

4190.301A Hardware System Design

Spring 2020

Hardware System Design Lab2 Report

Kim Bochang

2014-16757

1. <lab2> Introduce

Lab 2의 목적은 학습된 DNN을 이용하여 MINST 데이터셋을 해석하는 과정에서, 매우 큰 matrix 연산(1024 * 768)을 FPGA에서 처리할 수 있을 정도의 작은 matrix 연산으로 blocking 하여 계산할 수 있게 하는 C++ 코드를 작성하는 것이다.

2. Implementation

2.1 src/fpga_api.cpp

우리가 구현해야 하는 부분은 src 폴더의 fpga_api.cpp의 FPGA::largeMV 부분의 코드중, blocking할 부분에 대한 vector와 matrix를 초기화 하는 부분이다.

FPGA 클래스의 구조는 다음과 같다.

멤버변수

int m_size_ : FPGA에서 사용할 행렬의 행 개수

int v_size : FPGA에서 사용할 행렬의 열 개수 or 행렬계산시 입력으로 들어올 벡터의 크기

int data_size_ : input vector + partial matrix 크기.

int num_block_call_ : blocking한 행렬을 계산하는 함수가 몇번 불렸는지 카운팅 해주는 변수.

float* data_ : 데이터 (partial matrix와 input vector)를 저장하는 위치를 가리키는 포인터.

unsigned int* output_ : output vector가 저장된 위치를 가리키는 포인터

메소드

float* matrix(void) : data_ 포인터에서 matrix가 저장된 위치의 포인터를 리턴하는 메소드.

float* vector(void) : data_ 포인터에서 vector가 저장된 위치의 포인터를 리턴하는 메소드.

const float* blockMV() : 현재 FPGA 클래스에 저장된 input vector와 partial matrix를 행렬곱한 결과를

output vector에 저장하고, output vector의 포인터를 리턴하는 메소드.

void largeMV(const float* mat, const float* input, float* output, int num_input, int num_output)

: 전체 matrix가 저장된 mat 포인터, input vector가 저장된 input 포인터와 output vector를 저장할 output 포인터,

input vector의 크기인 num_input과 output vector의 크기인 num_output을 받고, blocking을 통한 여러번의 partial matrix 연산으로 결과를 output 포인터가 가리키는 위치에 저장해주는 메소드.

우리가 해야하는 것은 largeMV에서 partial matrix 연산을 할때 에 필요한 partial vector와 partial matrix를 초기화해주는 루틴을 구현하는 것이다.

이는 다음과 같이 구현할 수 있었다.

```
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);

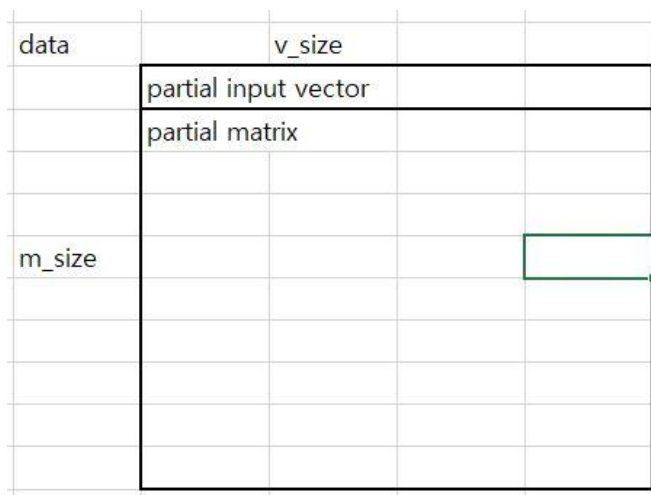
        // 2) Assign a matrix
        /* IMPLEMENT */
        memset(mat, 0, sizeof(float) * m_size_ * v_size_);
        for(int k = 0; k < block_row; k++)
        {
            memcpy((mat + v_size_ * k), (large_mat + (i + k) * num_input + j), sizeof(float) * block_col);
        }

        // 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];
        }
    }
}
```

<src/fpga_api.cpp>

먼저, partial matrix와 partial input vector가 FPGA의 데이터 부분에 저장되어야 하는 위치는 다음과 같다.



따라서, 먼저 저장공간을 memset함수를 이용하여 0으로 초기화 하였다.

memset(dst, val, size)함수는 지정된 포인터 위치 (dst)에서 size만큼의 공간을 val로 초기화 하므로,

이를 이용해 data의 모든 공간을 0으로 초기화 하였다.

초기화 과정을 거치지 않으면 그 전에 남아있던 데이터가 현재 계산에 영향을 주는 경우가 생길 수 있으므로,

(대표적으로, block_row, block_col이 m_size나 v_size보다 작아지는 경우. 후에 설명할것)

초기화 과정이 반드시 필요하다.

그 후, memcpy(dst, src, size) 함수는 목적지 (dst)에 src 포인터의 size만큼에 해당하는 길이를 복사하므로,

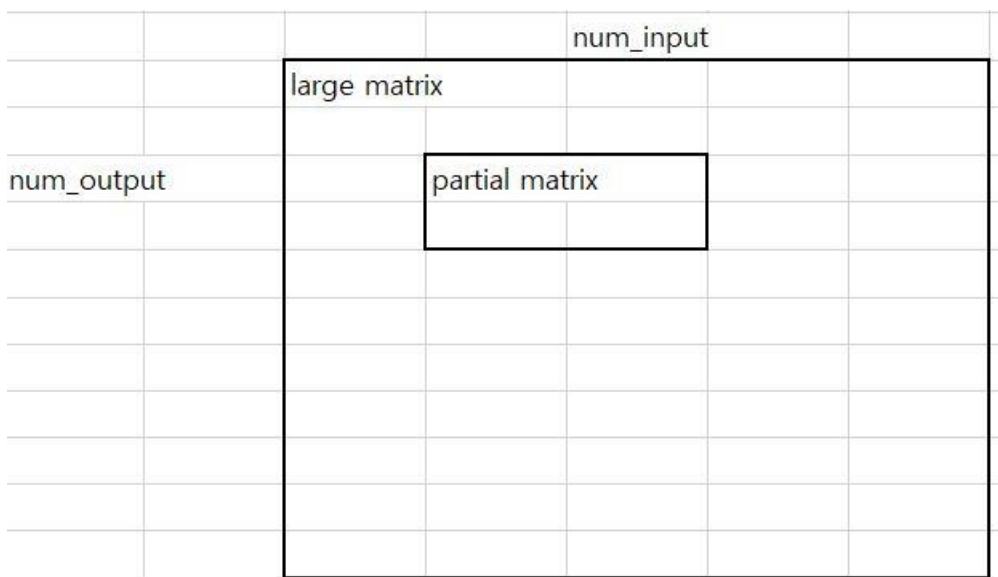
memcpy 함수를 이용해 input vector와 large matrix에서의 값을 FPGA로 복사하도록 하였다.

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

<vector copy code>

matrix의 경우는 vector를 복사할 때와 다르게 for문을 사용해서 여러번 memcpy함수를 호출하게 되는데,

이는 전체 matrix에서 partial matrix로 데이터를 복사해야 할 때 복사해야하는 데이터들이 인접한 데이터가 아니기 때문이다.



partial matrix를 복사하려면, 한줄 복사할때마다 num_input만큼의 주소를 건너뛰어야 하므로 다음과 같은 식을 사용하였다.

```
// 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);
}
```

<matrix copy code>

또한, 전체 곱을 계산하기 위해서 전체 matrix를 partitioning 하는 과정에서, 대부분의 경우는 partition된 matrix의 행과 열의 크기를 나타내는 block_row와 block_col이 FPGA에서의 m_size와 v_size와 같지만, partitioning 도중 large_matrix의 행이나 열의 길이가 block_row나 block_col로 나누어 떨어지지 않는 경우가 존재하여 block_row, block_col이 m_size나 v_size보다 작은 경우가 존재하게 되는데,

이 경우에는 memset을 이용해 초기화를 진행하였기 때문에, partial matrix에 0으로 채워지게 되는 영역이 생기게 되고, 이 경우 행렬 곱을 실행한 결과에는 영향을 미치지 않으므로 위 코드를 사용해도 전혀 문제가 없다.

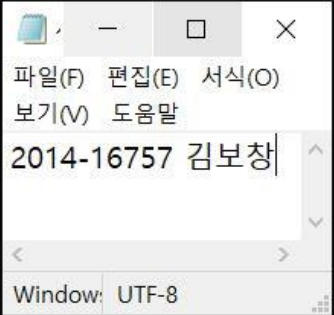
따라서, 먼저 input vector에서 필요한 부분을 fpga의 vector 영역으로 memcpy를 이용하여 불러오고, 그 후 large matrix에서 필요한 부분을 fpga의 matrix 영역으로 불러오게 되었다.

fpga로의 데이터 이동이 끝난 후로는, FPGA::blockMV() 메소드를 이용하여 partial 행렬곱 결과를 FPGA 객체의 output 부분에 저장하고, 이 부분을 전체 output vector의 부분에 계속 더해 나가면서 전체 계산을 진행해 나가게 된다.

3. Result & Discussion

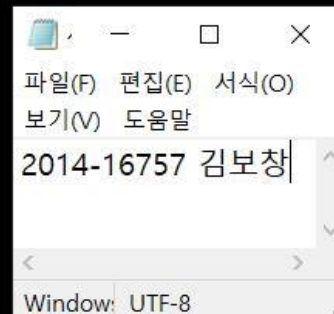
m-size와 v-size를 각각 바꿔가며 partial matrix의 크기를 바꿨을 때, 실행결과는 다음과 같다.

```
root@0775c973907f:~# python eval.py
read dataset...
('images', (10000, 28, 28))
create network...
run test...
{'accuracy': 0.9159,
 'avg_num_call': 627,
 'm_size': 64,
 'total_image': 10000,
 'total_time': 31.92252492904663,
 'v_size': 64}
root@0775c973907f:~#
```



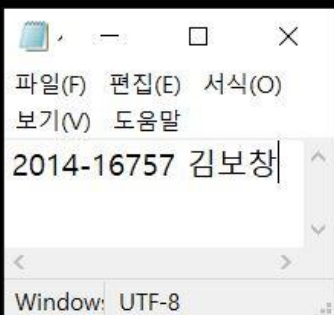
64 x 64 (function call 횟수 627, 실행 시간 31.9초)

```
root@0775c973907f:~# python eval.py
read dataset...
('images', (10000, 28, 28))
create network...
run test...
{'accuracy': 0.9159,
 'avg_num_call': 9375,
 'm_size': 16,
 'total_image': 10000,
 'total_time': 32.07178521156311,
 'v_size': 16}
root@0775c973907f:~#
```



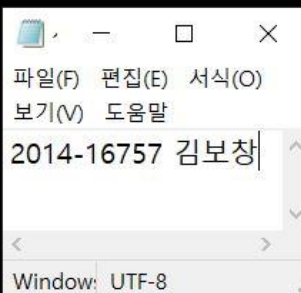
16 x 16 (function call 횟수 9375, 실행 시간 32.1초)

```
root@0775c973907f:~# python eval.py
read dataset...
('images', (10000, 28, 28))
create network...
run test...
{'accuracy': 0.9159,
 'avg_num_call': 18750,
 'm_size': 16,
 'total_image': 10000,
 'total_time': 41.631871938705444,
 'v_size': 8}
root@0775c973907f:~#
```



16 x 8 (function call 횟수 18750, 실행시간 41.6초)

```
root@0775c973907f:~# python eval.py
read dataset...
('images', (10000, 28, 28))
create network...
run test...
{'accuracy': 0.9159,
 'avg_num_call': 18750,
 'm_size': 8,
 'total_image': 10000,
 'total_time': 34.681451082229614,
 'v_size': 16}
root@0775c973907f:~#
```



8 x 16 (function call 횟수 18750, 실행시간 34.7초)

대체적으로 partial matrix의 크기가 클수록 실행 시간이 적게 걸림을 알 수 있는데,

이는 partial matrix의 크기가 클수록 function call을 적게 해도 되어 function call overhead가 줄어들기
때문으로 해석할 수도 있다.

하지만 단순히 function call overhead의 영향이라고 설명하기에는 partial matrix의 크기가

16 x 8과 8 x 16일때, function call의 횟수가 같음에도 불구하고 실행 시간이 극단적으로 차이나므로,
완벽한 설명은 아니다.

이 경우 실행 시간의 차이를 설명하는 원인으로는, 여러가지 원인이 있을 수 있지만 가장 가능성이
높은것은 cache-hit rate때문이라고 추측할 수 있다.

16 x 8의 경우, large matrix에서 partial matrix로 데이터를 복사하는 과정에서 16줄을 복사해야 하는데,
연속되지 않은 데이터를 복사하는 경우가 총 16번 발생한다. 이때 large matrix의 row 길이가 충분히
길어 cache size보다 크게 되므로, large matrix의 다른 줄의 데이터에 접근할 때 cache miss가 발생하여
데이터를 불러오는데 더 많은 시간이 걸리게 된다.

즉, partial matrix의 한 row가 cpu의 cache size에 들어갈 만큼 충분히 작다고 가정하고,

large matrix의 한 row가 cpu의 cache size에 들어가지 못한다고 가정한다면

a x b large matrix, m x n partial matrix의 경우 large matrix의 부분을 fpga의 partial matrix 영역으로 로드
할 때의 cache miss의 대략적인 횟수는

($a \bmod m = 0, b \bmod n = 0$ 으로 근사함. 즉, 서로 나누어 떨어진다고 가정.)

$m * a/m * b/n = a * b/n$ 으로, 즉 partial matrix의 열 길이에 반비례하게 된다.

또한, FPGA에서 partial matrix를 곱연산할 때 역시 cache hit & cache miss가 일어날 것이므로 이러한
경우도 생각해 볼 수 있을것이다.

결론적으로, 16 x 8과 8 x 16의 실행 시간이 크게 차이 나는 이유는 cache hit rate로 설명할 수 있을 것이다.

하지만 cache hit rate만으로는 partial matrix의 크기가 64×64 와 16×16 일때 시간 차이가 크게 나지 않는 이유를 설명할 수 없는데, 이는 64×64 행렬의 데이터를 모두 cache에 담을 수 없기때문에, 계산 도중 빈번하게 cache miss가 일어나게 되고, 따라서 16×16 의 경우와 비교해서 전체적인 계산과정에서의 cache miss와 hit 횟수의 차이가 크게 나지 않아서 그럴 것이라는 추측을 할 수 있겠다.

4. Conclusion

과제를 수행하면서 large matrix의 연산을 partitioning을 통해 수행할 수 있음을 배웠고, 그 과정에서 같은 계산이라도 partitioning 방법에 따라 시간 차이가 크게 날 수도 있다는 것을 알 수 있었다.

더욱 더 효율적인 계산을 하려면 어떻게 partitioning을 진행해야 할지 고민해볼 수 있는 좋은 기회가 되었다.