

4190.301A Hardware System Design

Spring 2020

Hardware System Design Lab7 Report

Kim Bochang

2014-16757

1. <lab7> Introduce

Lab 7의 목적은 학습된 DNN을 이용하여 MINST 데이터 셋을 해석하는 과정에서, convolution이 원활히 일어날 수 있도록 image와 준비된 filter를 행렬 곱 하기 편한 형태로 바꾸는 것을 구현하는 것이다.

2. Implementation

2.1 src/fpga_api.cpp, fpga_api_on_cpu.cpp

우리가 구현해야 하는 부분은 src 폴더의 fpga_api.cpp의 FPGA::convLowering 부분의 코드를 구현하는 것이다.

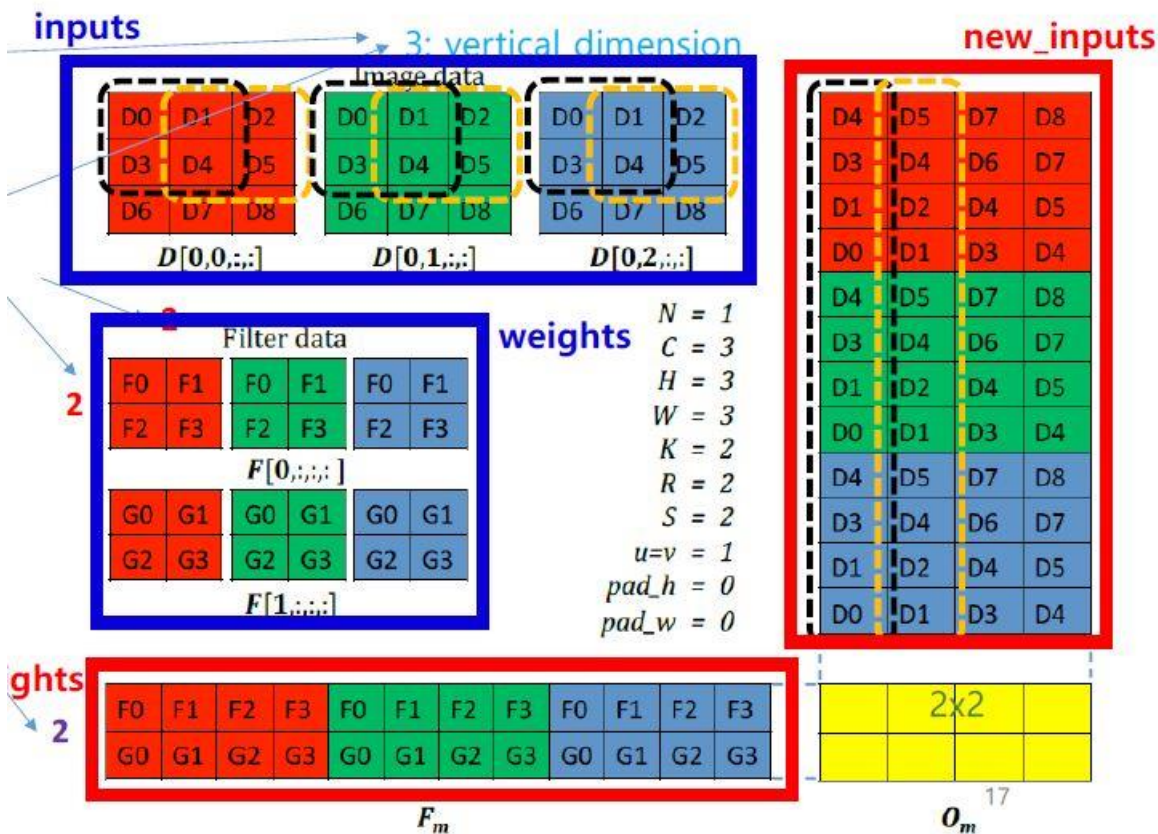
FPGA 클래스에 관한 내용은 Lab2에서 레포트에 서술 했으므로, convLowering 함수에 대해서만 서술한다.

FPGA::convLowering 함수가 하는 역할은 각각 4차원, 3차원 배열형태로 주어지는 cnn_weights, inputs

변수를 바로 matrix multiplication할 수 있는 2차원 배열 형태로 바꾸어

new_weights, new_inputs에 저장하는 것이다.

이를 통해 convolution을 편하게 진행할 수 있게 된다.



즉, 위 그림에서 파란 부분을 빨간 부분으로 바꾸어주는 역할을 하는 함수이다.

이는 다음과 같이 구현할 수 있었다.

```
int i,j;

int filter_dim, filter_row, filter_col, ref_row, ref_col, input_dim, input_row, input_col;

int filter_size = input_channel * conv_height * conv_width;
int block_size = conv_height * conv_width;
int block_number = (input_height - conv_height + 1) * (input_width - conv_width + 1);

for(i = 0; i < conv_channel; i++) //fill new_weights
{
    for(j = 0; j < filter_size; j++)
    {
        filter_dim = j / block_size; //for filter dimension
        filter_row = (j % block_size) / conv_width; //filter height
        filter_col = (j % block_size) % conv_width; //filter width

        new_weights[i][j] = cnn_weights[i][filter_dim][filter_row][filter_col];
    }
}
```

먼저, filter에 해당하는 cnn_weight의 멤버들을 new_weight로 바꾸어주는 부분이다.

각 filter channel마다 filter의 dimension, row, column을 계산하고, 이를 바로 data와 matrix multiplication으로 계산될 수 있는 형태로 바꿔 new_weights에 저장하게 된다.

그 후, input 역시 바로 matrix multiplication 하기 편한 형태로 바꾸어준다.

```
for(i = 0; i < filter_size; i++)
{
    for(j = 0; j < block_number; j++) //fill new_inputs
    {
        ref_row = j / (input_width - conv_width + 1); //block location. inputs[input_channel][ref_row][ref_col] represent left upper point of block.
        ref_col = j % (input_width - conv_width + 1);

        /*
        input_dim = i / block_size;
        input_row = ref_row + (conv_height - 1) - ((i % block_size) / conv_width);
        input_col = ref_col + (conv_width - 1) - ((i % block_size) % conv_width);
        */

        input_dim = i / block_size;
        input_row = ref_row + ((i % block_size) / conv_width);
        input_col = ref_col + ((i % block_size) % conv_width);

        new_inputs[i][j] = inputs[input_dim][input_row][input_col];
    }
}
```

위와같이 new_inputs에 new_weights과 matrix mutliplication하면 convolution이 되도록 만들어준다.

이를 구현하기 위해, convolution이 일어나는 frame(block)마다의 정보를 new_inputs에 복사하는데,

ref_row와 ref_col이라는 변수는 직사각형 모양의 frame의 좌측 위 좌표를 나타내고, 이 좌표를 이용하여 new_inputs에 해당하는 input의 데이터의 위치를 계산, 집어넣게 된다

여기서 한가지 주의할점이 있는데, lab이나 lecture의 pdf에서는 filter와 data를 convolution할때 filter의 값과 data의 값을, 서로 위치를 반전하여 계산하도록 되어있는데,

우리가 사용할 filter는 이미 반전이 적용되어 있어 filter와 data의 값을 그대로 계산하면 된다.

즉, $D = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ 인 2x2 행렬 (1행 : a,b 2행 : c,d) $F = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ 일 때,

pdf에는 D와 F의 convolution을 $4 * a + 3 * b + 2 * c + 1 * d$ 와 같이 진행해야 하지만,

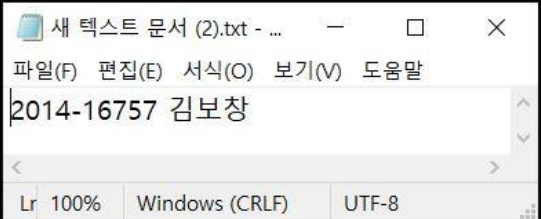
이번 LAB에서는 D와 F의 convolution을 $1 * a + 2 * b + 3 * c + 4 * d$ 와 같이 진행해야 정상적인 결과가 나오게 된다.

3. Result & Discussion

image 개수 10000개로, cnn 네트워크를 사용했을때,


m-size와 v-size를 각각 바꿔가며 partial matrix의 크기를 바꿨을 때, 실행결과와 다음과 같다.

```
root@0775c973907f:~/lab07/hsd20_lab07# python eval.py --num_test_images 10000 --m_size 64 --v_size 64 --network cnn --run_type cpu
[*] Arguments: Namespace(m_size=64, network='cnn', num_test_images=10000, run_type='cpu', v_size=64)
[*] Read MNIST...
[*] The shape of image: (10000, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 0.98,
 'avg_num_call': 741,
 'm_size': 64,
 'total_image': 10000,
 'total_time': 33.26527190208435,
 'v_size': 64}
root@0775c973907f:~/lab07/hsd20_lab07#
```



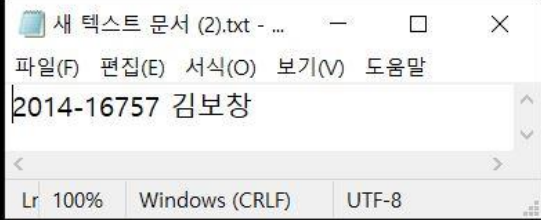
64 x 64 (function call 횟수 741, 실행 시간 33.26초)

```
root@0775c973907f:~/lab07/hsd20_lab07# python eval.py --num_test_images 10000 --m_size 64 --v_size 32 --network cnn --run_type cpu
[*] Arguments: Namespace(m_size=64, network='cnn', num_test_images=10000, run_type='cpu', v_size=32)
[*] Read MNIST...
[*] The shape of image: (10000, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 0.98,
 'avg_num_call': 804,
 'm_size': 64,
 'total_image': 10000,
 'total_time': 15.737563133239746,
 'v_size': 32}
root@0775c973907f:~/lab07/hsd20_lab07#
```




64 x 32 (function call 횟수 804, 실행 시간 15.74초)

```
root@0775c973907f:~/lab07/hsd20_lab07# python eval.py --num_test_images 10000 --m_size 32 --v_size 64 --network cnn --run_type cpu
[*] Arguments: Namespace(m_size=32, network='cnn', num_test_images=10000, run_type='cpu', v_size=64)
[*] Read MNIST...
[*] The shape of image: (10000, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 0.98,
 'avg_num_call': 741,
 'm_size': 32,
 'total_image': 10000,
 'total_time': 16.730325937271118,
 'v_size': 64}
root@0775c973907f:~/lab07/hsd20_lab07#
```



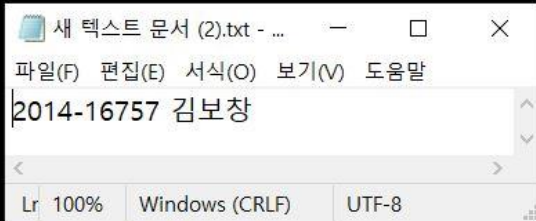
32 x 64 (function call 횟수 741, 실행 시간 16.73초)


```
root@0775c973907f:~/lab07/hsd20_lab07# python eval.py --num_test_images 10000 --m_size 32 --v_size 32 --network cnn --run_type cpu
[*] Arguments: Namespace(m_size=32, network='cnn', num_test_images=10000, run_type='cpu', v_size=32)
[*] Read MNIST...
[*] The shape of image: (10000, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 0.98,
 'avg_num_call': 804,
 'm_size': 32,
 'total_image': 10000,
 'total_time': 9.092642068862915,
 'v_size': 32}
root@0775c973907f:~/lab07/hsd20_lab07#
```




32 x 32 (function call 횟수 804, 실행 시간 9.09초)

```
root@0775c973907f:~/lab07/hsd20_lab07# python eval.py --num_test_images 10000 --m_size 16 --v_size 16 --network cnn --run_type cpu
[*] Arguments: Namespace(m_size=16, network='cnn', num_test_images=10000, run_type='cpu', v_size=16)
[*] Read MNIST...
[*] The shape of image: (10000, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 0.98,
 'avg_num_call': 1186,
 'm_size': 16,
 'total_image': 10000,
 'total_time': 4.947082996368408,
 'v_size': 16}
root@0775c973907f:~/lab07/hsd20_lab07#
```




16 x 16 (function call 횟수 1186, 실행 시간 4.95초)

```
root@0775c973907f:~/lab07/hsd20_lab07# python eval.py --num_test_images 10000 --m_size 16 --v_size 8 --network cnn --run_type cpu
[*] Arguments: Namespace(m_size=16, network='cnn', num_test_images=10000, run_type='cpu', v_size=8)
[*] Read MNIST...
[*] The shape of image: (10000, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 0.98,
 'avg_num_call': 2370,
 'm_size': 16,
 'total_image': 10000,
 'total_time': 6.138796091079712,
 'v_size': 8}
root@0775c973907f:~/lab07/hsd20_lab07#
```



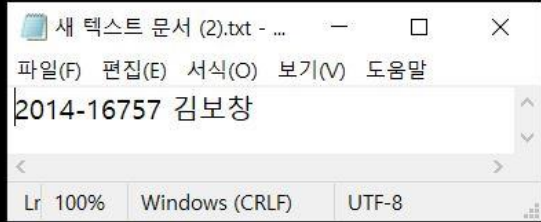
16 x 8 (function call 횟수 2370, 실행시간 6.14초)

```
root@0775c973907f:~/lab07/hsd20_lab07# python eval.py --num_test_images 10000 --m_size 8 --v_size 16 --network cnn --run_type cpu
[*] Arguments: Namespace(m_size=8, network='cnn', num_test_images=10000, run_type='cpu', v_size=16)
[*] Read MNIST...
[*] The shape of image: (10000, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 0.98,
 'avg_num_call': 1696,
 'm_size': 8,
 'total_image': 10000,
 'total_time': 4.865091800689697,
 'v_size': 16}
root@0775c973907f:~/lab07/hsd20_lab07#
```



8 x 16 (function call 횟수 1696, 실행시간 4.86초)

```
root@0775c973907f:~/lab07/hsd20_lab07# python eval.py --num_test_images 10000 --m_size 8 --v_size 8 --network cnn --run_type cpu
[*] Arguments: Namespace(m_size=8, network='cnn', num_test_images=10000, run_type='cpu', v_size=8)
[*] Read MNIST...
[*] The shape of image: (10000, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 0.98,
 'avg_num_call': 3388,
 'm_size': 8,
 'total_image': 10000,
 'total_time': 5.223487138748169,
 'v_size': 8}
root@0775c973907f:~/lab07/hsd20_lab07#
```



8 x 8 (function call 횟수 3388, 실행시간 5.22초)

대체적으로 partial matrix의 크기가 작을수록 실행 시간이 적게 걸림을 알 수 있다.

이는 MLP를 사용했던 lab2에서의 결과와는 크게 다른 결과이다. (lab2에서는 partial matrix의 크기가 클수록 실행시간이 적게 걸렸었음)

이러한 이유를 생각해보았다.

```

void run(const float *src, float *dst)
{
    vector<vector<float>> new_weights(conv_channel_, vector<float>(conv_height_ * conv_width_ * input_channel_));
    vector<vector<vector<float>>> src_(input_channel_, vector<vector<float>>(input_height_, vector<float>(input_width_)));
    vector<vector<float>> new_src_(new_weights[0].size(), vector<float>((input_height_ - conv_height_ + 1) * (input_width_ - conv_width_ + 1)));

    for (int i = 0; i < input_channel_; i++)
        for (int j = 0; j < input_height_; j++)
            for (int k = 0; k < input_width_; k++)
                src_[i][j][k] = *(src + i * input_height_ * input_width_ + j * input_width_ + k);

    dev_>convLowering(raw_weights_, new_weights_, src_, new_src_);

    float *weights_ = vectorToArray(new_weights_);
    for (int i = 0; i < new_src_[0].size(); i++)
    {
        vector<float> vec_src(new_src_.size());
        for (int j = 0; j < new_src_.size(); j++)
            vec_src[j] = new_src_[j][i];

        float *new_src = &vec_src[0];
        dev_>largeMV(weights_, new_src, dst + i * conv_channel_, conv_height_ * conv_width_ * input_channel_, conv_channel_);
    }
}
};

```

<\include\ops.h 의 ConvOP class의 run 함수.>

위에서 구현한 ConvLowering 함수는 CNN을 이용하여 이미지가 어떤 종류인지 score를 계산할 때,

\include\ops.h에 정의되어있는 ConvOP class의 run 함수에서 쓰이게 된다.

이렇게 ConvLowering으로 flatten 된 filter와 image는, 위 코드에서 알 수 있듯

image의 각 block (filter와 곱해지는 영역)과 flatten된 filter를 곱해나가며 convolution된 값을 계산하게 되는데, 이 과정은 각 block마다 이루어지게 된다.

즉, 28 x 28 image를 3x3 filter를 이용하여 26 x 26으로 만드는 과정에서, block이 26 x 26개이므로 이 곱셈은 26 x 26번 이루어지게 되고, 각 filter의 크기는 3x3 = 9, block의 크기 역시 3x3 = 9이므로 크기 9인 행으로 이루어진 행렬과, 크기 9인 벡터를 곱하게 된다.


```

for(int i = 0; i < num_output; i += m_size_)
{
    for(int j = 0; j < num_input; j += v_size_)
    {
        // 0) Initialize input vector
        int block_row = min(m_size_, num_output-i);
        int block_col = min(v_size_, num_input-j);

        // 1) Assign a vector
        /* IMPLEMENT */
        memset(vec, 0, sizeof(float) * v_size_);
        memcpy(vec, (input + j), sizeof(float) * block_col);

        // 2) Assign a matrix
        /* IMPLEMENT */
        memset(mat, 0, sizeof(float) * m_size_ * v_size_);
        for(int k = 0; k < block_row; k++)
        {
            memcpy((mat + v_size_ * k), (large_mat + (i + k) * num_input + j), sizeof(float) * block_col);
        }

        // 3) Call a function `block_call()` to execute MV multiplication
        const float* ret = this->blockMV();

        // 4) Accumulate intermediate results
        for(int row = 0; row < block_row; ++row)
        {
            output[i + row] += ret[row];
        }
    }
}

```

<fpga_api_on_cpu.cpp의 largemv함수>

그런데, 우리가 lab2에서 구현한 largeMV를 생각해보면, partial matrix가 input 크기보다 작은 경우를 고려하고 계산한 것이 아니기 때문에,

(사실 FPGA에 matrix를 옮겨야 하므로 원래 행렬의 크기에는 관계없이 크기를 맞춰 주는 것이 당연하긴 하다)

$n \times 9$ matrix와 크기 9인 vector 곱을 단순히 하면 되는 것을,

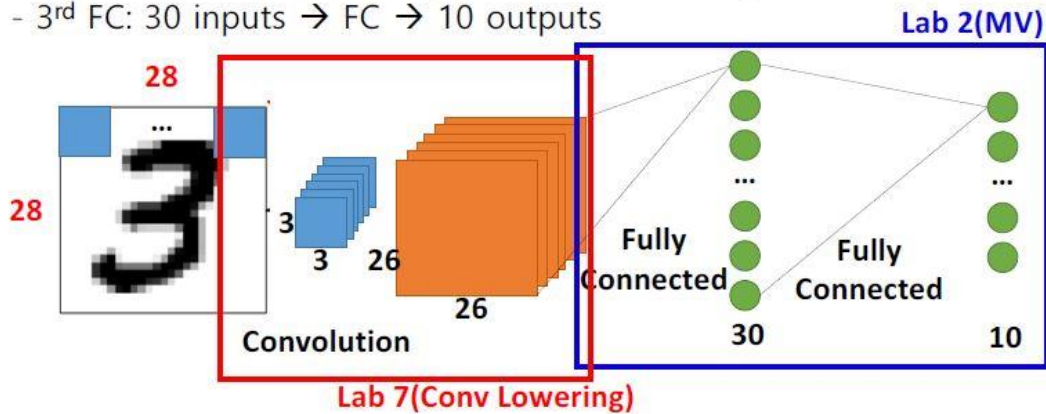
$n \times 9$ matrix를 partial matrix의 크기로 키우고, 벡터 역시 크기를 키워서 계산하게 된다.

즉, 그냥 곱하면 될 것을 partial matrix로 바꾸면서 크기를 키우는 과정에서, 빈 부분에는 0이 들어가게 되고, 이러한 필요없는 0을 곱하는 연산이 많아지기 때문에 속도가 느려지는 것이다.

따라서, 부분 행렬의 크기가 허용 가능한 크기인 16×16 정도로 (9열을 모두 포함하는 행렬) 줄어들 때까지 실행 시간이 적어지는 것은, 이러한 이유로 설명할 수 있게 된다.

(pretrained) Convolutional Network Network(CNN)

- Input: **28x28** pixels \rightarrow **6 3x3** Conv \rightarrow **30** values \rightarrow **10** values
 - 1st Conv: **28x28** inputs \rightarrow 6 3x3 Conv \rightarrow 6 26x26 outputs
 - 2nd FC: $6 \times 26 \times 26 (=4056)$ inputs \rightarrow FC \rightarrow 30 outputs
 - 3rd FC: 30 inputs \rightarrow FC \rightarrow 10 outputs



<cnn 구조>

또한, 우리의 cnn 구조에서, 여기서 필터의 개수는 6개이므로, $n = 6$ 으로, 6×9 행렬과 크기 9인 벡터를 곱하는 연산은 partial matrix의 크기가 8×16 일때 연산으로 한번에 계산되기 때문에, 8×16 을 부분 행렬로 사용하면 추가적인 연산없이 한번에 matrix multiplication을 이용하여 곱을 계산할 수 있게 되므로 이때 가장 실행 시간이 적어지는 것이라고 해석할 수 있다.

물론, Conv lowering 외에도 Fully connected matrix multiplication을 할 때도 partial matrix의 크기가 연산 시간에 영향을 주지만, 두번째 fully connected layer는 10×30 matrix와 size 30짜리 vector 곱을 한번만 하기때문에 partial matrix의 크기에 따라 받는 영향이 그렇게 크지 않은 편이고,

첫번째 fully connected layer의 경우 26×26 짜리 vector와 $30 \times (26 \times 26)$ matrix 곱을 data당 한번 하게된다. 큰 matrix를 곱하므로 partial matrix의 크기에 따라 영향을 받긴 하지만, 횟수가 1번이기 때문에 Conv lowering에서 일어나는 data 1개마다 일어나는 (26×26) 번의 곱셈 중, partial matrix의 크기 때문에 발생하는, 필요 없는 0 연산때문에 발생하는 overhead보다는 상대적으로 영향이 적을 것이라 해석할 수 있다.

4. Conclusion

과제를 수행하면서 network의 구조에 따라, partial matrix의 크기를 잘 조정해야 한다는 것을 알게 되었다. LAB2의 경우처럼 fully connected layer로 이루어진 네트워크에서는 단순히 partial matrix의 크기가 클수록 좋은 퍼포먼스를 보이지만, LAB 7에서처럼 Convolution layer를 이용할 때는, data를 filter를 이용해 처리하는 과정에서, partial matrix의 크기를 너무 크게 잡으면 쓸모 없는 연산이 늘어나게 되어 시간이 더 오래걸릴 수 있음을 알게 되었다.

물론, 실제 하드웨어를 사용하여 partial matrix의 곱으로 ConvLowering을 할 때는 이러한 영향이 CPU에서 연산을 돌릴 때보다 상대적으로 작을 수 있지만, 작은 partial matrix를 사용하여 연산하는 것과, 큰 partial matrix를 사용하여 연산 할 때 속도차이가 거의 없다고 해도, 작은 partial matrix 기준으로 하드웨어를 설계하면 당연히 들어가는 회로의 개수가 적어지므로, 큰 partial matrix를 사용하는 것은 비용적으로 낭비가 될 것이다. 때문에 우리 문제를 정확히 파악하고, 목적에 알맞는 하드웨어를 설계하는 것이 중요함을 알 수 있었다.