# Systems for Machine Learning: Final Report

Jinwoo Kim 20160171

June 20, 2020

- Avg $\equiv$ Averaged over three trials for each sample ($n = 9$ total) in department machine
- Row major ordering assumed in every implementation
- Training-centric papers [Lin+17][Par+17b][Xia+18][Lin+17][Sha+16] not discussed

## 1 Develop a Convolution function in C/C++

### 1.1 Results

Convolution elapsed time (avg): 0.169sec

## 2 Quantization, performance-accuracy tradeoff

### 2.1 Results

NRMSE (avg): calcaulated with $X$ quantized convolution output and $Y$ floating-point convolution output

| | NRMSE (avg) | Convolution (avg sec) | Quantization (avg sec) |
|---|---|---|---|
| `INT32` | $4.90\times10^{-8}$ | 0.137 | $3.25\times10^{-3}$ |
| `INT16` | $9.93\times10^{-5}$ | 0.132 | $2.49\times10^{-3}$ |
| `INT8` | $1.22\times10^{-2}$ | 0.126 | $2.58\times10^{-3}$ |

### 2.2 Choosing scaling factor

For reported experiments, I used the following scales:

| Scale | Input | Kernel |
|---|---|---|
| `INT32` | $2^{19}$ | $2^{19} \times 500$ |
| `INT16` | $2^{7}$ | $2^{7} \times 500$ |
| `INT8` | 1 | $1 \times 500$ |

I provide the justification below.

- Given a datatype (e.g., `INT8`), quantized values are bounded by its min/max values (-128 to 127 for `INT8`). In terms of minimizing information loss, we want as many as input values to fall into this bounded range; that is, we want a scaling factor that transforms given data values into the range of the datatype.
- The problem is that the DNN input and kernel weights typically have different distribution. Below are the statistics of sample tensors;
  - Sample 1: Input $0.0731 \pm 0.0137 \iff$ Kernel$-0.000486 \pm 0.000168$
  - Sample 2: Input $-0.252 \pm 0.0202 \iff$ Kernel$-0.000358 \pm 0.0000600$
  - Sample 3: Input $0.145 \pm 0.0281 \iff$ Kernel$-0.000937 \pm 0.0000171$
- If we were to scale them by the same factor, one of below scenarios happen:
  - If we pick a scaling factor that minimize input information loss, most of the kernel values will be quantized to zero, resulting in zero convolved values.

  - If we pick a scaling factor that minimize kernel information loss, most of the input values will be clamped to either `INT?_MAX` or `INT?_MIN`, resulting in intolerable computation error.

- Therefore, to scale the input and kernel values to a similar range, we scale kernel values $500\times$ more than input values (see below).
  - Sample 1: Input $0.0731 \pm 0.0137 \iff$ Kernel($500\times$) $-0.243 \pm 0.0840$
  - Sample 2: Input $-0.252 \pm 0.0202 \iff$ Kernel($500\times$) $-0.179 \pm 0.0300$
  - Sample 3: Input $0.145 \pm 0.0281 \iff$ Kernel($500\times$) $-0.468 \pm 0.00857$

- Then, what is left is to determine a scaling factor for input. To search in logarithmic scale, I simply let the input scaling factor $S = 2^N (N \in \mathbf{Z}^+)$ and sought for exponent $N$ that minimizes NMRSE. Below log-log plot shows NRMSE (averaged over samples) of `INT8`, `INT16` and `INT32` versus input scaling factor.

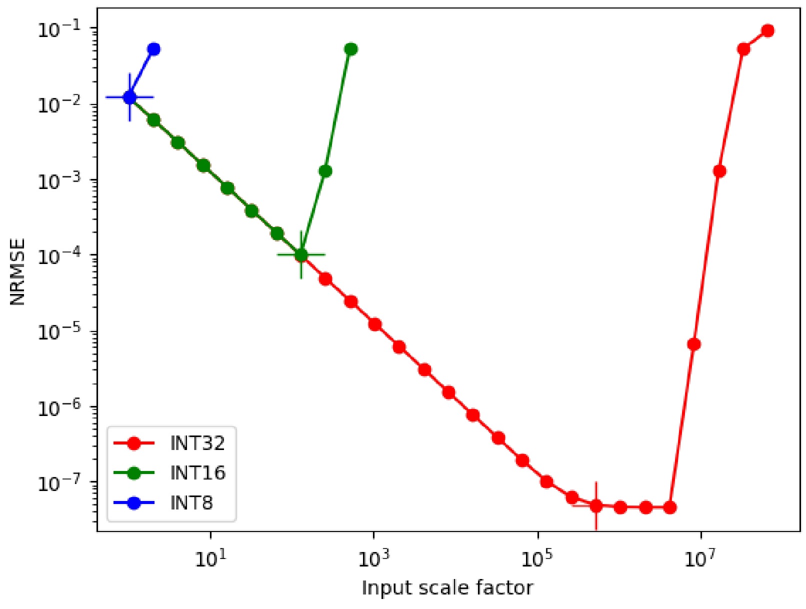

Figure 1: V-curve of NRMSE versus scaling factor

- To some degree, an increase in scaling factor results in a decrease in information loss (by quantization), and therefore, NRMSE decreases. But once the quantized values begin to be clamped by `INT?_MAX` or `INT?_MIN`, NRMSE explosively increases.

- From this observation, I chose large scaling factors as possible (marked by $+$) that doesn't lead to clamping. Note that `INT32` tolerates larger scaling factors than `INT16` due to its larger bitwidth, and the same holds between `INT16` and `INT8`.

## 3 CPU vectorization with lower precision

### 3.1 Results

(Quantization: used same code from problem 2, so not measured)

| | NRMSE (avg) | Convolution (avg sec) |
|---|---|---|
| `FP32` | $4.80\times10^{-8}$ | 0.0187 |
| `INT32` | $4.90\times10^{-8}$ | 0.0202 |
| `INT16` | $9.93\times10^{-5}$ | 0.0131 |

## 4 GPU vectorization

### 4.1 Results

NRMSE (avg): $4.61 \times 10^{-8}$

Convolution elapsed time (avg): 0.00421sec

## 5 Performance-Accuracy Tradeoff

Word count: 768, excluding captions and headers, measured at TeXcount

### 5.1 Visualization

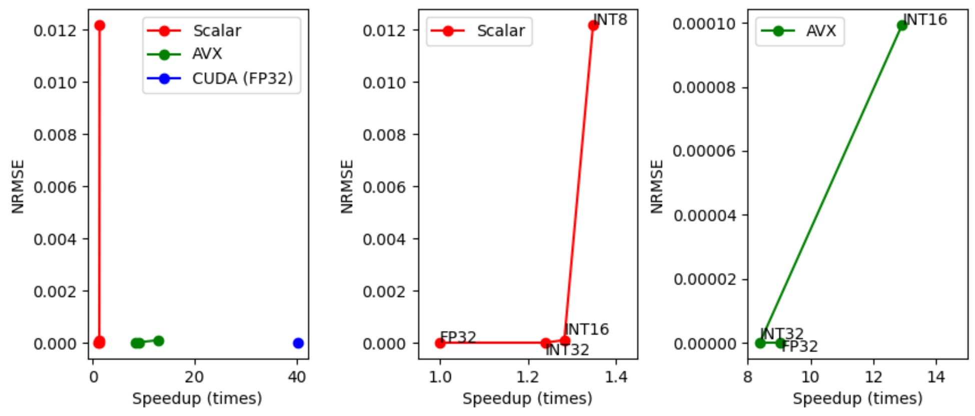

Figure 2: Left shows summarized NRMSE versus speedup relative to scalar-`FP32`, and middle/right show quantization-only / quantization with AVX+pthread.

### 5.2 Tradeoff Analysis

#### 5.2.1 Implementation strategies worth mentioning

- Quantization
  - Scaled values that exceed quantization range were clamped to `INT?_MAX` or `INT?_MIN`.
  - As scaling was always performed to floating-point values (before quantization/after restoration), scaling factor $S$ was declared as a floating point value.
- Convolution
  - To avoid over/underflow, $N$-bit integer multiplication was always held after sign-extending to larger-bit integers, both in scalar (`int64_t`) and AVX+pthread ($2N$-bit).
  - In CUDA, I used an input stationary dataflow to maximize on-chip data reuse [Che16].

#### 5.2.2 Comparing accuracy loss and performance

In discussed papers, following factors are primarily considered for evaluation of DNN accelerators:

- Performance (throughput/latency) & energy
- Area & power
- Accuracy loss (EIE[Che16], BitFusion[Sha+17], SCNN [Par+17a])

As we have no model nor simulator for inferring energy consumption and area [Par+17a][Jou+17], we first evaluate based on speedup, accuracy loss and their tradeoff. In this regard, **input stationary GPU parallelization achieves $\sim 40\times$ speedup while not compromising accuracy loss**, so it seems to be the best option [Che16].

### 5.2.3 Considering energy efficiency

When we weight energy efficiency, other implementations could be preferred over GPU parallelization which exploits 32-bit floating-point multiplication. **Floating-point multiples are energy- and area-inefficient**; 16-bit floating-point multiples use $6\times$ more energy and $6\times$ more area than eight-bit integer multiples [Jou+17]. **using GPU itself can be problematic in terms of energy** because the whole input and kernel in DRAM (or swap disk) should be loaded into off-chip memory and on-chip storage before computation [Sha+17].

Long story short, considering energy efficiency, **I suggest that AVX-`INT16` can be considered a competitive candidate to CUDA in terms of performance-accuracy tradeoff.** Justification is presented below.

First, we exclude quantized scalar operations from consideration because they provide marginal speedup of $< 1.4\times$. This leaves us CPU-based parallelization with AVX+pthreads, which provide a better tradeoff of $\sim 10\times$ speedup over quantized scalar in cost of same NRMSE. Interestingly, **quantization does not always guarantee a speedup over floating-point operation; AVX-`INT32` takes 8% more time to `FP32`, while NRMSE is 2% higher.** Why does AVX-`INT32` fail to achieve a better tradeoff? Explanation is given in the next subsubsection.

### 5.2.4 Why AVX-`INT16`?: Inefficiency of `INT32` quantization

The main reason seems to be the cumbersome AVX2 operations needed for accurate integer multiplication. **To prevent over/underflow or truncation of multiplication results, quantization-based accelerators like Bit-Fusion assign larger bitwidth for partial-/final-sum (e.g. 32-bit) than operands (e.g. 2- and 4-bit)** [Sha+17]. Although such a process is effectively achieved by hardware implementation involving bit shifts [Li+18] [Sha+17], implementation with AVX operations is tricky.

Take `INT32` for example. Given operand `_mm256` vectors $[x_1 ... x_8]$ and $[y_1 ... y_8]$ of 4-byte values, out goal is to multiply-and-add the values (psum= $x_1 * y_1 + ... + x_8 * y_8$). For precision, the values are first sign-extended to 8-byte and compose new `_mm256` vectors $[x_1 ... x_4]$, $[x_5 ... x_8]$, $[y_1 ... y_4]$ and $[y_5 ... y_8]$. Only then, SIMD integer multiplication can be done to compute psum accurately. In my implementation, **this costs ten AVX2 SIMD instructions for `INT32`, while `FP32` only requires four.** Even if AVX2 integer multiplication is faster than floating point, the mere number of operations for processing makes it pointless.

To overcome, we need to exploit quantization more: **With `INT16`, we can pack 16 quantized values in a `_mm256` vector.** Although we still need 10 AVX2 operations for a multiplication, the number of iterations is reduced by half because 16 values are processed each iteration. This gives us 10/16 required opr-per-value, comparable to $4/8$ of `FP32`. **With these factors, AVX-`INT16` finally achieve a $\sim 1.5\times$ speedup over AVX-`FP32`, while compromising a tolerable increase in accuracy loss (NRMSE) from 4.80$\times 10^{-8}$ to 9.93$\times 10^{-5}$.**

Therefore, considering energy efficiency and performance-accuracy tradeoff, AVX-`INT16` can be regarded as a competitive candidate to CUDA since it provides **less energy consumption than CUDA and better performance-accuracy tradeoff than AVX-`INT32` and AVX-`FP32`**.

### 5.2.5 Potential of `INT8`

Then, a natural question is whether we can achieve better performance with 8-bit quantized AVX-`INT8`. **Discussed papers indicate that 8-bit quantization is good enough for inference [Jou+17] [Sha+17], and TensorFlow Lite now supports full post-training quantization (both activation and weight) of CNN models to 8-bit[1].** As we can pack thirty-two 8-bit values in one `_mm256` vector, a further speedup over `FP32`, `INT32` and `INT16` is certainly expected in CPU environment. NRMSE of $1.22\times 10^{-2}$ measured in scalar quantization also makes `INT8` an appealing quantization bitwidth. Despite such appealing tradeoff, to determine whether or not to accept AVX-`INT8`, we need to evaluate it using full-CNN task accuracy.

[1]4- or 1-bit compression can be done with quantization-aware training [Han+16], but we focus on post-training quantization here.

### 5.2.6 Can we exploit sparsity?

As I only applied naïve fixed-point quantization, there is room for improvement. A method could be switching to other compression method such as lossy compression in INCEPTIONN [Li+18], but it is unclear whether the compression scheme enables effective multiplication because it is communication-oriented. Instead, I explore the **possibility that sparsity of quantized values can be leveraged for further acceleration** [Han+16][Par+17a]. The fraction of zero-quantized values (sparsity) in `INT32`, `INT16` and `INT8` are presented in Figure 3.

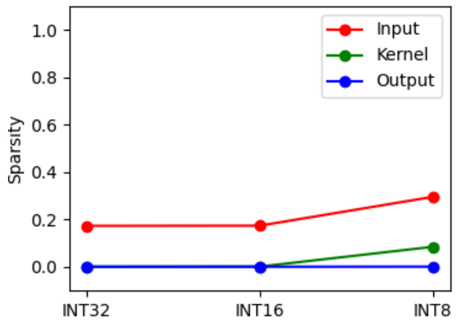

Figure 3: Sparsity of input, kernel and output for each quantization. (pooled over all samples)

Although both weight and activation sparsity is too low to leverage sparse acceleration, **note that when we decrease scaling factor, the sparsity is expected to increase: This gives us an additional tradeoff in accuracy-performance, as reduced scaling factor compromises an increased NRMSE (shown in Figure 1)**. For example, when we set the input scaling factor to $S = 1$ for `INT16`, input sparsity jumps to 23% at the cost of increased NRMSE to 0.0049. It should also be noted that the **sample tensors are from the first layer of CNN, while deep layers tend to show much higher sparsity** [Par+17a]. Therefore, a further sparsity-based acceleration of deep layers could be made by controlling scaling factor, with increased NRMSE as a tradeoff.

## 6 References

[Che16] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne. "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks". In: *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*. 2016, 262–263.

[Han+16] Song Han et al. *EIE: Efficient Inference Engine on Compressed Deep Neural Network*. 2016. arXiv: `1602.01528 [cs.CV]`.

[Sha+16] H. Sharma et al. "From high-level deep neural models to FPGAs". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016, pp. 1–12.

[Jou+17] Norman P. Jouppi et al. *In-Datacenter Performance Analysis of a Tensor Processing Unit*. 2017. arXiv: `1704.04760 [cs.AR]`.

[Lin+17] Yujun Lin et al. *Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training*. 2017. arXiv: `1712.01887 [cs.CV]`.

[Par+17a] Angshuman Parashar et al. *SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks*. 2017. arXiv: `1708.04485 [cs.NE]`.

[Par+17b] J. Park et al. "Scale-Out Acceleration for Machine Learning". In: *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2017, pp. 367–381.

[Sha+17] Hardik Sharma et al. *Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Networks*. 2017. arXiv: `1712.01507 [cs.NE]`.

[Li+18] Y. Li et al. "A Network-Centric Hardware/Algorithm Co-Design to Accelerate Distributed Training of Deep Neural Networks". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018, pp. 175–188.

[Xia+18] Wencong Xiao et al. "Gandiva: Introspective Cluster Scheduling for Deep Learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 595–610. ISBN: 978-1-939133-08-3. URL: `https://www.usenix.org/conference/osdi18/presentation/xiao`.