Alihan Mahanoglu

2019-81292

Janghoon Han

2008-12407

2019.03.14

Hardware System Design Lab #1

The purpose of this lab was to divide a large matrix (which is in the adress "*large_mat" in our case) and a large vector (which is in the adress "*input" in our case) into smaller matrix and vector (where adresses of those small matrix and vector are already defined in the beginning of the function).

```
float* vec = this->vector();
float* mat = this->matrix();
```

Image 1 vec is the pointer for small vector, mat is the pointer for small matrix

```
memset(vec, 0, v_size_ * sizeof(float));(1)
memcpy(vec, input + j, block_col * sizeof(float));(2)
```

Image 2 Usage of memset and memcpy for initializing the small vector

Using memset function – memset(*destination, value, amount of memory) – the information within "vec" was set to 0.000. The amount of memory starting from the adress "vec" is written as v_size_*sizeof(float) as it can be also seen from image 2 above. The reason for using v_size_ instead of block_col in the memory initialization is the fact that "vec" was allocated a fixed size in somewhere else in the code. The memset function does not allocate any memory. Therefore if block_col is used instead of v_size_ in (1), there will be an excess memory part within the "vec", in cases which block_col is less than v_size_.

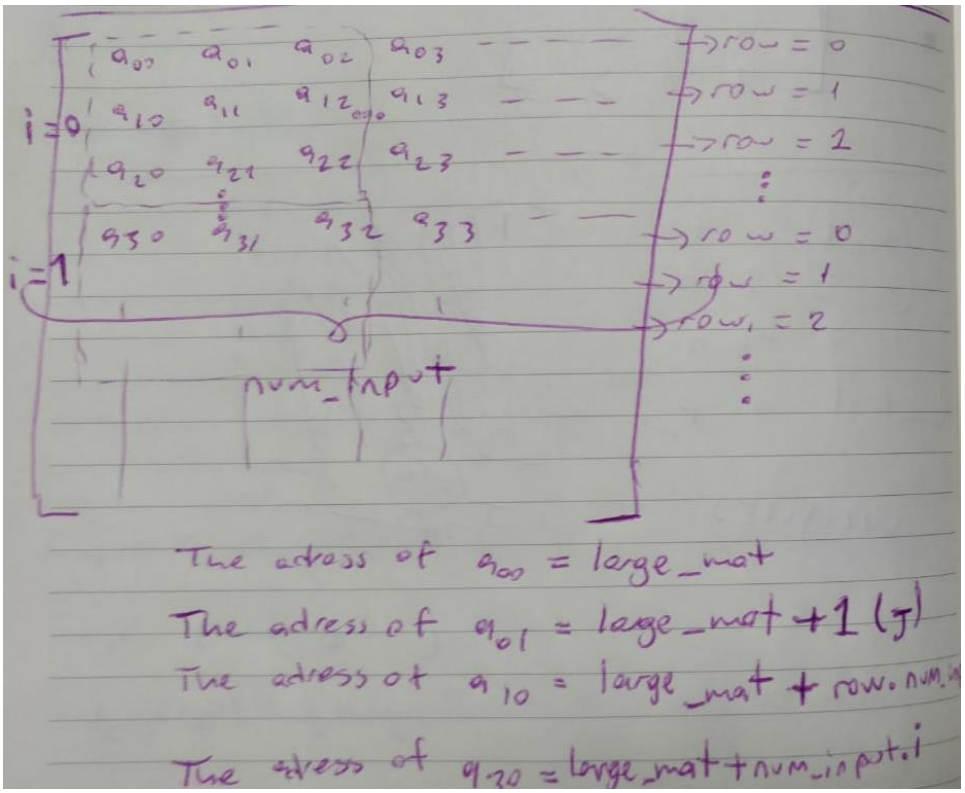
Using memcpy function – memcpy(*destination, source, amount of memory) – a part of the big vector was copied to the smaller vector. As it can be seen in (2) the destination is *vec, the source is input+j (whole vector is sweeped through the index j) and the memory to be copied is block_col * sizeof(float). In this line block_col is used because sometimes the number of columns within the small vector is less than v_size_ (at the end of the iterations for i and j there may be remainders in the end of the loop).

```
memset(mat, 0, m_size_ * v_size_ * sizeof(float));
for(int row = 0; row < block_row ; row++){
        memcpy(mat + v_size_ * row, large_mat + (i+row) * num_input + j, sizeof(float) * block_col);
}
```

Same initialization process is done for small matrix as in the first line of code above. Using memset first m_size_*v_size_*sizeof(float) bytes of memory are initialized with 0. Then memcpy is used for copying the contents of the larger matrix into the smaller matrix. The tricky part here was the changing address of destination and source in every iteration.

Every time within the for loop, as it can be seen from the image above, memcpy function copies a single row of small matrix from the larger matrix. The for loop was necessary because the indexes i and j are multiples of m_size_ and v_size, therefore another index was needed to copy the rows within smaller matrix. The adress for the small matrix is written to be mat + v_size_ * row, for when row is increased by 1, memcpy starts to copy to next row of the small matrix.

The source adress is explained with examples in the image below:



In the example above adress of a_{ij} is can be shown as large_mat + num_input×(i+row)+j which is what was used as the adress in the code.

The Code

```
void FPGA::largeMV(const float* large_mat, const float* input, float* output, int num_input, int num_output)
{
        float* vec = this->vector();
        float* mat = this->matrix();

        // 0) Initialize output vector
        for(int i = 0; i < num_output; ++i)
        {
                output[i] = 0;
        }

        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, v_size_ * sizeof(float));
                        memcpy(vec, input + j, block_col * sizeof(float));

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

The Results

```
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': 37.413089990615845,
 'v_size': 16}
```

```
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': 36.63147521018982,
 'v_size': 64}
```

```
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': 48.08348989486694,
 'v_size': 8}
```

```
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': 40.416162967681885,
 'v_size': 16}
```