CS492: Systems for Machine Learning

Final Project

Junoh Moon

Korea Advanced Institute of Science and Technology

June 19, 2020

1. Develop Convolutional layer in C/C++ and measure the elapsed time exclusively for running the convolution operations.

   Answer:

	testcase (group 1)			
	1	**2**	**3**	**average**
elapsed time (ms)	24.010000	23.59800	9.548000	19.052000

Table 1: Experiment results of problem 1.

2. Quantize the operands in a lower precision and analyze the performance-accuracy tradeoff

   Answer: Since the scale S in $\mathbb{Q} = S \times \mathbb{R}$, where $\mathbb{Q}$ represents the quantized integer and $\mathbb{R}$ represents the real number, played a key-role in minimizing information loss during quantization — narrowing the numeric range — , it was required to pick a proper S. Considering the importance of the problem, the S was set dynamically for given data rather than hand-picked. At first, it was designed to share S between an input and a kernel. However, as shown in Figure 1, the distribution problem arose.

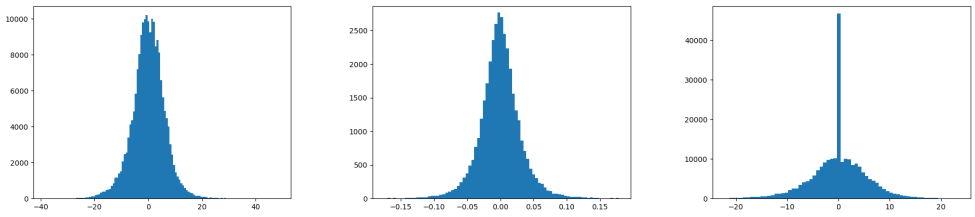

Figure 1: Plotted histograms. Each histogram represents input, kernel, and input+kernel respectively from the left. The data came from the first testcase of group 1.

Although both of input and kernel was distributed as *bell-curve*, variances differed; while most values of the input were in $[-0.15, 0.15]$, those of the kernel were in $[-30, 30]$. Thus, if we naively use only one S, then the S would scale the values in the right figure in Figure 1. This would result in overflow input values or underflow kernel values into 0. This tendency was observed in every testcase.

To solve the overflow/underflow problem in quantization, the input and kernel had each S respectively as S_{input} and S_{kernel}. Now then S was defined as follows to fully utilize the given integer range:

$$S = \frac{q_{max} - (-q_{max})}{r_{max} - r_{min}} \tag{1}$$

where r_{min} and r_{max} represent the smallest and the biggest number in the input or kernel, q_{max} represents the biggest number of the type for quantization.

Given S_{input} and S_{kernel}, result r_{result} was calculated as below.

$$\begin{aligned} r_{result} &= r_{input} \times r_{kernel} \\ &\approx \frac{q_{input}}{S_{input}} \times \frac{q_{kernel}}{S_{kernel}} \\ &\approx \frac{1}{S_{input}S_{kernel}} q_{input}q_{kernel} \\ &\approx \frac{1}{S_{input}S_{kernel}} q_{result} \end{aligned} \tag{2}$$

By Equation 1, real numbers are uniformly mapped into quantized numbers. This approach offered us a small overhead with fair results [1]. Thus, real numbers could be minimized to integers without meaningful accuracy loss.

The problem was, however, Convolutional Layer consists of MAC operations, and hence the result of MAC operations would be overflowed after calculation. For example, when given n-bit integers, $k \times k$ kernel, and the number of the input channels, c, then the accumulator should be at least

$$2n + log_2(c \cdot k^2) \tag{3}$$

bits to avoid overflow. One of the simple ideas to solve this problem is making a accumulator type large. Consequently, the result of 8-bit convolutional layer was set to 32-bit, that of 16-bit was set to 64-bit, and that of 32-bit was set to 64-bit. 32-bit inputs and kernels were mapped into 64-bit accumulators (2× precision) unlike others (4 × precision) because the runtime environment did not support 128-bit integer.

bit length	input scale	kernel scale	conv2d	dequantization	quantization	NRMSE
8→32	3.982	436.630	18.545ms	0.362ms	2.801ms	0.95%
16→64	1,019.353	111,777.267	35.977ms	0.452ms	2.431ms	0.90%
32→64	66,804,233.333	7,325,443,333.333	40.495ms	0.256ms	2.397ms	9.56%

Table 2: Averaged experiment results of problem 2.

As shown in Table 2, the results in 8-bit and 16-bit showed outstanding performances. Both of them showed less than 1% errors. In contrast, 32-bit integer quantization showed relatively poor performance. One of the reasons may be due to relatively small accumulator bit-length. The overall quantization overhead was neglectable. In every case, the portion of the overhead was less than 15%, and thus calculating convolutional layer still took a dominent portion.

From a point of view of throughput, small bit-length integer had a tendency to return the results faster, but not significantly faster than floating points. One possible explanation is the code was fully optimized by the compiler, and hence the calculation is pipelined. In the given environment (Intel Skylake Xeon 4-core processors, GCC 5.4.0), O3 optimization implicitly vectorized codes into SSE instructions. Therefore, the gap between floats and integers might be reduced.

3. CPU vectorization with lower precision

Answer: In this experiment, the given codes were accelerated by explicit AVX2 intrinsics

and a *pthread* library. The input matrices were divided into sub-matrices that correspond to one element in the result matrix, and thus multiple elements in the result matrix were calculated in parallel. Moreover, the multiplication function was parallelized with AVX2 intrinsics, so n elements (n depends on the numeric type) could be calculated simultaneously.

numeric type	input scale	kernel scale	conv2d	dequantization	quantization	NRMSE
INT16	1,019.35	111,777.27	24.488ms	0.135ms	2.192ms	9.56%
INT32	66,804,233.33	7,325,443,333.33	18.864ms	0.133ms	2.236ms	9.56%
FP32	66,804,233.33	7,325,443,333.33	18.394ms	0.144ms	2.356ms	0.00%

Table 3: Experiment results using AVX2 and pthread.

From a point of view of precision, the errors of all integers was 9.56% (Table 3). This result was same as the 32-bit experiment in the second problem (Table 2). Thus, it is supposed that these high error rates were due to the overflow problem since the intrinsics API did not support high precision accumulator types, so an accumulator type was fixed to the source type.

On the other hand, the elapsed time reduced compared to the previous code, especailly in integers. (Table 2). For integers, the time was reduced to 1.47× and 2.15× respectively. But, not a significant improvement occured in floating points (1.04×). These results implied that floating points were implicitly vectorized in O3 flag, but not in integer vice-versa.

4. GPU vectorization

Answer:

	testcase (group 1)			
	1	2	3	average
elapsed time	0.03749ms	0.03574ms	0.03562ms	0.03628ms

Table 4: Experiment results of problem 4

5. Performance-Accuracy Tradeoff Analysis and Discussion

Answer:

In this section, we collected four experiments — executing conv2d functions, measuring the elapsed time, and calculating the error if required — and plotted into two graphs (Figure 2). The left graph showed the elapsed time during invoking `conv2d` function. The result showed that manually parallelized experiments (AVXInt16, AVXInt32, AVXFloat) were as fast as the baseline (problem 1). Integer types, however, did not satisfy our expectation. One assumption is, as described in the previous section, floating points were already vectorized (SSE instructions) to exploit ILP by a compiler, while integers were not.

In contrast, the GPU offered 525× remarkable speedup over the baseline. Due to SIMD-like architecture and hundreds of execution-units, it overpowered CPUs. However, GPUs are coprocessors of a CPU, and thus the data must be copied to the device memory. In this discussion, copying overhead (170 ms) took the dominent portion in the program. Thus,

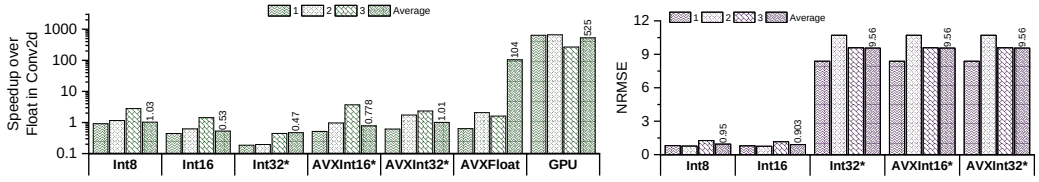

Figure 2: Speedup over Problem 1 and its corresponding NRMSE graphs. Types with asterisk (*) mean their accumulator bit-widths are not wide enough to avoid overflow

this implies that GPU memory management is the key factor for showing the performance without interference.

In the perspective of precision, quantized low-precision bit-widths (Int8, Int16) showed less than 1.0% accuracy loss. On the other hand, Int32, AVXInt16, and AVXInt32 showed a relatively poor performance. The point was they had two common things. First, due to the lack of language-level and library-level supports, an overflow countermeasure could not be provided to them. Whereas more then double-bit-length was required for fully avoiding the overflow (Equation 3), only under $2\times$ bit-length types were provided to them. Thus, the first point implied that enough bits for acculation is mandatory to keep the accuracy.

Second, the error rate of AVXInt16 was equal to that of AVXInt32 and that of Int32. This implied that limited ranges could be overcome by setting a scaling factor dynamically to fully utilize a quantized type, but as already mentioned, keeping quantized values correctly is more important.

To sum up, adding hardware-level resources gave us the remarkable improvement. However, if the device is fixed, a proper quantization technique would show a reasonable performance without losing accuracy; this technique required a dynamic scaling to fully mapping the input into whole range of the quantized type for keeping the information as much as possible and a large-sized accumulator not to lose information. Consequently, we could reach the optimal point in the tradeoff by quantizing to 8-bit (Equation 1) and using 32-bit accumulator to avoid overflow.

References

[1] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713.