HW6: VGG-16 in CUDA

Giwon, Seo
Dept. of Computer Science. Yonsei University
2015147531 / mgp20029

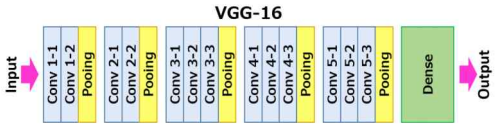
1. Summary

이번 과제의 목적은 CUDA programming을 통해 VGG-16을 구현하여 CPU만을 사용하여 작동시켰을 때와 CUDA를 사용하여 작동시켰을 때의 비교해보는 것이다.

2. Introduction

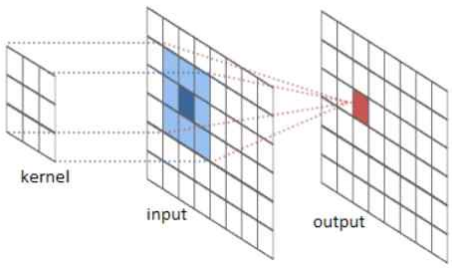
다음은 VGG-16의 정의와 구현에 필요한 연산들이다.

2-1. VGG-16



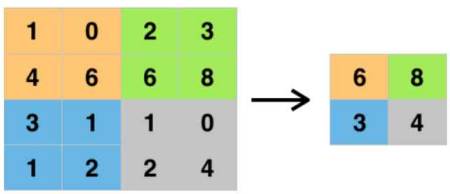
Conv와 pooling으로 이루어진 블록 5개와 fully connected layer 하나로 구성된 네트워크이다.

2-2. Convolution



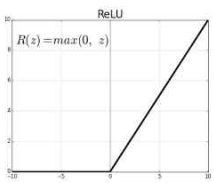
이미지 내에서 원하는 feature를 추출하기 위해 주어진 Kernel(Filter)과 Input feature map을 합성곱한다.

2-3. Max Pooling



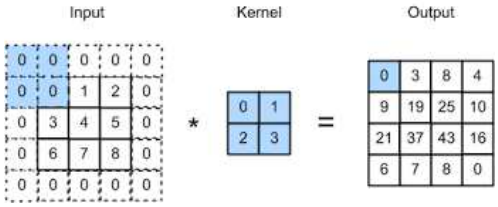
Feature map의 크기와 파라미터 개수를 줄이고 학습 과정에서 주어진 input feature에 overfilling 되는 것을 방지하는 기법이다. 범위 내의 가장 큰 신호만 남기고 나머지는 무시하는 방향으로 진행된다.

2-4. ReLU



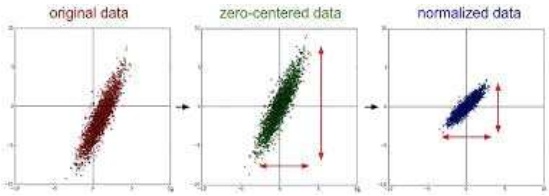
Gradient vanishing 문제를 해결하기 위한 활성화 함수로 0보다 큰 gradient에 대해서만 출력하는 기법이다.

2-5. Zero-Padding



합성곱 연산 과정에서 output의 크기가 현저히 줄어드는 것을 방지하기 위해 주어진 크기만큼 0으로 주변을 채우는 기법이다.

2-6. Normalization



입력 데이터를 표준화시켜 학습 속도를 높이고 local optimal에 빠지게 될 가능성을 줄여주는 기법이다.

3. Code description

3-1. CUDA Grid, Block

```
dim3 dimBlock(col, row);
dim3 dimGrid(batch, channel);
```

2D grid to 2D block으로 구현하였고, 각각은 (batch, channel), (col, row)로 구성되어있다.

```
// index calculation
int B = gridDim.x; int b = blockIdx.x;
int C = gridDim.y; int c = blockIdx.y;
int W = blockDim.x; int w = threadIdx.x;
int H = blockDim.y; int h = threadIdx.y;
int idx = b * (C * H * W) + c * (H * W) + h * (W) + w;
```

따라서, 대부분의 커널 함수의 thread 인덱싱은 위 사진과 같이 구현되었다. Block size의 경우 연산 과정에서의 feature map의 크기가 warp size인 32의 배수 또는 약수로 설정되어 있어 크게 신경 쓰지 않고 병렬 연산의 편리성을 위해 C, W, H는 각각 input, output 중 하나로 설정하여 진행하였다.

3-2. Normalization

```
// Normalize
if (idx < B * C * W * H) {
    input[idx] = image[idx] / max_int;       // transforms.ToTensor();
    input[idx] = (input[idx] - mean) / var;  // transforms.Normalize();
}
```

Image는 0부터 255 사이의 수로 이루어져 있으므로 0과 1 사이의 수로 정규화해주는 과정이다. 정규화 과정에서 사용된 mean과 var의 경우 CPU 코드에 정의된 0.5L를 사용하였다.

3-3. Zero-Padding

```
// length changed after padding
int H_OUT = H + 2 * P;
int W_OUT = W + 2 * P;

int input_base = b * (C * H * W) + c * (H * W) + h * (W) + w;
if (input_base < B * C * W * H) {
    int output_idx = b * (C * H_OUT * W_OUT) + c * (H_OUT * W_OUT) + (h + P) * W_OUT + (w + P);
    input_padded[output_idx] = input[input_base];
}
```

Warp size가 32이기 때문에 블록의 크기를 32의 배수로 하는 것이 시간적 이점을 가질 수 있다. 따라서 주변을 0으로 채워 크기가 이상적인 32의 배수가 되지 않는 output이 아니라 input을 하나의 thread에 배정한다. 이후 output index를 계산하여 zero-padding 결과를 저장한다.

3-4. Convolution

```
// convolution
int output_idx = b * (OC * H_OUT * W_OUT) + oc * (H_OUT * W_OUT) + h * W_OUT + w;
if (output_idx < B * OC * H_OUT * W_OUT) {
    float acc = bias[oc];
    for (int ic = 0; ic < IC; ic++) {
        int input_base = b * (IC * H * W) + ic * (H * W) + h * (W) + w;
        int kernel_base = oc * (IC * K * K) + ic * (K * K);

        for (int kh = 0; kh < K; kh++)
            for (int kw = 0; kw < K; kw++) {
                acc += input[input_base + kh * (W) + kw] * weight[kernel_base + kh * (K) + kw];
            }
    }
    output[output_idx] = acc;
}
```

3-3과 반대로 output feature map의 크기가 이상적인 32의 배수이므로 output element 하나를 하나의 thread에 배정하는 방식으로 진행한다. Global memory access를 줄이기 위해 acc 변수를 설정하여 연산한 이후 결과로 저장한다.

3-4-1. Conv_Block5 최적화

```
__shared__ float bias_shared[512];

// convolution
int block_id = b * (OC) + oc;
if (block_id < B * OC) {
    for (int bi = 0; bi < 512; bi++) bias_shared[bi] = bias[bi];
    __syncthreads();
```

Conv_Block 5의 경우 output feature map의 행과 열이 각각 2밖에 되지 않아 충분한 thread의 개수를 만족하게 하지 못한다. 따라서, Grid 크기를 batch, Block 크기를 channel로 하여 한 matrix를 한 스레드가 연산하게끔 하는 형태로 thread의 개수를 충족시켜주도록 하였다. 그 과정에서 bias에 대한 global memory access 횟수를 줄이기 위해 shared memory에 저장하는 방식을 사용하였다.

3-5. ReLU

```
// relu if (idx < size)
if (idx < B * C * W * H) {
    feature_map[idx] = feature_map[idx] > (float)0.0 ? feature_map[idx] : (float)0.0;
}
```

Input size 내에 존재하는 feature map 원소 중에서 0보다 큰 값에 대해서만 활성화를 진행한다.

3-6. Max Pooling

```
float max_val = -256;
int output_index = b * (C * H_OUT * W_OUT) + c * (H_OUT * W_OUT) + h * W_OUT + w;
if (output_index < B * C * H_OUT * W_OUT) {
    // Find maximum
    for (int sh = 0; sh < scale; sh++)
        for (int sw = 0; sw < scale; sw++) {
            float val = input[b * (C * H * W) + c * (H * W) + (2 * h + sh) * (W) + (2 * w + sw)];
            if (val - max_val > 0) {
                max_val = val;
            }
        }
    // Set output with max value
    output[output_index] = max_val;
}
```

Output element 하나당 thread 하나를 배정하여 주어진 scale(2)을 기준으로 input index를 계산하여 MaxPooling을 진행한다.

3-7. Fully Connected Layer

```
int B = gridDim.x; int b = blockIdx.x;
int OC = blockDim.x; int oc = threadIdx.x;
int idx = b * OC + oc;
```

마지막 Layer의 경우 input 크기가 [batch][512][1][1]로 주여졌고 output이 [batch][10]으로 주어졌기 때문에 batch를 블록 개수, channel을 thread 개수로 처리하였다.

```
if (idx < B * OC) {
    float acc = bias[oc];
    for (int ic = 0; ic < IC; ic++) {
        acc += weight[oc * IC + ic] * input[b * IC + ic];
    }
    output[idx] = acc;
}
```

Global memory access 횟수를 줄이기 위해 acc 변수를 설정하여 곱 연산을 진행한 후 결과값을 저장한다.

4. Result

결과 시간대가 여러 개로 측정되어 평균값으로 설정하였다.

Case	Time(ms)
CPU only	41781
Conv_basic_kernel	425.3
Conv_kernel_b5	400.266

5. Discussion

CPU만을 사용하여 많은 양의 행렬 연산을 진행할 때보다 GPU를 활용하여 단순 계산을 진행하여도 10배 정도의 성능 향상을 보여주었다. 충분한 Thread의 개수 충족과 share memory를 활용하여 Global memory access 횟수를 줄인 결과 더욱 단축된 결과를 보여주었고, Im2Col 형태로 바꾸어 계산을 한다면 더욱 향상된 결과를 보여줄 것으로 예상된다.

6. Reference

[1] Rizgar R.Zebari, Lailan M Haji etc., 「GPUs Impact on Parallel Shared Memory Systems Performance」, International Journal of Psychosocial Rehabilitation, 2020