

하드웨어 시스템 설계

Lab 2 (SW app on MNIST)

2017-19428 서준원

0. Overview

이번 실습은 하드웨어시스템설계 수업에서 만드는 가속기를 위한 첫걸음이다. 딥러닝을 빠르게 실행하기 위해 하드웨어 가속기를 이용하는 경우가 많다. 딥러닝은 아주 많은 양의 행렬곱(Matrix Multiplication)을 사용하게 되는데, 이는 cpu만이 하기에는 무리가 있어 다양한 하드웨어 가속기를 사용하게 된다. 이 수업에서는 프로젝트로 mv multiplication을 위한 가속기를 만들게 된다. 이번 실습은 이 가속기를 위한 인터페이스를 만드는 작업을 한다.

Multi-Layer Perceptron(MLP)는 인공신경망의 가장 간단한 버전인 퍼셉트론을 여러 층으로 만들어 비선형적인 분류가 가능하도록 한 머신러닝 모델이다. 입력값들에 대해 가중치들을 곱한 값들의 합이 Activation Layer를 거치는 간단한 퍼셉트론이 여러 층을 거쳐서 최종적인 결과를 산출하게 된다. 입력값의 차원보다 더 많은 가중치들의 차원을 통해 비선형적인 분류를 할 수 있다. 위 과정에서 가중치 곱에는 행렬 곱 연산이 사용된다.

이번 실습에서는 이 행렬곱을 효과적으로 하기 위한 가속기 인터페이스를 만들었다. 하드웨어 가속기는 정해진 크기의 행렬곱만을 수행할 수 있다. 따라서 임의의 입력값에 대해서 행렬곱 연산을 수행하기 위해서는 행렬을 블록 단위로 나누어서 행렬곱 연산을 수행해야만 한다. 결과값은 Multiply-Add 형태로 누적되어 최종 결과가 산출되게 된다. 여기서 계산된 결과는 블록 블록 나누어서 하지 않았을 때와 같은 결과이며, 하드웨어 가속기를 사용하기 위해서 Blocking을 하는 것이다.

실습에서 Blocking의 성능을 검증하기 위해서 MNIST 데이터셋을 분류하는 테스트를 했다. 학습 과정은 거치지 않고, 미리 학습된 가중치를 그대로 사용하였다. 행렬곱이 잘 구현되었다면 미리 학습된 가중치를 사용하여 높은 정확도(90% 이상)이 산출 되어야한다. 이와 더불어 행렬곱 Call 횟수를 확인하여 Blocking이 잘 되었는지 분석해보았다.

이번 실습의 구조는 아래 사진과 같다. Eval.py는 fpga_api.cpp에서 정의된 함수를 불러와 MNIST 데이터를 이용해 테스트를 하는 코드이다. 실습에서 직접 수정하는 파일은 fpga_api.cpp이다. 이 파일 중 `largeMV(const float* large_mat, const float* input, float* output, int`

num_input, int num_output) 함수만을 수정하게 된다. 이 함수는 해당 파일의 정해진 크기의 행렬 곱이 정의된 `const float* FPGA::blockMV()` . 이 함수를 사용할 수 있도록 Blocking을 해주는 함수이다.

```
.
|- Makefile
|- src
|   |- fpga_api.cpp
|   |- digit_dnn.cpp
|   `-- py_lib.cpp
|
|- eval.py
|
|- pretrainend_weights
|- include
`-- dataset
```

1. Introduction

(1) Perceptron

퍼셉트론(perceptron)은 인공신경망의 한 종류이다. 퍼셉트론이 동작하는 방식은 다음과 같다. 각 노드의 가중치와 입력치를 곱한 것을 모두 합한 값이 활성화함수에 의해 판단되는데, 그 값이 임계치(보통 0)보다 크면 뉴런이 활성화되고 결과값으로 1을 출력한다. 뉴런이 활성화되지 않으면 결과값으로 -1을 출력한다. 이 퍼셉트론 층이 여러 개가 되면 Multi-Layer Perceptron이라고 부른다.

(2) MNIST

MNIST 데이터베이스 (Modified National Institute of Standards and Technology database)는 손으로 쓴 숫자들로 이루어진 대형 데이터베이스이며, 다양한 화상 처리 시스템을 트레이닝하기 위해 일반적으로 사용된다. 이 데이터베이스는 또한 기계 학습 분야의 트레이닝 및 테스트에 널리 사용된다. MNIST 데이터베이스는 60,000개의 트레이닝 이미지와 10,000개의 테스트 이미지를 포함한다. 이 실습에서는 트레이닝 데이터는 사용하지 않고, 10,000개의 테스트 이미지에 대해서 분류의 정확도를 판단했다.

(3) Hardware Acceleration

하드웨어 가속(Hardware acceleration)은 컴퓨팅에서 일부 기능을 CPU에서 구동하는 소프트웨어 방식보다 더 빠르게 수행할 수 있는 하드웨어의 사용을 말한다. 하드웨어 가속은 이를테면, 그

래픽 처리 장치 (GPU)의 블리팅 가속 기능과 CPU의 복잡한 기능에 대한 함수가 있다.

보통 프로세서는 연속적이며 함수는 하나씩 실행된다. 다양한 기술들은 성능을 개선하는 데에 사용되고 하드웨어 가속은 그 기능들 가운데 하나이다. 하드웨어가 소프트웨어보다 훨씬 빠르게 해 준다. 하드웨어 가속은 계산을 많이 하는 소프트웨어 코드를 위해 고안된 것이다. 하드웨어 가속은 사소한 기능 장치부터 MPEG2와 같은 커다란 기능 블록까지 다양하다.

2. Results

해당 함수를 아래 코드와 같이 만들었다. 가중치 행렬과 입력값의 정해진 부분을 memcpy 함수로 로컬 변수인 mat과 vec으로 옮겨 주었고, 끝의 Padding을 고려하여 해당 부분에 쓰레기 값이 들어가지 않도록 memset으로 0으로 세팅해주었다. 그리고 blockMV 함수를 실행하여 나온 결과를 더해주었다.

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

```
root@8373551e3327:~# 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': 33.001636028289795,
 'v_size': 16}
```

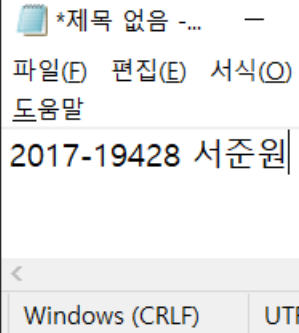


Figure 1 8 x 16

```
root@8373551e3327:~# 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': 39.346094846725464,
 'v_size': 8}
```

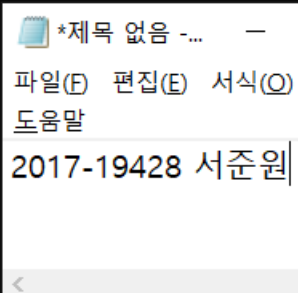


Figure 2 16 x 8

```
root@8373551e3327:~# 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': 30.65532612800598,
 'v_size': 16}
```

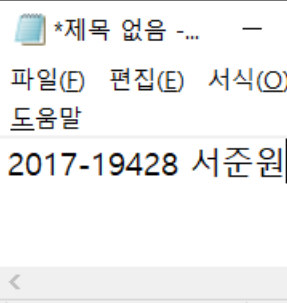


Figure 3 16 x 16

```
root@8373551e3327:~# 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': 31.56501603126526,
 'v_size': 64}
```

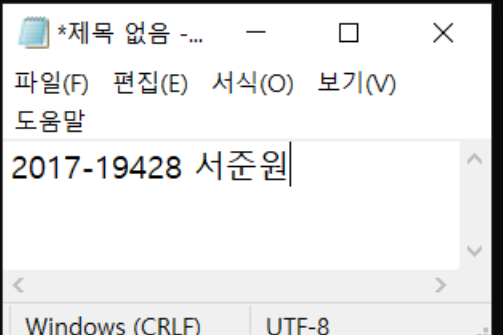


Figure 4 64x64

3. Discussion

- Fpga_api.cpp 구현 내용 설명.

첫째로, Input의 해당되는 부분을 vec으로 옮겼다. 가중치 행렬의 (i, j) 번째부터 block_row, block_col 만큼이 블록이 되므로, Input의 j번째부터 block_col만큼을 vec으로 옮겨주었다. 인풋의 경우에는 패딩을 고려하지 않았다. Matrix의 해당 부분을 0으로 세팅해주면 결과값에 영향을 미치지 않을 것이기 때문이다.

그 후에는 행렬(Matrix)를 mat으로 옮겨주었다. 인풋 벡터와 달리 매트릭스는 2차원이기 때문에 for 문을 통해 한 줄씩 옮겼다. 또한 마지막에 $k < m_size_$ 인 경우에는 패딩에 걸리는 경우로, 남은 행만큼을 모두 0으로 채워줬다. 마지막 블록인 경우가 해당되는데, 이 경우에 기존에 담겨있던 쓰레기 값들이 계산 결과에 영향을 주지 않도록 했다.

- 결과 분석

위의 Figure 1,2,3,4 가 실험 결과이다. 블록의 크기와 상관 없게 모두 같은 가중치이므로 같은 정확도가 나와야 한다. 모두 0.9159의 정확도를 얻었다. 따라서 블록킹이 잘 되었다고 판단할 수 있다.

하지만 블록의 크기 (m_size, v_size)에 따라 함수의 실행 횟수에 큰 차이를 보였다. 블록의 크기가 작아질수록 BlockMV의 실행 횟수가 이에 비례하여 많아진다. Input의 차원이 $28*28 = 764$ 이고 그 다음 레이어는 1200이다.

논리적으로 m_size와 v_size가 1/2이 되면 실행 횟수는 2배가 된다. (64x64)일 때 평균 627회가 실행되었고, (16x16) 일 때 약 9375회가 실행되었다. 이는 약 16배로 패딩을 제외하고는 실행 횟수가 비례했다고 판단할 수 있다. 마찬가지로 (16x8)일 때는 (8x16)일 때와 실행횟수가 18750회로 일치했다. 8이 인풋과 아웃풋의 약수여서 패딩또한 없어 값이 정확히 일치하게 되었으며, 16x16의 정확히 두배를 보였다.