Hardware System Design

Term Project V1

이다운 2016-13919

1. Overview

Term project v0 에서 딥러닝 학습용 행렬 곱 프로그램을 verilog로 구현하여 zed보드의 hardware accelerator에 올리는 것을 하였다. 이 구현의 성능을 향상시키는 방법으로 quantization이라는 것이 있다. cpu와 hardware accelerator는 32bit의 floating point data를 서로 교환하는데 이 32bit floating point를 8bit integer data로 변환하여 transfer time과 calculation time 모두 줄이는 방법을 quantization이라고 한다. 이번 V1 에서는 V0에서 구현상황에 quantization을 추가해 성능 향상을 이루는 것이 목적이다.

2. Implementation

```
float sz = (comp->act_min)*(-1.0);
float act_scale = (comp->act_max + sz)/act_bits_max; // TODO calculate the scale factor
int act_offset = (sz/act_scale); // TODO calculate the zero-offset
quantize(input, qinput, num_input, act_bits_min, act_bits_max, act_offset, act_scale);

int weight_bits_min = 0;
int weight_bits_max = (1<<(comp->weight_bits-1))-1;

sz = (comp->weight_min)*(-1.0);
float weight_scale = (comp->weight_max + sz)/weight_bits_max; // TODO calculate the scale factor
int weight_offset = (sz/weight_scale); // TODO calculate the zero-offset
quantize(large_mat, qlarge_mat, num_input*num_output, weight_bits_min, weight_bits_max, weight_offset, weight_scale);
```

먼저 fpga_api.cpp부터 살펴보겠다. 위 사진과 같이 floating point data를 int8로 quantization하기 위해 offset과 scale을 계산하였다. input과 mat의 값을 각각 따로 구해주고 각각 quantize() 함수를 통해 floating point 배열을 int 배열로 변환시켜주었다. 이때 quantize 과정에서 offset은 안 더해 줬다. 왜냐하면 0이 변환후에도 0이 되도록 하여 7bit integer로 변환하기 위함이다.

```
  // 0) Initialize output vector
  for (int i = 0; i < num_output; ++i)
    qoutput[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);
      memset(vec, 0, sizeof(int)*v_size_);
      memset(mat, 0, sizeof(int)*m_size_*v_size_);

      memcpy(vec, qinput+j, sizeof(int)*block_col);
      for(int k=0; k< block_row; k++){
        memcpy(mat+k*v_size_ , qlarge_mat +j+ (k+i)*num_input, sizeof(int)*block_col);
      }

      // 3) Call a function `qblockMV() to execute MV multiplication
      const int* ret = this->qblockMV(comp);

      // 4) Accumulate intermediate results
      for(int row = 0; row < block_row; ++row)
        qoutput[i + row] += ret[row];
    }
  }

  dequantize(qoutput, output, num_output, 0, act_scale*weight_scale);
}
```

quantize후 V0과 마찬가지고 input으로 받는 matrix는 너무 크므로 tiling 하는 과정을 거쳤다. tiling으로 각 타일들을 따로따로 qblockMV함수로 계산한 뒤 qoutput에 누적해주고, 마지막으로 dequantize 함수로 int data를 다시 floating point data로 되돌렸다.

Verilog 구현은 V0과 거의 같다. 다른점으로는 mype 모듈에서 floating point ip를 빼고 대신에 int타입이므로 아래 사진과 같이 바로 MAC연산을 해주었다.

```
    if(buf1==1) begin
      psum<=buff+psumreg;
      buf1<=0;
      dvalid<=1;
    end

    else if(valid ==1)begin
        buff <= ain * bin;
        buf1<= 1;
    end


    if(dvalid ==1)begin
        dvalid <= 0;
        psumreg <=psum;
    end

end
```

v0과 마찬가지로 연산기능을 하는 my pe 모듈을 genvar로 여러 개 생성하여 pe array를 구성하였다. 이렇게 pearray를 구현한 pe_con 모듈을 my_pearray 모듈이 관리하는데 my_pearray 모듈은 bram과의 data transfer을 하기위해 input, output을 그에 맞췄다.

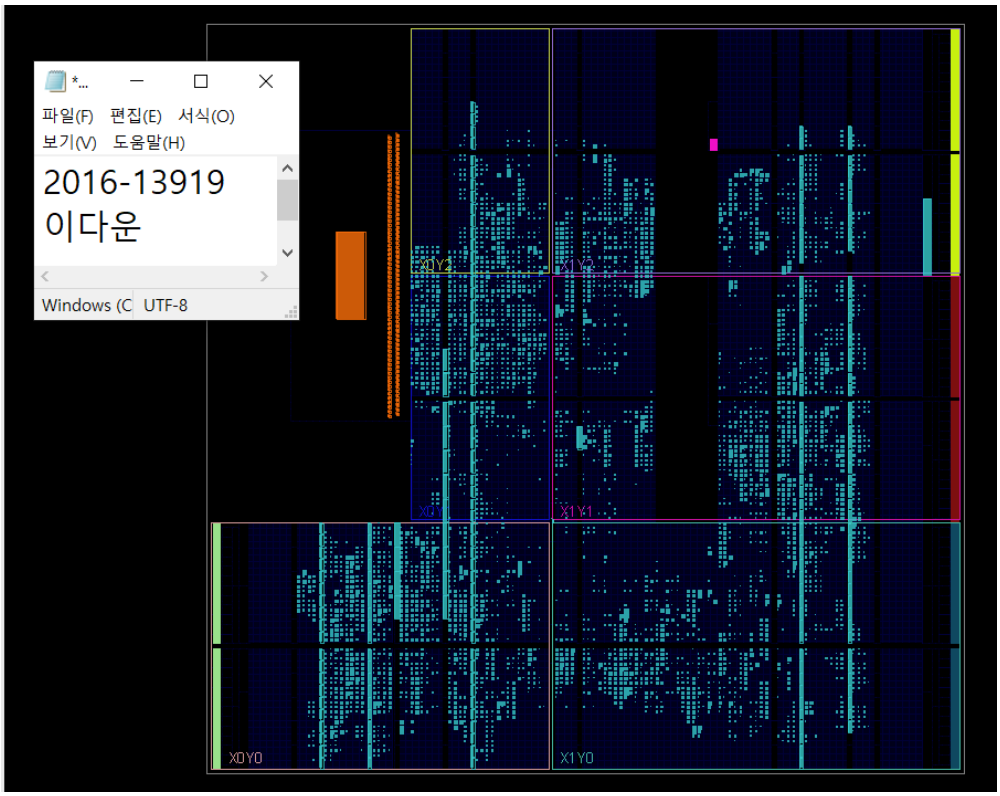
```
generate for(i=0; i<2**MSIZE; i=i+1) begin : pe

my_pe # (.L_RAM_SIZE(VSIZE))
pe(
    .aclk(~clk),
    .aresetn(pe_areset),
    .ain(ain),
    .din(din),
    .addr(local_addr),
    .we(we[i]),
    .valid(valid),
    .dvalid(dvalid[i]),
    .dout(doutii[i])
);
end
endgenerate
```

my_pearray은 start 신호가 들어오면 작동을 시작해 bram과 계속 data transfer을 일으킨다. 구현 방법은 V0과 똑같으므로 구체적인 내용을 생략하겠다.

3. 실행 방법

압축파일에 같이 포함되어 있는 zynq.bit 파일과 fpga code과 담긴 hsd_v1폴더를 같이 zedboard의 sd카드에 넣어준다. 그러고 zedboard와 host pc와 연결해주고 /sdcard/hsd_v1 디렉토리로 이동한다. sudo make로 컴파일을 한번 해주는데 이때 zedboard의 날짜가 이상하면 warning이 뜨므로 sudo date로 현재 시간으로 설정하고 make를 돌려준다. 그리고 sudo bash benchmark.sh 커맨드로 벤치마크 스크립트를 돌려 실행 결과를 본다. 이때 zynq.bit 은 64x64 타일링을 위한 파일이므로 benchmark.sh의 msize와 vsize 모두 64로 그대로 돌려야 하고, zynq_16.bit 파일은 16x16 타일링을 위한 파일이므로 msize와 vsize 모두 16으로 설정하고 돌려야 한다.

4. Result



위 사진은 64x64 타일링을 위한 모듈의 implementation 결과이다. v0과 비교하자면 v0 같은 경우는 보드 전체에 불이 들어왔음을 확인 했었는데 이 사실을 통해 quantization이 하드웨어의 부담을 줄었다는 것을 알 수 있다.

```
[*] Arguments: Namespace(a_bits=32, m_size=64, network='cnn', num_test_images=100, quantized=False, run_type='cpu', v_size=64, w_bits=32)
[*] Read MNIST...
[*] The shape of image: (100, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 1.0,
 'avg_num_call': 741,
 'm_size': 64,
 'total_image': 100,
 'total_time': 24.942937999999998,
 'v_size': 64}

=> Accuracy should be 1.0

[*] Arguments: Namespace(a_bits=8, m_size=64, network='cnn', num_test_images=100, quantized=True, run_type='cpu', v_size=64, w_bits=8)
[*] Read MNIST...
[*] The shape of image: (100, 28, 28)
[*] Load the network...
[*] Run tests...
random: nonblocking pool is initialized
[*] Statistics...
{'accuracy': 1.0,
 'avg_num_call': 741,
 'm_size': 64,
 'total_image': 100,
 'total_time': 53.104561000000004,
 'v_size': 64}

=> Accuracy should be 1.0

[*] Arguments: Namespace(a_bits=8, m_size=64, network='cnn', num_test_images=100, quantized=True, run_type='fpga', v_size=64, w_bits=8)
[*] Read MNIST...
[*] The shape of image: (100, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 1.0,
 'avg_num_call': 741,
 'm_size': 64,
 'total_image': 100,
 'total_time': 30.022726000000006,
```

*제목 없음...
파일(F) 편집(E) 서식(O) 보기(V)
도움말(H)
2016-13919
이다운
Windows (CRLF) UTF-8

benchmark 스크립트를 돌린 결과이다. 이미지 100개를 64x64 타일링하여 돌렸을 때 약 30초 정도가 소요됨을 알 수 있다. V0과 비교했을 때 시간차이가 거의 없다는 것을 알 수 있다. 이를 통해 구현한 V1 모듈에서 최적화 요소들이 많이 남았음을 알 수 있다.

4. Discussion

benchmark 결과 quantization를 적용하여도 시간 차이가 미미하다는 것을 보아 최적화 요소가 많이 남았음을 알 수 있다 하였다. 그 최적화 요소들에 생각해보면 먼저 cpp 코드상에서 data transfer를 int 타입으로 하였다. 하지만 quantization 결과는 8bit이므로 int 타입 대신 char 타입을 쓰면 소모 시간이 더 줄어들 것이다. 그리고 Verilog 상에서도 마찬가지로 data bit를 모두 8비트로 바꾸면 하드웨어의 부담이 줄어들 것이다.

data type을 바꾸어 최적화를 하는 quantization을 이번 프로젝트에서 직접 구현해 보았다. 이를 통해 하드웨어 가속기의 최적화에 대해 생각해 볼 수 있었고 그 방법에 대해 탐구 또한 해보았다. V2 프로젝트를 보니 V1에서 최적화 추가 구현상황을 추가하는 것으로 단순한 프로그램 구현에 그치지 않고 최적화를 위한 설계도 중요함을 이번 프로젝트를 통해 알게 되었다.