ISSN : 1226-7244 (Print)
ISSN : 2288-243X (Online)
논문번호 19-04-29

j.inst.Korean.electr.electron.eng.Vol.23,No.4,1314~1320,December 2019
http://dx.doi.org/10.7471/ikeee.2019.23.4.1314

GPU 가속기를 통한 비트 연산 최적화 및 DNN 응용

Bit Operation Optimization and DNN Application using GPU Acceleration

김 상 혁*, 이 재 흥*★

Sang Hyeok Kim*, Jae Heung Lee*★

Abstract

In this paper, we propose a new method for optimizing bit operations and applying them to DNN(Deep Neural Network) in software environment. As a method for this, we propose a packing function for bitwise optimization and a masking matrix multiplication operation for application to DNN. The packing function converts 32-bit real value to 2-bit quantization value through threshold comparison operation. When this sequence is over, four 32-bit real values are changed to one 8-bit value. The masking matrix multiplication operation consists of a special operation for multiplying the packed weight value with the normal input value. And each operation was then processed in parallel using a GPU accelerator. As a result of this experiment, memory saved about 16 times than 32-bit DNN Model. Nevertheless, the accuracy was within 1%, similar to the 32-bit model.

요 약

본 논문에서는 소프트웨어 환경에서 비트연산을 최적화 하고 DNN으로 응용하는 방법을 제안한다. 이를 위해 비트연산 최적화를 위한 패킹 함수와 DNN으로 응용을 위한 마스킹 행렬 곱 연산을 제안한다. 패킹 함수의 경우는 32bit의 실제 가중치 값을 2bit로 변환하는 연산을 수행한다. 연산을 수행할 땐, 임계값 비교 연산을 통해 2bit 값으로 변환한다. 이 연산을 수행하면 4개의 32bit값이 1개의 8bit 메모리에 들어가게 된다. 마스킹 행렬 곱 연산의 경우 패킹된 가중치 값과 일반 입력 값을 곱하기 위한 특수한 연산으로 이루어져 있다. 그리고 각각의 연산은 GPU 가속기를 이용해 병렬로 처리되게 하였다. 그 결과 HandWritten 데이터 셋에 환경에서 32bit DNN 모델에 비해 약 16배의 메모리 절약을 볼 수 있었다. 그럼에도 정확도는 32bit 모델과 비슷한 1% 이내의 차이를 보였다.

Key words : AI, Deep Learning, Neural Network, Memory Saving, Optimization

Ⅰ. 서론

최근에 딥러닝 알고리즘은 하드웨어가 발전해감에 따라 단순히 데스크톱 환경만이 아닌 모바일, 임베디드 환경에서도 딥러닝 연산을 쓸 수 있도록 연구가 진행되고 있다.

이와 같은 연구 중에서는 딥러닝의 모델에 불필요한 가중치가 많다는 점에 착안하여 필요 없는 연

* Dept. of Computer Engineering, Hanbat National University
★ Corresponding author
E-mail : jhlee@hanbat.ac.kr, Tel : +82-42-821-1208
※ Acknowledgment
This research was supported by the research fund of Hanbat National University in 2019.
Manuscript received Dec. 10, 2019; revised Dec. 24, 2019; accepted Dec. 27, 2019.

산을 줄이게 하는 가지치기(Prunning) 연산과 가중치나 입력 데이터의 크기를 줄이는 양자화(Quantization) 등의 연구가 진행되었으며, 이렇게 최적화된 모델은 정확도에 큰 영향을 미치지 않는다는 것이 증명되어있다[1-3].

다만, 양자화에 관한 연구의 경우 컴퓨터 및 GPU의 연산이 8bit 이하의 데이터의 경우 데이터 포맷 및 가속화를 지원하지 않는다는 문제로 인해 FPGA등의 외부 하드웨어를 이용하여 구현을 하며, 실제로 소프트웨어 상에서 8bit 이하의 양자화를 구현한 논문들은 거의 없다.

본 논문에서는 위의 문제점을 해결하기 위해 32bit 형태의 가중치를 2bit 데이터로 양자화하고, SIMT (Single Input Multiple Thread) 연산을 통해 연산 가속을 하는 방법을 제시한다.

II. 본론

본 장에서는 모델의 양자화를 위해 32bit 연산을 2bit로 양자화하고 8bit 메모리에 저장하는 패킹 연산과, 저장된 8bit 양자화 데이터를 순차적으로 접근하며 입력 값과 행렬 곱을 수행하는 마스킹 행렬 곱 연산을 제시한다. 해당 연산들은 전부 SIMT 연산을 통해 병렬적으로 처리되며, 이를 통해 메모리 절약 및 연산속도 향상을 꾀하였다.

양자화 모델은 학습용 모델과 실제 테스트를 위한 모델 2가지로 이루어져 있다. 학습용 모델에서는 역 전파를 통해 가중치 값 갱신이 필요하며, 가중치 값을 양자화 하여 갱신하면 정보의 손실 량이 많아 제대로 학습이 진행 되지 않으므로, 역 전파를 통해 갱신하는 가중치 값은 32bit 가중치 값을 이용한다.

학습용 모델의 구현은 실수 값으로 초기화 된 변수를 패킹 함수를 통해 양자화 및 패킹을 진행하고 패킹이 끝나게 되면 입력 값과 마스킹 행렬 곱을 수행한다. 해당 과정을 통해 나온 결과 값을 softmax 함수를 통해 결과 값과 비교하고 Cross-entropy를 비용함수로 설정하여 역 전파를 수행한다.

위의 과정을 반복하여 학습이 끝나게 되면, 실수 가중치 값을 마지막으로 양자화 한 뒤 저장한다. 실제 테스트 모델에서는 저장된 양자화 가중치 값을 불러와서 모델을 사용하므로 패킹 함수가 더는 필요하지 않다.

두 모델에 대한 전체적인 연산 수행 과정은 아래 그림 1과 같다.

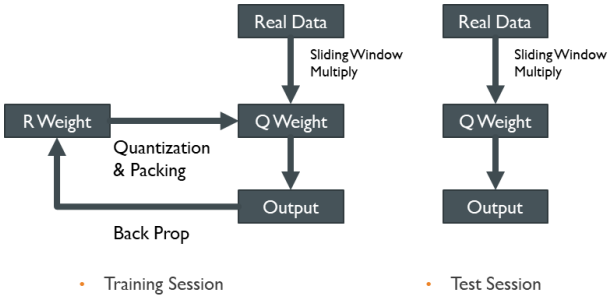

Fig. 1. Schematic of whole process.
그림 1. 전체 과정 개략도

1. 2Bit 패킹 연산

2bit 패킹 연산은 32bit의 실수 값 데이터를 2bit 값으로 양자화 하는 것이 주 역할이다. 또한 2bit의 데이터를 저장하고 연산을 수행하는 데이터 포맷이 존재하지 않으므로, 8bit의 메모리에 각 데이터를 담아야 하는데, 이 과정에서 6bit의 메모리가 남기에 이를 막기 위해 4개의 실수 값을 2bit로 양자화 하여 묶은 후, 8bit의 데이터 포맷에 저장하는 과정을 거친다.

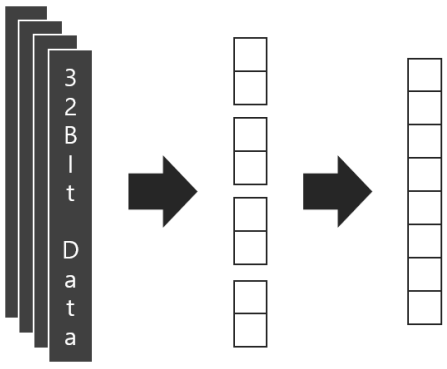

Fig. 2. Schematic of Packing function.
그림 2. 패킹 함수 동작 모식도

패킹 연산을 수행할 때, 실수 값을 양자화하기 위해 임계값 비교 연산을 사용한다. 임계값으로 쓰는 값은 0.004를 이용하였으며, 이는 0의 평균값을 가지고 0.01의 표준편차를 가진 정규분포 함수의 데이터를 3등분하는 구간이다. 만약 모델의 변수 초기 값이 다르다면 임계값을 변경하여 적절하게 양자화가 진행되도록 하는 것이 학습 속도 향상에 도움이 된다.

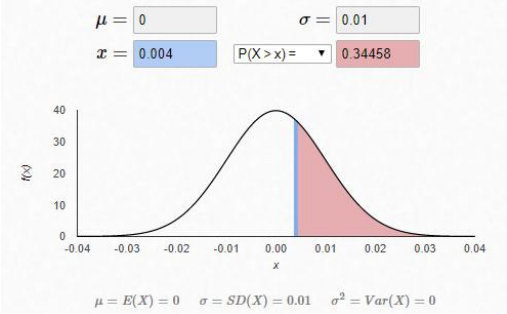

Fig. 3. Probability that x deviates from greater than 0.004 in the normal distribution with mean 0 and standard deviation 0.01.

그림 3. 평균 0, 표준편차 0.01인 정규분포에서 x가 0.004 보다 큰 값을 벗어날 확률 값

아래의 수식 (1)과 같이 임계값 비교 연산을 통해 양자화를 진행한 뒤에는 양자화 된 가중치 값을 1개의 8bit 데이터 형식에 저장하는 패킹 과정을 수행한다.

$$pack_b = \begin{cases} 0b10 & \text{if } pack_b > +0.004 \\ 0b00 & \text{if } pack_b < -0.004 \\ 0b01 & otherwise \end{cases} \quad (1)$$

위의 과정을 구현한 패킹 함수 알고리즘에 대한 Pseudo Code는 아래 표 1과 같다.

Table 1. Packing Function.

표 1. 패킹 연산

Algorithm	Packing Fucntion
Input	: 32bit [M, N] Size Array I
Output	: 8bit [M/4, N] Size Array O

```
1   for x=0 to M/4-1 do
2       for y=0 to N-1 do
3           for z=0 to 3 do
4               pack_num = 0
5               if I[x*4+z][y] < -0.004
6                   pack_num |= 0b00 << (3-z)*2
7               else if I[x*4+z][y] > 0.004
8                   pack_num |= 0b10 << (3-z)*2
9               else
10                  pack_num |= 0b01 <<(3-z)*2
11          O[x,y] = pack_num
```

1, 2, 3번 줄의 for문은 각 4개의 32bit 가중치 값을 그룹화 하여 1개의 8bit 변수에 담기 위해 참조하는 행렬의 값을 참조하기 위한 indexing 코드이며, 행을 기준으로 그룹화를 진행하기에 출력 값인 O행렬은 입력 행렬인 I의 행 값보다 4배 더 작은 값을 가진다. 따라서 전체 출력 행렬 O의 크기는 입력 행렬 [M, N]에서 M을 4로 나눈 [M/4, N]의 크기를 갖게 된다.

4번 줄의 pack_num은 8bit 데이터를 임시로 저장하는 변수를 의미하며 이를 0으로 초기화한다.

5, 7, 9 번 줄은 임계값 비교용 if 연산으로 위의 수식의 부등식을 코드로 구현한 것이다.

각 임계값 비교 if 연산의 처리결과인 6, 8, 10번 줄은 pack_num 변수에 부등식 결과 값을 집어넣는 코드로 << 연산자는 비트를 왼쪽으로 이동시키고 0을 집어넣는 산술 시프트 연산자이다.

같은 줄에 있는 |= 연산자는 OR 연산자로 0으로 초기화 되어있는 pack_num 변수에 자신의 위치에 해당하는 비트에만 0 또는 1의 값을 집어넣고 그 이외의 비트는 기존의 값 그대로 놔두는 연산을 수행한다.

마지막 줄은 양자화를 거치고 패킹까지 처리를 한 변수를 출력행렬 O에 집어넣는 것을 의미한다. pack_num에는 연산을 마친 양자화 데이터가 big-endian 방식에 맞추어 저장되어 있다. 또한 각 패킹 연산의 모든 for문은 SIMT 연산을 통해 GPU 내부 커널에서 병렬로 처리되게 구현하였다. 그로 인해 실제 표 1의 알고리즘을 그대로 CPU에서 구현하는 것 보다 짧은 연산 시간을 가진다. 패킹 함수에 대한 전체적인 연산 예시는 아래 그림 4와 같다.

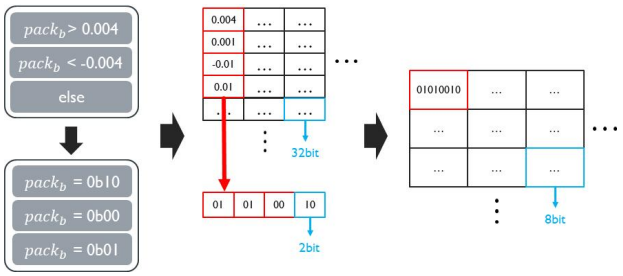

Fig. 4. Packing Function example.

그림 4. 패킹 함수 동작 예시도

2. 마스킹 행렬 곱 연산

마스킹 행렬 곱 연산은 양자화와 패킹이 진행된 가중치 값과 입력 값을 곱하는 연산으로 양자화 된 2bit 가중치 4개가 모여서 하나의 8bit 데이터 포맷에 저장되어있기에 일반적인 행렬 곱을 사용할 경우 연산 결과에 문제가 생긴다. 따라서 이를 처리

해줄 특별한 행렬 곱 연산이 필요하다.

행렬 곱 연산을 수행할 땐 입력 값 4개와 패킹된 8bit 가중치 하나를 곱하여 더하도록 연산을 수행한다. 8bit 가중치 하나에는 4개의 2bit 데이터가 big-endian 방식으로 저장되어 있으므로, 입력 값과 대응되는 양자화 가중치에 접근할 수 있도록 가중치 데이터에 마스크 값을 씌워 연산을 진행하도록 구현한다. 양자화 가중치 값인 [0b00, 0b01, 0b10]은 각각 정수 값 [-1, 0, 1]에 대응하며 연산 이후의 출력 데이터 값은 32bit값을 가진다.

이를 구현한 Masking Matrix Multiplication 알고리즘에 대한 Psuedo Code는 다음과 같다.

Table 2. Masking Matrix Multiplication.
표 2. 마스킹 행렬 곱

Algorithm	Masking Matrix Multiplication
Input	: 32bit [M, K] Size Array I 2bit [K/4, N] Size Array W
Output	: 32bit [M, N] Size Array O
1	*for* x=0 *to* M-1 *do*
2	*for* y=0 *to* N-1 *do*
3	*for* z=0 *to* K/4-1 *do*
4	*for* w=0 *to* 3 *do*
5	O[x,y] += I[x][z*4+w] *(W[z][y] >>((3-w)*2&0b11)-1)

1, 2, 3, 4 번의 각 줄은 입력 값과 가중치 값의 곱을 수행하기 위해 필요한 데이터에 접근하기 위한 Indexing 코드로 입력 값의 경우 열을 우선하여 접근해야 하고 가중치 값의 경우 행을 우선하여 접근해야 한다. 또한 4개의 입력 열마다 1개의 가중치 행에 접근하여야 하며, 1개의 가중치 행은4번의 곱 연산을 수행하도록 Indexing 한다.

5번 줄은 마스킹 행렬 곱을 수행하는 코드로 >> 연산은 비트를 오른쪽으로 이동시키고 왼쪽에 0을 채워 넣는 산술 시프트 연산자이다. &는 AND 연산자로 마스킹을 씌우는 과정이며 해당하는 비트만 원본 값을 출력시키고 그 이외의 값은 전부 0으로 바꾼다.

산술 시프트와 AND 마스킹 연산을 통해서 나온 데이터는 양자화 된 가중치 값인 [0b00, 0b01, 0b11] 값이 나오게 된다. 이 값들은 각각 정수형으로 [0, 1, 2]의 값을 가지며 곱할 때는 실제 대응하는 정수 값으로 맞추기 위하여 1을 뺀 값을 취해준다. 그렇게 되면 각 값은 [-1, 0, 1]이 되며 이는 각 대응하는 정수 값에 맞게 된다.

위의 연산이 전부 끝나면 그 값을 출력행렬 해당 위치에 더해주는 작업을 반복하면 마스킹 행렬 곱 연산이 종료된다. 이 과정 또한 패킹 연산과 마찬가지로 각 for문은 전부 SIMT 연산을 통해 GPU 커널 단에서 병렬로 동작하므로 4중의 for문이 필요한 연산임에도 불구하고 빠른 시간 안에 연산을 끝마칠 수 있다. 또한, 대응하는 정수 값이 [-1, 0, 1] 이기에 곱 연산 없이 비교 연산을 통한다면 덧셈 연산과 뺄셈 연산만으로도 해당 코드를 구현할 수 있다.

마스킹 행렬 곱에 대한 전체적인 연산 예시는 아래 그림 5와 같다.

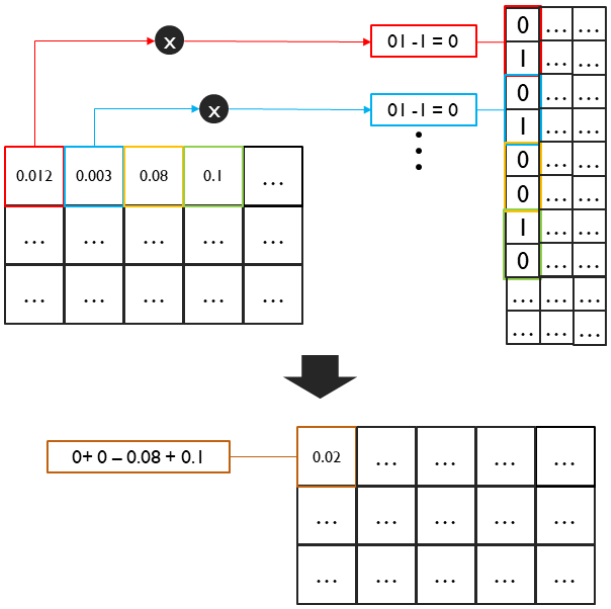

Fig. 5. Masking Matrix Multiplication Example.
그림 5. 마스킹 행렬곱 연산 예시

III. 실험

본 논문에서 구현한 양자화 모델을 테스트하기 위해 HandWritten 손 글씨 데이터를 사용하여 분류하는 모델을 만들어 동일한 층으로 구성된 32bit 모델을 만들어 메모리와 속도의 차이를 확인하였다.

HandWritten 데이터 셋은 알파벳에 대한 손글씨를 분류할 수 있게 만든 데이터로 28*28 사이즈의

사진 30만장 가량으로 이루어져 있다. 해당 데이터의 약 30%를 테스트용으로 사용하고 나머지 70%를 학습에 이용하였다.

학습 모델로는 3개의 층으로 이루어진 DNN 모델을 사용하였으며, 각각의 층은 [784, 256], [256, 128], [128, 26] 의 크기를 가지고 있다.

양자화 모델의 경우 행의 크기가 32bit 모델의 가중치 크기에서 행의 크기가 4배가량 압축되기 때문에 각 층마다 [196, 256], [64, 128], [32, 26] 의 크기를 갖는다.

각 모델의 활성화 함수로는 Sigmoid 함수를 사용하였다. 가중치 갱신은 AdamOptimizer를 이용하였고, 이때의 학습률은 0.001로 하였으며 비용 계산 함수로는 softmax cross entropy를 사용하였다.

테스트 환경으로는 i7-6700K intel CPU, GTX 1060 3GB VRAM GPU, 32GB RAM, Window 10 Pro 의 환경에서 실험을 진행하였다.

위의 조건대로 행한 모델 테스트의 정확도 결과는 아래와 같다.

```
------- 0 th ---------
2bit_prediction :  [19 11  0  3  1 20 14 15 17 20]
real value :  tf.Tensor([19 11  0  3  1 20 14 15 16 20], shape=(10,), dtype=int32)

------- 1 th ---------
2bit_prediction :  [19 18 25 15 17 11 14 10  4 22]
real value :  tf.Tensor([19 18 25 15 17 11 14 10  4 22], shape=(10,), dtype=int32)

------- 2 th ---------
2bit_prediction :  [ 7 22  4  1 10 18 13 18 14 18]
real value :  tf.Tensor([ 7 22  4  1 10 18 13 18 14 18], shape=(10,), dtype=int32)

------- 3 th ---------
2bit_prediction :  [ 3 14 14 12  0  7 19 12  9  4]
real value :  tf.Tensor([ 9 14 14 12  0  7 19 12  9  4], shape=(10,), dtype=int32)

------- 4 th ---------
2bit_prediction :  [20 18 18 18 20  9 12  0  4  0]
real value :  tf.Tensor([20 18 18 18 20  9 12  0  4  0], shape=(10,), dtype=int32)

2bit Accuracy :  0.92555

------- 0 th ---------
my 32bit_prediction :  [14 10 14 13  1 18  9 20 17 12]
real value :  tf.Tensor([14 10 14 13  1 18  9 20 17 12], shape=(10,), dtype=int32)

------- 1 th ---------
my 32bit_prediction :  [ 9 18 18 18 17 15 24 20 13 24]
real value :  tf.Tensor([ 9 18 18 18 17 15 24 20 13 24], shape=(10,), dtype=int32)

------- 2 th ---------
my 32bit_prediction :  [22 14 14 20 10 14 14 20 18 19]
real value :  tf.Tensor([22 14 14 20 10 14 14 20 18 19], shape=(10,), dtype=int32)

------- 3 th ---------
my 32bit_prediction :  [19 19 18 14 20  4 19 15 12  2]
real value :  tf.Tensor([19 19 18 14 20  4 19 15 12  2], shape=(10,), dtype=int32)

------- 4 th ---------
my 32bit_prediction :  [14  2 20  2 14 18 13 17 20 18]
real value :  tf.Tensor([14  2 20  2 14 18 13 17 20  6], shape=(10,), dtype=int32)

my 32bit Accuracy :  0.93721664
```

Fig. 6. Top : 2bit Model Accuracy
Bottom : 32 bit Model Accuracy.
그림 6. 위 : 2bit 모델의 정확도
아래 : 32bit 모델의 정확도

그림 6의 결과를 보면 2bit 양자화 모델의 경우 모든 가중치가 양자화 되었음에도 정확도가 32bit과 큰 차이 없는 1%를 보였다.

다음에는 메모리 사용량을 확인하였다. 본 논문에서 구현한 모델과 32bit 모델 또한 전부 GPU 메모리에서 동작하므로, 컴퓨터의 RAM 메모리가 아닌 GPU 내부 RAM 메모리의 값을 확인해야 한다. 이를 확인하기 위한 함수로 CUPY에서 지원하는 mempool 함수를 사용하였다.

```
----------initial----------
0
0
----------packing----------
59392
59392
```

```
----------initial----------
0
0
----------packing----------
947200
947200
```

Fig. 7. Left : Memory Usage for 2bit Model
Right :Memory Usage for 32bit Model.
그림 7. 왼쪽 : 2bit 모델 메모리 사용량
오른쪽 : 32bit 모델 메모리 사용량

```
Calc_time :  0.3893760144710541
Calc_time :  0.3399679958820343
Calc_time :  0.4208639860153198

Calc_time :  0.5737919807434082
Calc_time :  0.5181440114974976
Calc_time :  0.5286399722099304

Calc_time :  0.9577280282974243
Calc_time :  0.9912319779396057
Calc_time :  0.7833600044250488

Calc_time :  1.4306559562683105
Calc_time :  1.4358400106430054
Calc_time :  1.277951955795288
```

```
Calc_time :  1.563647985458374
Calc_time :  2.0347840785980225
Calc_time :  1.4318079948425293

Calc_time :  2.6863040924072266
Calc_time :  2.6470398902893066
Calc_time :  2.6757121086120605

Calc_time :  5.673984050750732
Calc_time :  7.2775678634643555
Calc_time :  5.182688236236572

Calc_time :  12.125184059143066
Calc_time :  12.268671989440918
Calc_time :  12.078335762023926
```

Fig. 8. Left : 2bit Model Running Time, Right : 32bit Model Running Time, Batch size : 20, 40, 60, 80.
그림 8. 왼쪽 : 2bi 모델 계산 시간, 오른쪽 : 32bit 모델 계산 시간, 배치 사이즈 : 20, 40, 60, 80

메모리 점유율 예측 점유 값은 32bit 모델의 경우 (784*256 + 256*128 + 128*26)*4 byte로 이를 계산해보면 947,200 byte의 메모리를 차지하는 것으로 예측 할 수 있다. 2bit 모델의 경우 196*256 + 64*128 + 32*26 byte로 이를 계산해보면 59,200 byte

가 된다. 이는 실제로 계산된 GPU 메모리 점유율을 나타낸 그림 7과 큰 차이가 없다.

마지막으로 시간속도 측정으로, Intel i7-6700K @4.00GHz CPU와 GTX1060 VRAM 3GB를 가진 데스크톱 환경에서 측정하였으며, 행렬 곱 연산의 속도를 늘리기 위한 아무런 기법도 적용하지 않았다. 예측되는 계산시간 증가량은 메모리의 접근횟수를 약 4배가량 줄였고, 곱 연산의 경우도 복잡한 float 연산을 없앴기에 추가적인 2~3배가량의 속도 향상을 예상하였다. 배치사이즈를 늘려가며 시간 속도를 3회씩 측정하였으며, 이 조건하에 측정된 시간은 그림 8과 같다.

시간측정과 메모리측정 결과 본 논문에서 제시한 양자화 모델은 32bit 동일한 모델에 대해 약 16배의 메모리 절약과 약 4~8배의 계산속도가 상승하였다. 이를 정리한 그래프는 아래 그림 9와 같다.

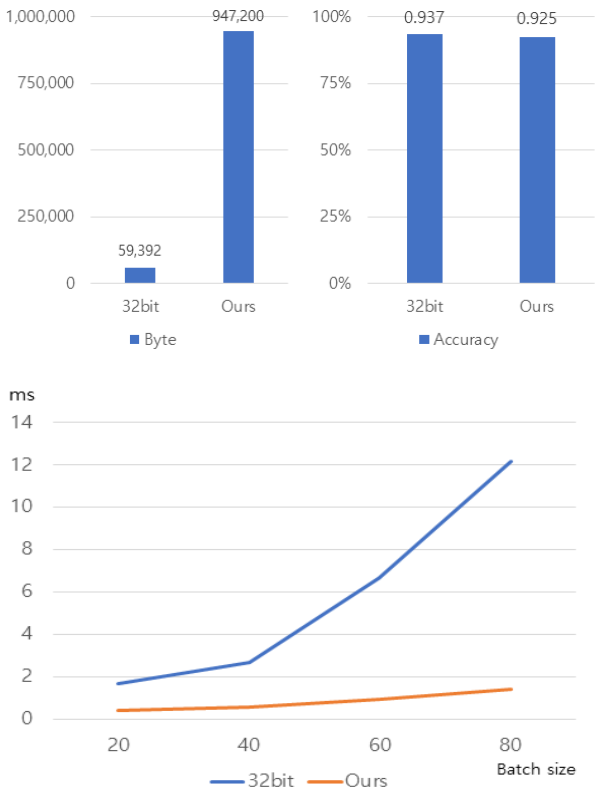

Fig. 9. Comparison between 32bit and ours.
그림 9. 32비트 모델과의 비교

Ⅳ. 결론 및 향후 연구

본 논문에서 제안한 모델은 추가적인 하드웨어 없이 2bit로 양자화를 하고 계산하는 방법을 제안하였다. 제안된 방법을 검증하기 위해 DNN모델을 만들고 HandWritten 모델을 이용하였으며 기존 32bit 모델과 1%의 정확도 하락이 있었으나 약 16배의 메모리 절약과 4~8배가량 속도 상승 효과를 보았다.

향후 연구에서는 행렬 곱 연산을 보다 빠르게 하는 기법을 적용하고 DNN뿐 아닌 CNN의 Fully Connected 영역에도 적용을 하여 제안된 성능보다 높은 연산속도 및 다양한 곳에 응용할 수 있는 모델로 발전시켜나갈 것이다.

References

[1] M. Courbariaux, Y. Bengio and J. David, “Binary Connect: Training Deep Neural Networks with binary weights during propagations,” 2015. https://arxiv.org/abs/1511.00363

[2] C. Zhu, S. Han, H. Mao and W. J. Dally, “Trained ternary quantization,” *International Conference on Learning Representations,* 2017.

[3] S. Han, H. Mao and W. J. Dally, “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding,” *NIPS Deep Learning Symposium*, 2015.

[4] Nikola Sakharnykh, “Maximizing Unified Memory Performance in CUDA,”https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda

[5] J. Choi, P. I-Jen, C. Z. Wang, S. Venkataramani, V. Srinivasan and K. Gopalakrishnan, “Bridging the Accuracy Gap for 2-bit Quantized Neural Networks(QNN),” https://arxiv.org/abs/1807.06964

[6] S. Uhlich, L. Mauch, K. Yoshiyama, F. Cardinaux, J. A. Garcia, S. Tiedmann, T. Kemp and A. Nakamura, “Differentiable Quantization of Deep Neural Networks,” https://arxiv.org/abs/1905.11452

[7] M. Rastegari, V. Ordonez, J. Redmon and A. Farhadi., “Xnor-net: Imagenet classification using binary convolution neural networks,” *European Conference on Computer Vision,* pp.525-542, 2016.

[8] J. Choi, S. Venkataramani, V. Srinivasan, K. Gopalakrishana, Z. Wang, and P. Chuang, “Accurate And Efficient 2-bit Quantized Neural Networks,”

https://sysml.cc/doc/2019/168.pdf
[9] F. Li, B. Zhang and B. Liu, “Ternary Weight Networks,” https://arxiv.org/abs/1605.04711
[10] N. S. Sohoni, C. R. Aberger, M. Leszczyynski, J. Zhamg and C. Re “Low Memory Neural Network Training: A Technical Report,” https://arxiv.org/abs/1904.10631

BIOGRAPHY

Sang Hyeok Kim (Member)

2018 : BS degree in Computer Engineering, Hanbat National University.
2018~Now : MS degree course in Computer Engineering, Hanbat National University.

Jae Heung Lee (Member)

1983 : BS degree in Electrical Engineering, Hanyang University.
1985 : MS degree in Electronic Engineering, Hanyang University.
1994 : PhD degree in Electronic Engineering, Hanyang University.
1989~Now : Professor in Dept. of Computer Engineering, Hanbat National University.