Hardware System Design, Lab 02

2013-11392 김지현

구현체 설명

이번 과제는 행렬-벡터 곱셈에서의 Tiling을 구현하는것이 핵심이다. FPGA는 메모리가 제한되어있기 때문에 입력으로 주어지는 큰 행렬의 곱셈을 한번에 모두 수행할 수 없다. 이때문에 FPGA로 행렬곱을 가속하기 위해선, 큰 행렬-벡터 곱셈을 여러 개의 작은 행렬-벡터 곱셈으로 나눈 뒤 이것의 결과를 더해야하는데, 이번 과제는 그 작업을 위한 연습이라고 생각할 수 있다.

```
// 1) Assign a vector
int k = 0;
for (; k < block_col; ++k) { vec[k] = input[j + k]; }
for (; k < v_size_; ++k) { vec[k] = 0; }

// 2) Assign a matrix
int row = 0;
for (; row < block_row; ++row) {
  int col = 0;
  for (; col < block_col; ++col) {
    mat[v_size_*row + col] = large_mat[num_input*(i + row) + j + col];
  }
  for (; col < v_size_; ++col) {
    mat[v_size_*row + col] = 0;
  }
}
for (; row < m_size_; ++row) {
  for (int col = 0; col < v_size_; ++col) {
    mat[v_size_*row + col] = 0;
  }
}
```

Figure 1. FPGA::largeMV() 함수의 일부

뼈대 코드에 이미 어떻게 입력 벡터와 입력 행렬을 Tiling할지 주어져있었다. 아래의 두 부분만 추가적으로 구현하였고 정상적으로 실행되는 것을 확인할 수 있었다.

- 입력으로 주어지는 큰 행렬 large_mat의 일부를 mat에 복사하는 코드
- 입력으로 주어지는 큰 벡터 input의 일부를 vec에 복사하는 코드
- Tile의 크기가 blockMV 함수가 기대하는 입력보다 더 작을경우, 모자라는 부분을 0으로 초기화하는 코드

large_mat과 input의 크기는 Tile의 크기로 나누어 떨어지지 않을 수 있기때문에, Tile의 크기가 blockMV 함수의 입력보다 작은 경우를 반드시 처리해줘야 한다. blockMV 함수는 행렬과 벡터의 크기를 변경하는 것을 허용하지 않기 때문에, 별도의 처리를 하지 않을 경우 쓰레기 값이 계산 결과에 들어갈 수 있기 때문이다.

실험 결과 분석

과제의 요구사항대로 Block Size를 (64, 64), (16, 16), (8, 16), (16, 8) 로 변경시켜가며 blockMV 함수가 호출된 횟수, 정확도, 전체 수행시간을 비교하였다.

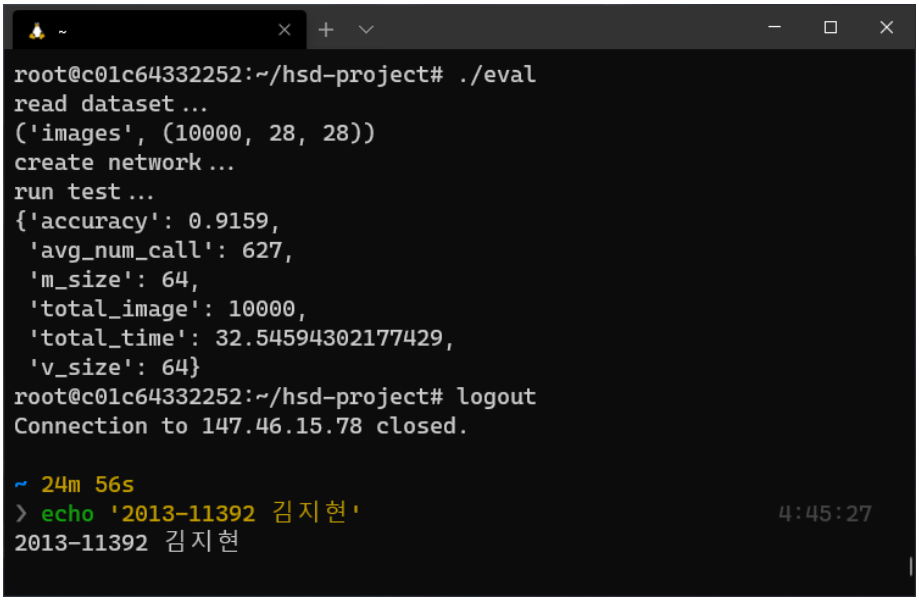

Figure 2. Block Size가 (64, 64) 일 때의 결과

표로 정리하면 아래와 같았다.

BLOCK SIZE	ACCURACY (%)	# OF BLOCKMV() CALL	ELAPSED TIME (초)
64, 64	91.59	627	32.56
16, 16	91.59	9375	38.83
8, 16	91.59	18750	40.28
16, 8	91.59	18750	53.66

과제로 주어진 Pretrained Model의 성능이 준수하여, 91.59%의 정확도를 보여주는 것을 확인할 수 있었다. Block Size는 코드 실행을 어떻게 최적화할지의 문제이지 코드 동작을 바꾸는 것은 아니여서, 어떤 Block Size를 쓰더라도 정확히 같은 Accuracy가 나오는 것 또한 확인할 수 있었다.

Block Size가 작아질수록, blockMV() 함수 호출 횟수가 반비례하여 증가하는 것을 확인할 수 있었다. Block Size가 클수록 입력으로 주어지는 행렬과 벡터에서 한번에 처리하게되는 Tile의 크기가 커지기 때문에, blockMV() 호출 횟수는 Block Size에 반비례한다.

한가지 특기할점으로 Block Size가 작아질수록 수행시간이 길어졌다는것이다. Lab 02는 blockMV() 밖의 계산도, 안의 계산도 모두 CPU에서 수행하기 때문에 단순하게 생각하면 Block Size와 전체 수행시간은 무관하지 않냐고도 생각할 수 있다. 그러나 Block Size를 CPU 캐시 크기가 허용하는 한도 내에서 충분히 크게 설정할 경우, 계산이 좀 더 Cache Friendly해진다. 반대로 Block Size를

불필요하게 작게 설정할 경우, blockMV() 함수 바깥쪽 2중 루프를 도는 과정에서 cache된 데이터가 불필요하게 덮어씌여진다. Block Size가 작아질수록 수행시간이 길어진 것은 이렇게 설명할 수 있고, 이 실험으로 적절한 Tiling이 하드웨어 가속에 유용할 뿐만 아니라 행렬곱을 좀더 Cache Friendly하게 바꾸는데에도 유용함을 알 수 있었다.

두번째 특기할 점으로 Block Size가 (8, 16)인 경우와 (16, 8)인 경우의 수행시간이 크게 차이가 난다는 점인데, 이 역시 CPU Cache로 설명할 수 있다. CPU Cache는 동작원리상 읽고 쓰는 데이터들이 메모리상에서 인접하게 붙어있을수록 성능에 유리하다. 단순하게 생각하면 (8, 16)과 (16, 8)은 크기가 같기 때문에 수행시간이 같을 것 같지만, Tile의 크기가 (8, 16)일 경우 8개의 떨어진 메모리에서 데이터들을 복사해야 하고, (16, 8)일 경우 16개의 떨어진 메모리에서 데이터들을 복사해야 한다. 이 때문에 Tile의 크기가 (8, 16) 일 때가 (16, 8)일 때보다 훨씬 Cache Friendly하여, 수행시간이 더 좋다는 것을 알 수 있다.

결론

큰 행렬곱 코드에 Tiling을 적용하는 방법을 익힐 수 있었고, Tile의 크기에 따라 blockMV 호출 오버헤드가 어떻게 변하는지 알 수 있었다. 그 뿐만 아니라, Tiling을 어떻게 하는지에 따라 프로그램의 Cache Friendliness가 어떻게 변화하는지 배울 수 있는 간단하고 좋은 실험이었다.