하드웨어 시스템 및 설계

LAB-02

이다운

2016-13919

1. Code

본 코드가 실행되기 위해서 1200 by 1200 matrix-vector multiplication을 실시해야 하는데, 사용해야 할 보드가 최대 64 by 64 matrix-vector multiplication을 지원한다. 이 문제를 해결하기 위해 Tiling 방법을 사용한다. Tiling에 대해 간략히 설명하자면 matrix을 잘게 나누어 주어 각각 곱 연산을 해준 뒤 결과값을 모아 matrix-vector multiplication을 실시하는 것이다.

fpga_api.cpp 에서 핵심 함수는 blockMV 와 largeMV 함수이다. blockMV 는 matrix-vector multiplication을 실시하는 함수이고 largeMV 함수는 큰 matrix를 잘게 나누어 각각 blockMV 함수로 연산을 해준 뒤 결과 값을 모아 반환하는 함수이다. 즉 Tiling을 실시해주는 함수이다.

lab02에서 구현해야 하는 부분은 largeMV 함수에서 Tiling을 위한 vector와 matrix의 초기화 및 특정 데이터 할당이다. 이를 구현하기 위해 memset 함수와 memcpy 함수를 사용하였다.

memset 함수는 메모리를 특정 값으로 초기화 하는 함수이고 memcpy 함수는 메모리를 복사하는 함수다.

memset(메모리 공간, 초기화 값, 메모리 공간크기)

memcpy(복사받을 메모리공간, 복사할 메모리공간, 메모리공간 크기)

```
memset(vec, 0, sizeof(float)*block_col);

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

먼저 벡터부터 메모리를 초기화 한 뒤 값을 배정해 주었다.

blockMV의 input 개수인 block_col 개수 만큼 값을 사용하고, 각 값의 데이터 타입은 float이므로 sizeof(float) 와 block_col을 곱한 값만큼을 vec에서 0으로 초기화 시켜준다.

그 뒤에 largeMV의 input에서 내부 for문의 j만큼 이동한 값을 vec 할당해 준다.

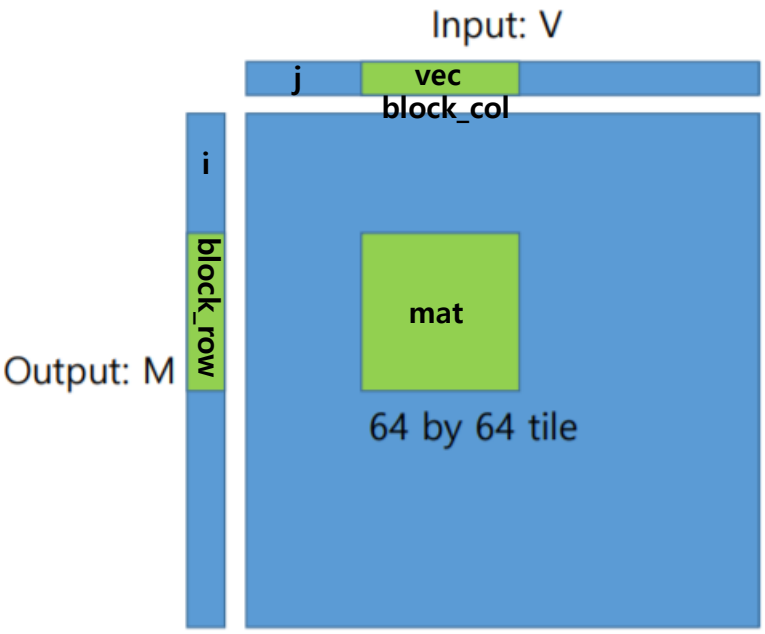
```
memset(mat, 0, sizeof(float)*(block_row *block_col));

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

input 백터에 데이터를 할당해 줬으면 이번엔 가중치에 해당하는 matrix에 값을 초기화 하고 데이터를 할당해 줘야한다.

먼저 전체 가중치 개수인 block_row와 block_col 곱만큼에서 데이터 타입을 곱한만큼 mat를 초기화 시켜준다.

그 뒤에 for문을 돌면서 mat에 block_MV에서 사용해야할 데이터를 할당시켜준다. 위 내용을 그림으로 도식화 하면 아래와 같다.



2. Result

```
root@82610af59901:~# python eval.py
read dataset...
('images', (10000, 28, 28))
create network...
run test...
{'accuracy': 0.918,
 'avg_num_call': 627,
 'm_size': 64,
 'total_image': 10000,
 'total_time': 31.435177087783813,
 'v_size': 64}
```

*제목 없음 - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말

2016-13919 이다운

Ln 1, Col 12 210% Windows (CRLF) UTF-8

```
root@82610af59901:~# 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': 29.69295597076416,
 'v_size': 16}
```

*제목 없음 - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말

2016-13919 이다운

Ln 1, Col 12 210% Windows (CRLF) UTF-8

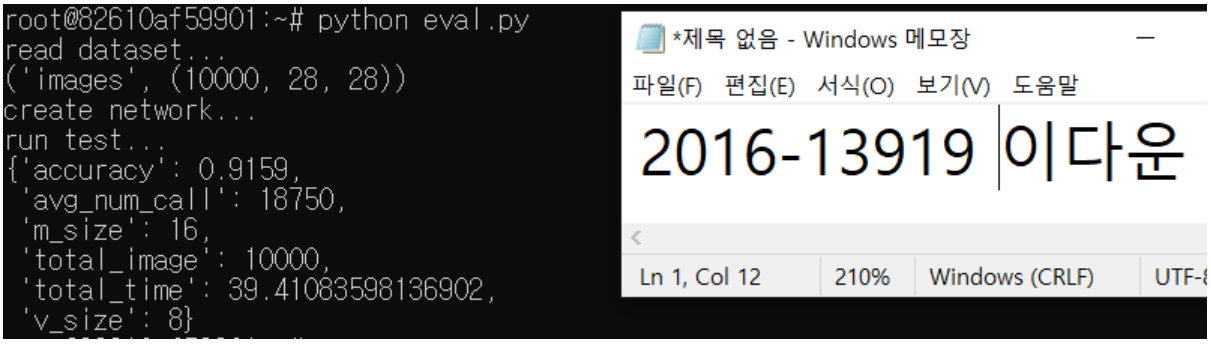
```
root@82610af59901:~# python eval.py
read dataset...
('images', (10000, 28, 28))
create network...
run test...
{'accuracy': 0.9159,
 'avg_num_call': 18750,
 'm_size': 8,
 'total_image': 10000,
 'total_time': 33.12688398361206,
 'v_size': 16}
```

*제목 없음 - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말

2016-13919 이다운

Ln 1, Col 12 210% Windows (CRLF) UTF-8



3. Discussion

block size인 m,n size가 작아질수록 blockMV call time이 줄어드는 것을 확인 할수 있다.

(m,v) size가 (8,16) 와 (16,8) 일때 call time은 같지만 수행 시간에서 차이가 나는 것을 확인할 수 있다. (16, 8)일 때 더 오래걸리는데 이는 cache miss에 의해 수행 시간에 차이가 생긴 것이다.

blockMV 함수 안에는 2중 for문이 있는데 이는 for(m times){ for(v times) } 구조로 되어 있다. 여기서 m+v가 고정일 때 m 이 커지면 내부 for문은 조금만 돌고 외부 for문 그리고 내부for문으로 수행되는데 이렇게 와리가리하면서 속도에 손실이 잃어남과 동시에 cache miss가 발생해 속도 차이가 나는것이다.

위 내용을 자세히 보기위해 극단적으로 (m ,v)을 설정하고 테스트를 해보았다.

왼쪽은 (1, 64)인 경우고 오른쪽은 (64, 1)일 때 인 경우다. (64, 1) 인 경우 수행시간이 훨씬 더 긴 것을 확인 할수 있다. 즉 이 경우 cache miss가 많이 발생해 속도에 손실이 발생한 것이다.