Mirali Ahmadli
20170847

Final Exam Report

I used the same logic as in project 3. I preprocessed given input and kernel tensors as it was in project 3 and the multiplied them to get the output matrix, then post-processed it as I did in project 3. Matrix multiplication part was the convolution operation and performance measurements have done based on that.

To confirm the correctness of the output, I implemented a simple python script that uses tf.nn.conv2d and compared each index of the resulting tensor to my output with small threshold values. And in the end, I calculated the sum of absolute values of differences over every value in TensorFlow and my results.

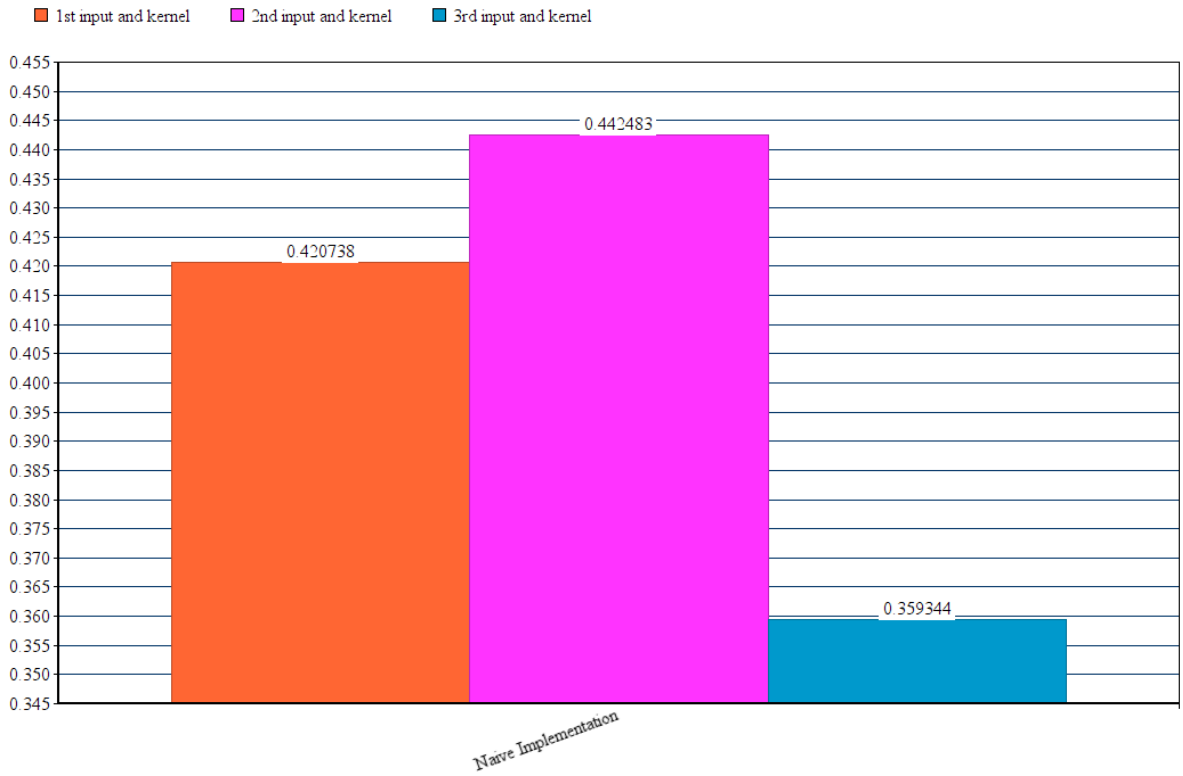
The script is also included in the submission.

Problem 1:

I implemented a simple matrix calculation using 3 loops.

Elapsed time:

1: 0.420738
2: 0.442483
3: 0.359344

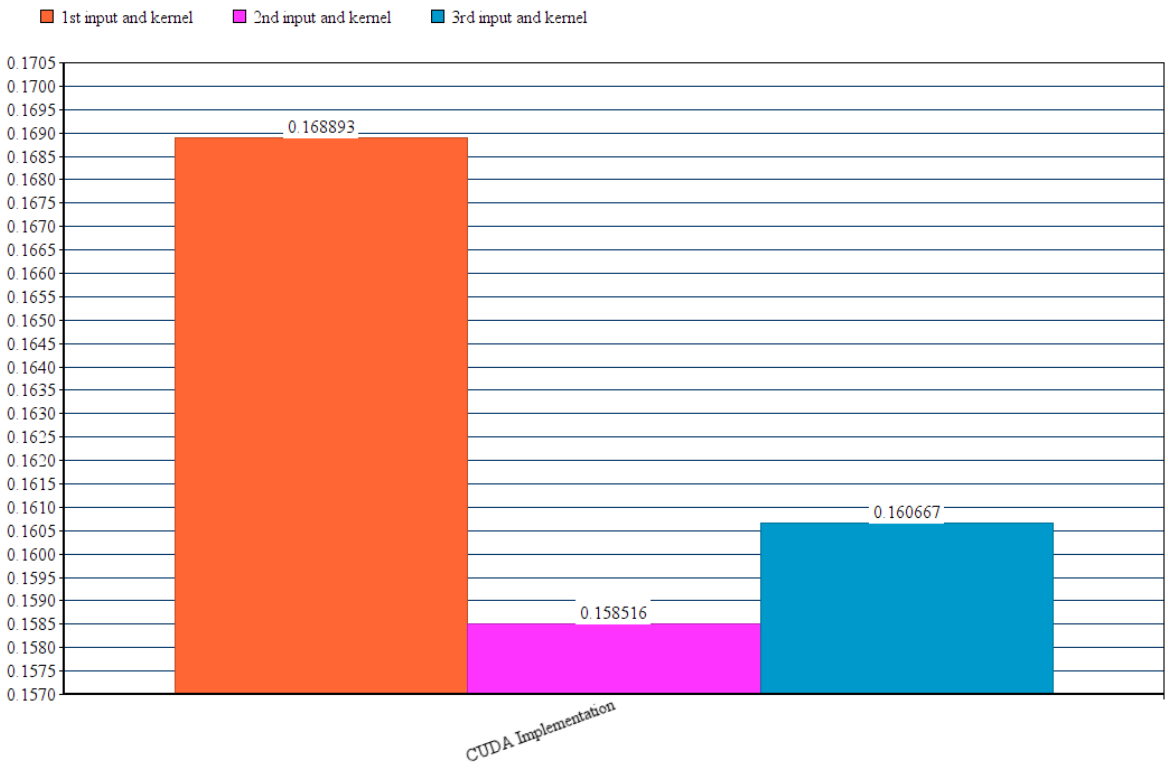


Problem 4:

I used the code from proect3.

Elapsed time:

1: 0.168893
2: 0.158516
3: 0.160667



Problem 2:

INT32:

SCALE1 = 85899344.000000

SCALE2 = 4294967296.000000

Elapsed time:

1: 0.365689

2: 0.388812

3: 0.298364

NRMSE:

1: 0.485985

2: 3.744504

3: 0.314973

Overhead time:

1: 0.004787

2: 0.004254

3: 0.008531

INT16:

SCALE1 = 1310.699951

SCALE2 = 65535.000000

Elapsed time:

1: 0.380087

2: 0.347232
3: 0.290997

NRMSE:

1: 0.003020
2: 3.744657
3: 0.317251

Overhead time:

1: 0.004637
2: 0.004132
3: 0.007954

INT8:

SCALE = 5.080000
SCALE2 = 256.000000

Elapsed time:

1: 0.341927
2: 0.326417
3: 0.269504

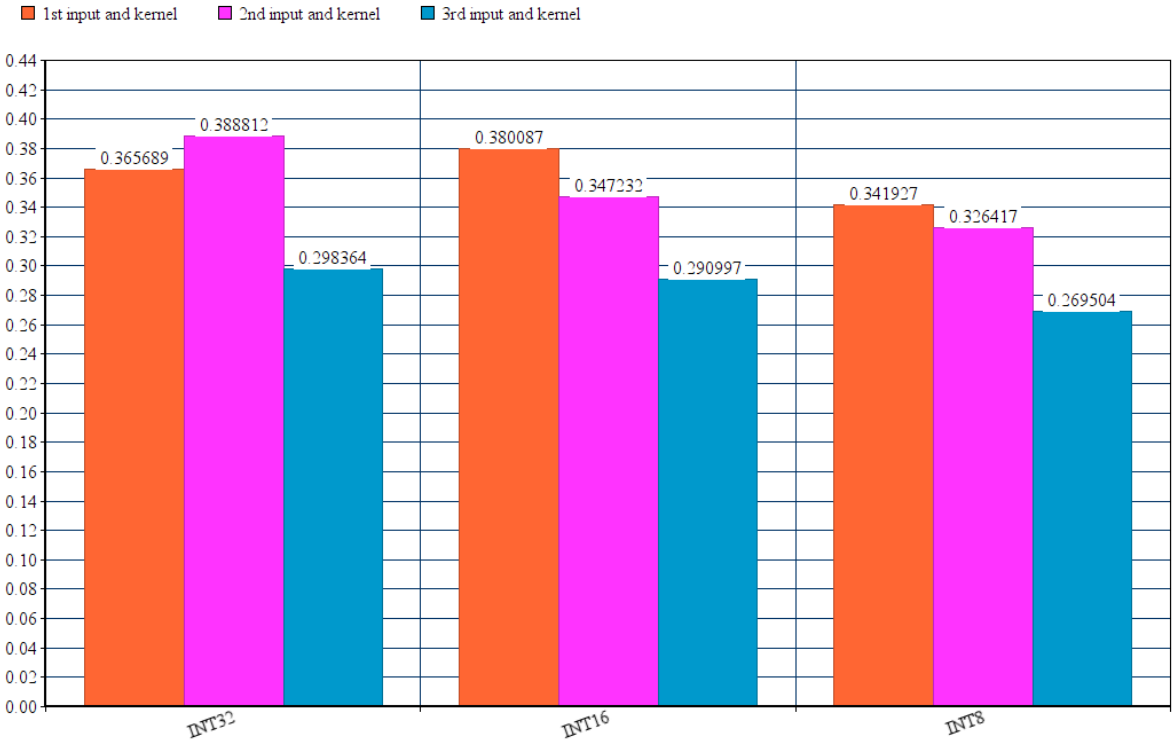
NRMSE:

1: 0.537890
2: 3.049910
3: 8.010243

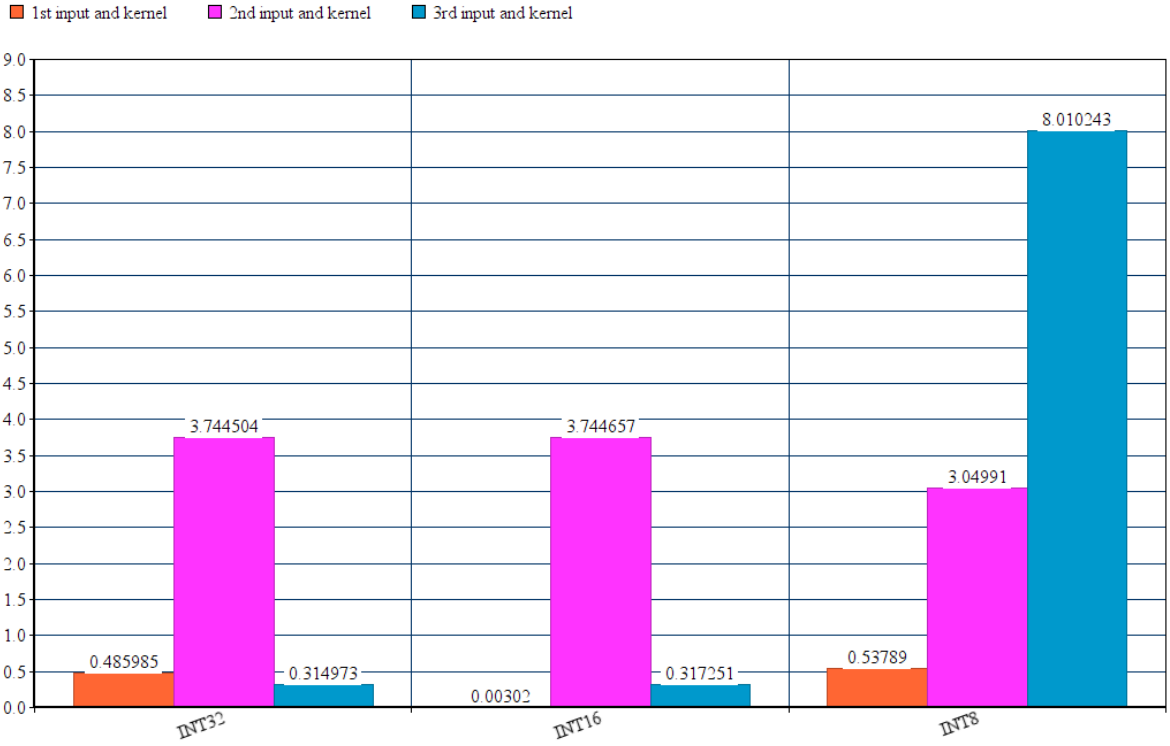
Overhead time:

1: 0.019487
2: 0.019266
3: 0.042094

Elapsed time graph



NRMSE graph



Problem 3:

I used the same code from project 3, changed AVX instructions appropriately for quantization parts.

NOTE: I implemented the pthreads as I did in the previous project. Despite it calculated values correctly, the value that passed through pthread_join is shown as zero. I could not fix that, so I stick with single-threaded implementation.

FP32:

- Elapsed time:
  - 1: 0.366783
  - 2: 0.342527
  - 3: 0.296927

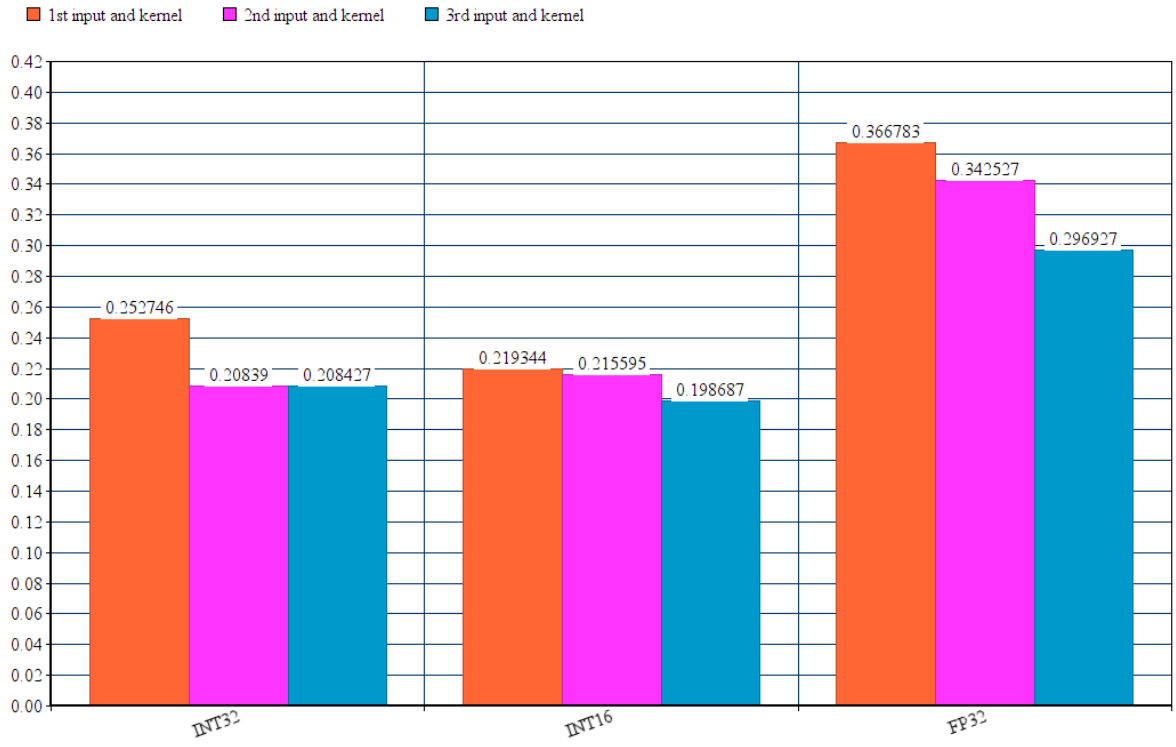
INT32:

- SCALE1 = 85899344.000000
- SCALE2 = 4294967296.000000
- Elapsed time:
  - 1: 0.252746
  - 2: 0.208390
  - 3: 0.208427
- NRMSE:
  - 1: 0.485985
  - 2: 3.744504
  - 3: 0.314973
- Overhead time:
  - 1: 0.009071
  - 2: 0.016292
  - 3: 0.019333

INT16:

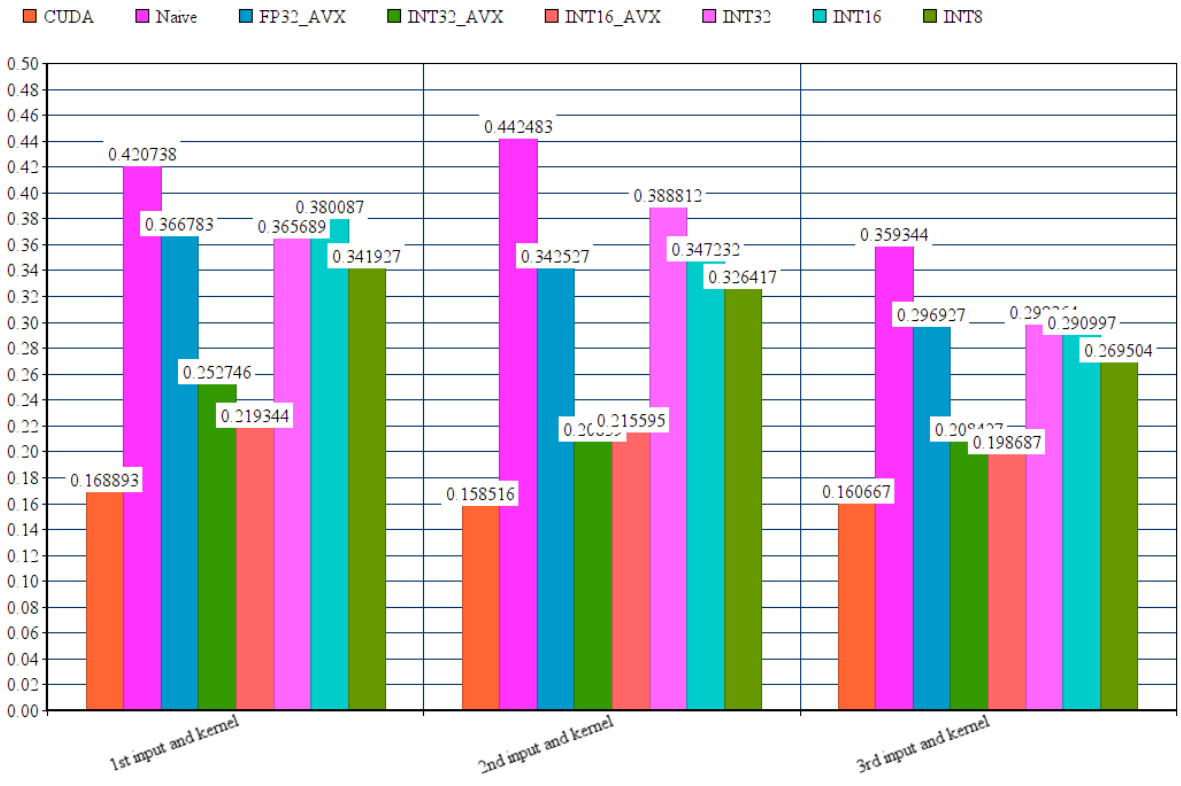
- SCALE1 = 1310.699951
- SCALE2 = 65535.000000
- Elapsed time:
  - 1: 0.219344
  - 2: 0.215595
  - 3: 0.198687
- NRMSE:
  - 1: 0.003020
  - 2: 3.744657
  - 3: 0.317251
- Overhead time:
  - 1: 0.003835
  - 2: 0.003524
  - 3: 0.007461

Elapsed time graphs



Discussion:

Elapsed times for each different implementation



For choosing a scalar value for quantization, I looked at the range of input, kernel, and output results from prob1 and prob4. Input is in the range between -25.0 and 25.0, the kernel is in the range between -0.6 and 0.5, and output is in the range between -27.0 and 25.0. Using these values, for kernel and input, I assigned different scaling factors. I used the following formula to obtain scaling factor:
scale_input = (max_val - min_val) / (max_input - min_input)
scale_kernel = (max_val - min_val)

I have tried several methods, but all except this one performed poorly.
Max_val and min_val are the maximum and minimum possible integers with given precision P (2^(P-1) - 1 and -2^(P-1)).

To avoid overflow, I rounded and kept the range while scaling, and I saved multiplication results in 2*P sized memory [1]. As I know that output also has similar range as input, we do not need more size.

From the results, we can see that CUDA performs much better than others as it uses GPU and GPU parallelization. The next best-performing implementations are AVX implementation with quantization and vectorized operations. INT8 quantization outperforms FP32 AVX because it is much faster to multiply int8s than fp32s. Moreover, we can see that applying quantization reduces the runtime a lot (~50% in AVX cases and ~20-30% in simple implementation). Also, our scaling factor for int16 and int32 works very well. For int8 it is harder to quantize and we, of course, lose more precision during conversion than others. We can see that the scaling factors that we chose do not perform well with the INT8 setup. It is most probably because of either overflow or computation mistake as the only 3rd given case returned value 8.

Also, the overhead time is negligible in every conversion, therefore we can say that lowering precision for speedup can be extremely useful if you have limited or poor computation power.

In general, while choosing the correct scaling factor, it is very important to consider the current ranges of given inputs and required size to save multiplication value as it is also discussed in article [1].

While checking recent papers[4] and frameworks [3], I saw that channel-wise quantization is widely used but due to a shortage of time, I could not implement and try it. I believe applying it or modifying the current formula would reduce accuracy loss.

In conclusion, the quantization of floating-point to lower precision speeds up the running time while not giving up much accuracy deduction with the correct choice of floating-point. Using GPU and GPU parallelization outperforms naive implementation (~2.5x times) and all other versions as expected. Furthermore, performance for AVX can be much better by using pthreads (unfortunately, I could not debug it) and NRMSE can be reduced by choosing better scaling factors or applying different scaling factors for each channel.

References:

[1] https://nervanasystems.github.io/distiller/quantization.html
[2] https://www.youtube.com/watch?v=VsGX9kFXjbs
[3] https://www.tensorflow.org/lite/performance/quantization_spec
[4] https://openreview.net/pdf?id=H1lBj2VFPS