# Hardware System Design
# Term Project v1

2013-11392 김지현

## 개요

Project v0 결과물 위에, Quantization을 구현해 뉴럴넷 평가를 더 가속하는 프로젝트이다. 동작하는 결과물에만 신경썼던 Project v0과는 다르게, 이번 구현체는 성능에 신경을 썼다.

## 구현체 설명

일단 첫번째로, 정상동작하는 CPU Quantization을 구현하였다. 주어진 뼈대코드에 이전 실습에서 구현했던 blocking과 convolution lowering 코드를 넣은 뒤, 아래와 같이 CPU quantization을 구현하였다.

```
static void quantize(float* input, int8_t* quantized, int num_input, float scale) {
  for(int i = 0; i < num_input; i++) {
    int16_t x = static_cast<int16_t>(ceilf(input[i] / scale));
    quantized[i] = x > 127 ? 127 : x < -128 ? -128 : x;
  }
}

static void dequantize(int32_t* quantized, float* output, int num_output, float scale) {
  for(int i = 0; i < num_output; i++) {
    output[i] = scale * static_cast<float>(quantized[i]);
  }
}
```

```
// NOTE: We'll ignore comp->act_bits and comp->weight_bits variable and
// always quantize into 8bit signed integer

float act_scale = (comp->act_max - comp->act_min)/127.0f;
float weight_scale = (comp->weight_max - comp->weight_min)/127.0f;

quantize(vec, qvec_, v_size_, act_scale);
quantize(mat, qmat_, m_size_*v_size_, weight_scale);

for (int i = 0; i < m_size_; ++i)
{
  short sum = 0;
  // NOTE: Overflow may occurs here during accumulation, but we'll ignore it
  for (int j = 0; j < v_size_; ++j)
    sum += qvec_[j] * qmat_[v_size_ * i + j];
  qout_[i] = sum;
}

dequantize(qout_, out, m_size_, act_scale*weight_scale);
```

*Figure 1. CPU Quantization*

기본적으로 Lab 12 PPT에 있는 “solution 2”를 그대로 적용시켰다. PPT에는 min값과 max값을 넘어가는 값들을 clamp해야한다는 설명이 생략되어있는데, 이부분도 적용시켰다.

특기할 점이 두가지 있는데, 첫째는 comp 파라미터로 주어지는 act_bits 변수과 weight_bits 변수를 무시하고 무조건 8bit로 quantize 할것이라는 점이다. 어차피 FPGA 구현을 8bit로 고정시켜

할것이기 때문에, 해당 변수를 사용해 동적으로 quantize하기 어렵다. 때문에 두 변수는 무시하였다. 그리고 quantize한 곱셈 결과를 더할 때, 오버플로우가 발생할 수 있기 때문에, 사용하는 자료형을 주의해야한다.

그 다음엔 quantize된 결과를 FPGA로 곱해주는 구현을 작성했다. 뼈대코드에는 실수를 32bit 정수로 quantize하도록 작성되어있었는데, I/O 시간을 절약하기위해 8bit 정수로 quantize하도록 고쳤다.

FPGA 구현은 Project V0의 구현에서 아래와 같은 수정이 들어갔다.

- LOAD 단계에선 8bit signed int를 한 사이클에 네개씩 읽는다.
- STORE 단계에선 곱셈의 결과인 16bit signed int를 한 사이클에 두개씩 저장한다.
- CALC state가 더 이상 존재하지 않는다. Matrix를 읽음과 동시에 계산을 병렬적으로 진행한다.

결과적으로 Project V0에 비해 훨씬 효율적인 구현체가 되었다. Project V0에선 곱셈을 한번 할때마다 16 cycle을 대기해야했는데, 이젠 정수 곱셈이기 때문에 사이클 소모 없이 행렬을 읽음과 동시에 계산할 수 있게 되었다.

여기까지 구현을 마친 뒤, 직접 만든 테스트케이스를 실행한 결과 정상적인 행렬곱이 이뤄짐을 확인할 수 있었고, 벤치마크를 실행해도 정상적인 결과가 나오는 것을 확인할 수 있었다.

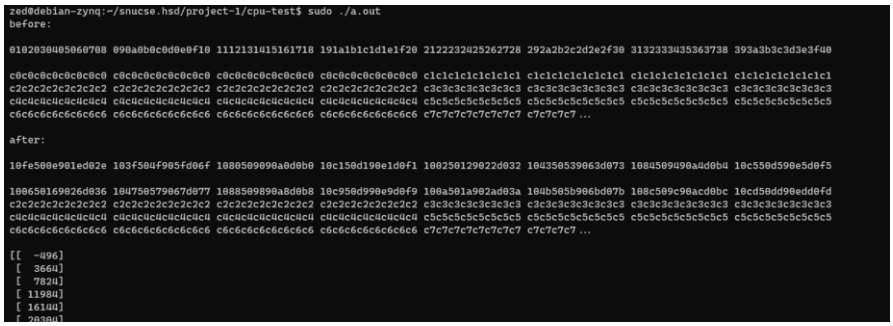


*Figure 2. Test result*

정상 동작 여부는 확인하였고, 성능을 더 좋게 만들기 위한 최적화를 시작했다. 가장 먼저 컴파일러 최적화를 -O3 로 켰다. 이 작업만 수행해도 실행속도가 상당히 빨라진 것을 확인할 수 있었다. 이 때 volatile 키워드를 써서 IP에 접근하는 코드가 최적화되지 않도록 신경써줘야한다.

```
// fpga version
volatile uint32_t *ip_status = output_;

*ip_status = 0x5555;
while (*ip_status == 0x5555);
```

*Figure 3. Using volatile keyword*

그 다음으로는 부동소수점으로 저장되어있는 pretrained weight를 미리 quantize하는 작업을 하였다. 행렬 weight들은 뉴럴넷을 평가하는 내내 값이 변하지 않기 때문에, 미리 quantize시켜서 저장해도 된다. 행렬 계산 특성상 똑 같은 weight를 여러 번 중복하여 quantize시키는 계산이 들어가기 때문에, weight들을 미리 quantize시키는 것 만으로도 상당한 성능 이득을 볼 수 있었다.

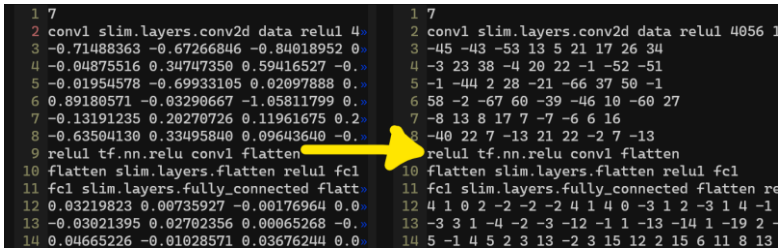


*Figure 4. Pre-quantizing weights*

64*64인 블록 크기를 128*128로 바꾸는 시도도 해보았으나, 오히려 성능이 더 떨어져서 롤백하였다.

# 실험 결과 분석

아래와 같이 성공적으로 벤치마크가 동작함을 확인할 수 있었다.


*Figure 5. Benchmark result*

최적화를 적용했을때의 각 단계별 성능은 다음과 같았다.

| | 최적화 없음 | -O3 옵션 | Weight들 미리 Quantize |
|---|---|---|---|
| CNN without quantization in CPU | 36.93 | 5.14 | N/A |
| CNN with quantization in CPU | 72.86 | 30.43 | 9.17 |
| CNN with quantization in FPGA | 60.43 | 52.76 | **50.67** |

*Figure 6. Benchmark result*

CPU와 FPGA 사이의 통신 오버헤드가 너무 커서, 코드를 최적화했을 때 FPGA를 사용한 코드는 상대적으로 최적화의 득을 보지 못했다.

# 결론

Quantization을 사용해 부동소수점 연산 없이 정수연산만으로 뉴럴넷 평가를 가속시키는 방법을 직접 구현해보았다. 비록 FPGA와 CPU 사이의 통신 병목이 너무 커 CPU로 실행하는 것이 더 빨랐지만, 실제 뉴럴넷 가속기가 어떤 방식으로 동작하는지 익힐 수 있는 프로젝트였다.