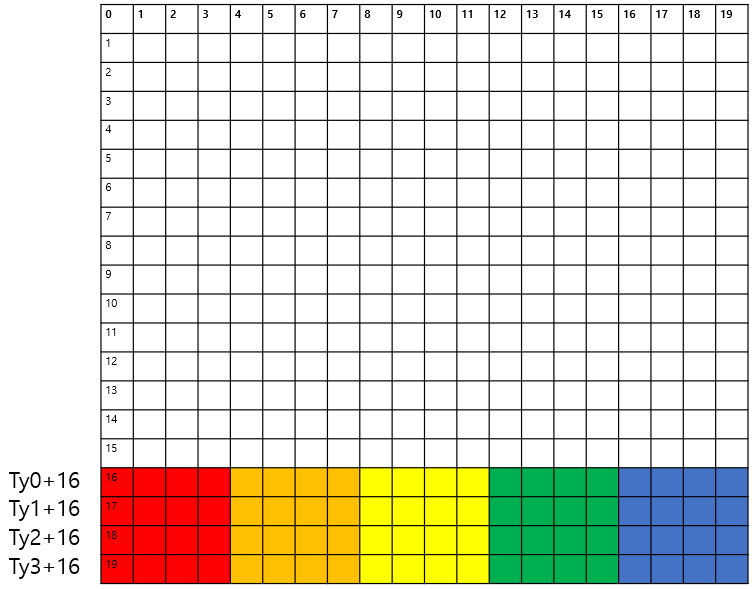




그리고 if(tx<5) 안에 있는 if(ty<4) 조건문은

s\_in\_layer[(16+ty)\*20+tx\*4] = uchar4\_tmp.x;

처럼 ty+16을 해주는데, tx에 최대 4라는 제한을 준 것과 같이 +16을 해줄 때 ty가 최대3이여야 s\_in\_layer[19][19] 처럼, y축 최대 길이에 딱 맞기 때문이다.

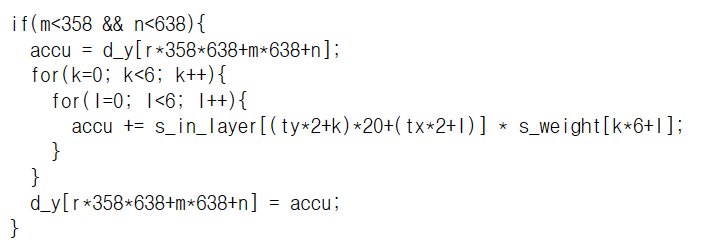


그리고 s\_in\_layer에 값을 입력하는 동안 \_\_syncthreads(); 동기화 함수를 사용해 쓰레드의 진행을 멈춘다.

S\_in\_layer에 값을 넣는 동작이 끝나면, s\_in\_layer에 필터값을 곱연산하는 동작을 수행한다.

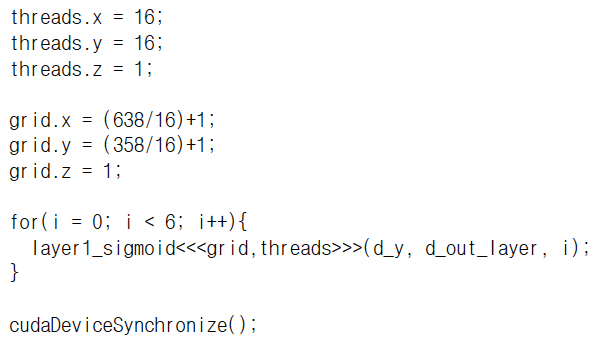
Grid에 +1을 한 만큼 m과 n의 최대 값이 358, 638을 넘어서기에 범위를 벗어나지 않도록 조건을 주고, 곱연산한 값을 저장했다가 한 번에 output 배열에 넣어줄 accu에 현재 위치를 주어 32번 동안 in\_layer에 Filter값을 곱해서 더하는 곱연산 동작을 수행한다.

그리고 ouput 배열인 d\_y 배열에 곱연산이 완료된 accu값을 넣어주는 것으로 layer1\_feature\_maps 커널의 동작이 끝이 난다.



다음으로 활성화를 위해 sigmoid 커널을 호출한다.

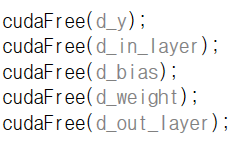
Threads의 개수는 각각 16으로, 그만큼 grid에 16을 나눠주고 소수점이 무시되기에 1을 더해준다.



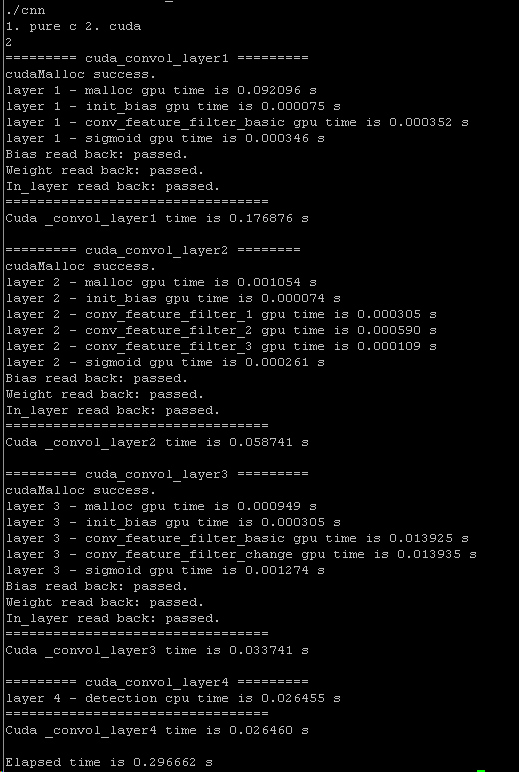
하나의 층에서 특징맵을 추출하는 동작은 이걸로 끝이 났다.

이제 지금까지의 과정을 통해 만들어진 d\_out\_layer를 매개변수로 넘겨받았던 host배열인 out\_layer에 복제해서 돌려준다.

이제 cudaMalloc으로 동적할당 했던 배열들은 필요가 없기에 할당을 해제해준다.



Cuda 코드로 구현 후 수행 시간 비교이다.



Cuda 에서 수행되는 과정을 C와 마찬가지로 각각의 과정에 맞게 시간을 측정한 모습이다.

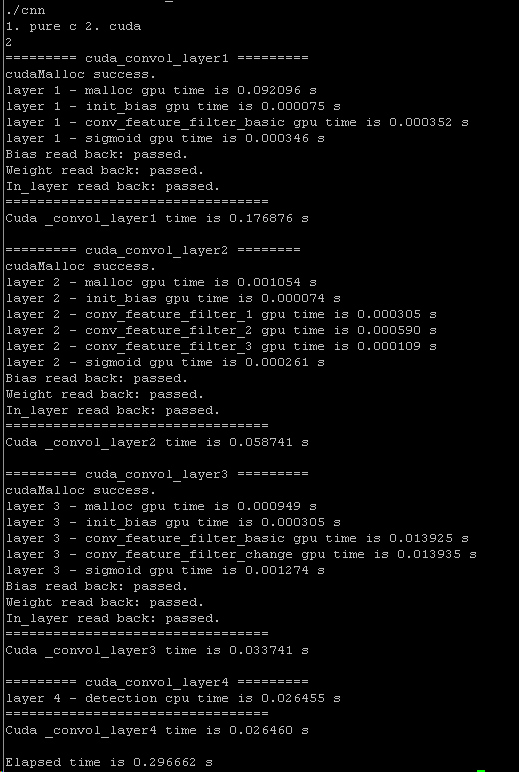
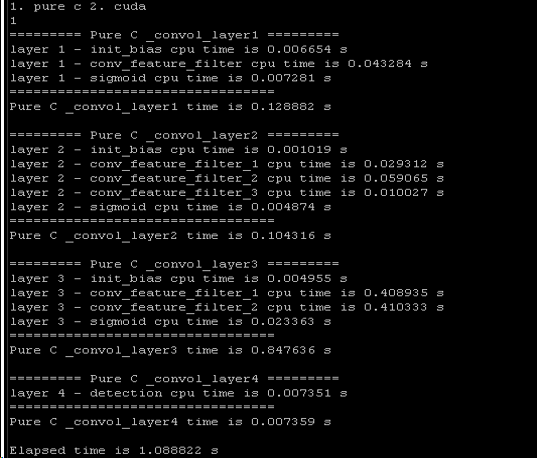
GPU 연산을 활용 하기 위해서는 GPU의 메모리로 연산하고자하는 값을 보내야 하는데

이 과정은 C에는 없는 과정이며 메모리에서 GPU로 데이터를 이동하는 과정인 malloc Gpu 는 많은 시간이 소요가 된다.

이러한 오버헤드가 발생함에도 불구하고 실제 수행 시간이 단위 정도의 차이가 발생 하므로 감수할 수 있는 오버 헤드이다.

그러나 연산이 오버헤드보다 작을 만큼의 작은 데이터면 적합하지 않은 과정이다.

수행시간 표를 비교해 보자.

Pure C (순차적 연산) Cuda (병렬 처리)

단순 연산 과정만 비교했을 때, CUDA 코드가 놀라울 정도의 속도 향상을 보여주고 있다. 더 좋은 GPU의 사용과 최적화 과정을 거친다면 더욱더 놀라울 정도의 속도 향상을 볼 수 있을 것으로 예상된다.