Supplementary Material: Towards Unified INT8 Training for Convolutional Neural Network

Feng Zhu[1] Ruihao Gong[1,2] Fengwei Yu[1] Xianglong Liu[2*] Yanfei Wang[1]
Zhelong Li[1] Xiuqi Yang[1] Junjie Yan[1]
[1]SenseTime Group Limited
[2]State Key Laboratory of Software Development Environment, Beihang University
{zhufeng1, yufengwei, wangyanfei, lizhelong, yangxiuqi, yanjunjie}@sensetime.com
{gongruihao, xlliu}@nlsde.buaa.edu.cn

A. Proof of Theorem 1

Assumption 1. *f_t is convex;*

Assumption 2. $\forall \mathbf{w}_p, \mathbf{w}_q \in \mathbb{S}, \|\mathbf{w}_p - \mathbf{w}_q\|_\infty \leq D_\infty$.

Proof. Considering the update for ith entry of weight,

$$w_{t+1,i} = w_{t,i} - \eta_{t,i}\hat{g}_{t,i} \tag{1}$$

we have

$$\begin{aligned}(w_{t+1,i} - w_i^*)^2 &= (w_{t,i} - \eta_{t,i}\,\hat{g}_{t,i} - w_i^*)^2 \\ &= (w_{t,i} - w_i^*)^2 - 2(w_{t,i} - w_i^*)\eta_{t,i}\,\hat{g}_{t,i} + \eta_{t,i}^2\,\hat{g}_{t,i}^2\end{aligned} \tag{2}$$

Rearrange the equation, and divide $2\eta_{t,i}$ on both side as $\eta_{t,i}$ is none-zero,

$$\begin{aligned}\hat{g}_{t,i}(w_{t,i} - w_i^*) = &\frac{1}{2\eta_{t,i}}(w_{t,i} - w_i^*)^2 + \frac{\eta_{t,i}}{2}\hat{g}_{t,i}^2 \\ &- \frac{1}{2\eta_{t,i}}(w_{t+1,i} - w_i^*)^2\end{aligned} \tag{3}$$

The error of quantized gradients is defined as

$$\epsilon_{t,i} = g_{t,i} - \hat{g}_{t,i} \tag{4}$$

Replace $\hat{g}_{t,i}$ in the (3) with $g_{t,i}$ and $\epsilon_{t,i}$, and we can get that

$$\begin{aligned}&g_{t,i}(w_{t,i} - w_i^*) = \\ &\quad\frac{1}{2\eta_{t,i}}[(w_{t,i} - w_i^*)^2 - (w_{t+1,i} - w_i^*)^2)] \\ &\quad+ \epsilon_{t,i}(w_{t,i} - w_i^*) + \frac{\eta_{t,i}}{2}(g_{t,i} - \epsilon_{t,i})^2\end{aligned} \tag{5}$$

According to assumption 1,

$$f_t(\mathbf{w}_t) - f_t(\mathbf{w}^*) \leq \mathbf{g}_t^\top(\mathbf{w}_t - \mathbf{w}^*) \tag{6}$$

So combine the (5) and (6), sum over the d dimensions of $\mathbf{w}$ and the T iterations, then the regret

$$\begin{aligned}R(T) \leq &\sum_{t=1}^{T}\sum_{i=1}^{d}(\frac{1}{2\eta_{t,i}}[(w_{t,i} - w_i^*)^2 - (w_{t+1,i} - w_i^*)^2] \\ &+ \epsilon_{t,i}(w_{t,i} - w_i^*) + \frac{\eta_{t,i}}{2}(g_{t,i} - \epsilon_{t,i})^2) \\ = &\sum_{i=1}^{d}[\frac{1}{2\eta_{1,i}}(w_{1,i} - w_i^*)^2 - \frac{1}{2\eta_{T,i}}(w_{T+1,i} - w_i^*)^2] \\ &+ \sum_{t=2}^{T}\sum_{i=1}^{d}(\frac{1}{2\eta_{t,i}} - \frac{1}{2\eta_{t-1,i}})(w_{t,i} - w_i^*)^2 \\ &+ \sum_{t=1}^{T}\sum_{i=1}^{d}[\epsilon_{t,i}(w_{t,i} - w_i^*) + \frac{\eta_{t,i}}{2}(g_{t,i} - \epsilon_{t,i})^2]\end{aligned} \tag{7}$$

Combine (7) with the assumption 2, and we can further relax the above (7) to

$$\begin{aligned}R(T) \leq &\sum_{i=1}^{d}\frac{D_\infty^2}{2\eta_{1,i}} + \sum_{t=2}^{T}\sum_{i=1}^{d}(\frac{1}{2\eta_{t,i}} - \frac{1}{2\eta_{t-1,i}})D_\infty^2 \\ &+ \sum_{t=1}^{T}\sum_{i=1}^{d}[\epsilon_{t,i}(w_{t,i} - w_i^*) + \frac{\eta_{t,i}}{2}(g_{t,i} - \epsilon_{t,i})^2]\end{aligned} \tag{8}$$

Assume that all layers have the same learning rate, then

$$R(T) \leq \frac{d\,D_\infty^2}{2\eta_T} + \sum_{t=1}^{T}\epsilon_{\mathbf{t}}(\mathbf{w_t} - \mathbf{w}^*) + \sum_{t=1}^{T}\frac{\eta_t}{2}(\mathbf{g_t} - \epsilon_{\mathbf{t}})^2 \tag{9}$$

Based on Cauchy's inequality and assumption 2, we finally

*corresponding author

get

$$
\begin{aligned}
R(T) &\leq \frac{d\,D_\infty^2}{2\eta_T} + \sum_{t=1}^{T} \|\epsilon_\mathbf{t}\| \cdot \|\mathbf{w_t} - \mathbf{w}^*\| + \sum_{t=1}^{T} \frac{\eta_t}{2} \|\mathbf{g_t} - \epsilon_\mathbf{t}\|^2 \\
&\leq \frac{d\,D_\infty^2}{2\eta_T} + D_\infty \sum_{t=1}^{T} \|\epsilon_\mathbf{t}\| + \sum_{t=1}^{T} \frac{\eta_t}{2} \|\hat{\mathbf{g}}_\mathbf{t}\|^2
\end{aligned}
\tag{10}
$$

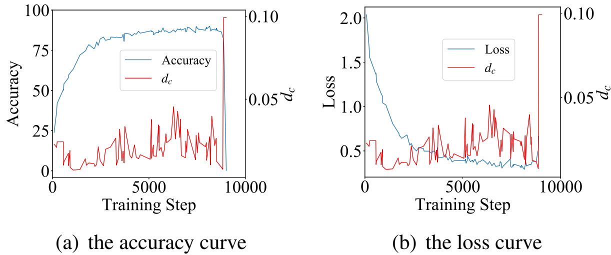
Thus the average regret

$$
\frac{R(T)}{T} \leq \underbrace{\frac{d\,D_\infty^2}{2T\eta_T}}_{(1)} + \underbrace{\frac{D_\infty}{T} \sum_{t=1}^{T} \|\epsilon_\mathbf{t}\|}_{(2)} + \underbrace{\frac{1}{T} \sum_{t=1}^{T} \frac{\eta_t}{2} \|\hat{\mathbf{g}}_\mathbf{t}\|^2}_{(3)} \tag{11}
$$

□

B. INT8 Training Stability

We train the quantized MobileNetV2 on CIFAR-10 dataset to explore the relationship between cosine distance d_c and training stability. As shown in Figure 1, when d_c increases to a certain level, the whole training crashes. There exists strong correlation between d_c and training stability, which proves that cosine distance can effectively reflect the influence of gradient quantization on the convergence.



(a) the accuracy curve (b) the loss curve

Figure 1. Model crashes when d_c exceeds limits.

We plot the accuracy and the loss curve of MobileNetV2 training on CIFAR-10 dataset and ResNet-50 training on ImageNet dataset to show the stability of INT8 training. From Figure 2 and Figure 3, we can see that our method makes INT8 training smooth and achieves accuracy comparable to FP32 training. The quantization noise increases exploratory ability of INT8 training since the quantization noise at early stage of training could make the optimization direction more diverse, and with properly reduced learning rate, INT8 training sometimes even converge faster than FP32 training.

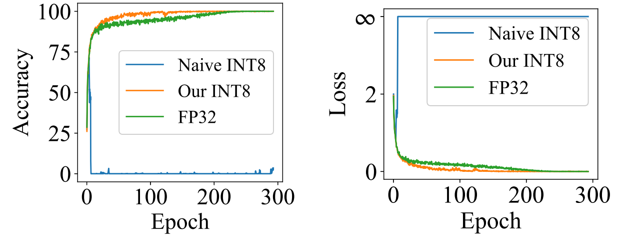
C. INT8 Convolution Speedup Algorithm

C.1. INT8 Convolution

On NVIDIA GPUs with Pascal architectures (such as GP102, GP104, and GP106), the new 8-bit integer 4-element dot product with accumulation (DP4A) [4] instruction is supported. This enables the NVIDIA GeForce GTX 1080Ti (based on GP102) to achieve a peak integer throughput of 44 Tera Operations Per Second (TOPS), while the peak float throughput is only 11 Tera Float Operations Per Second (TFLOPS).

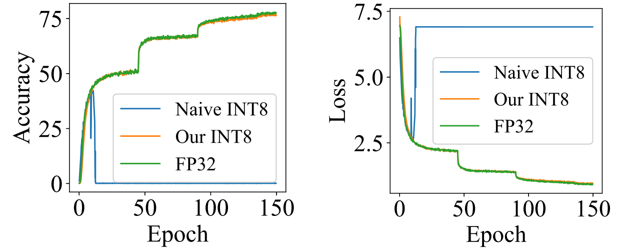
Since the release of cuDNN 6.0 [3], INT8 inference is supported but the INT8 backward process is not implemented. So we use the DP4A instruction to implement the INT8 backward process by ourselves. Moreover, we find that the quantization process before INT8 convolution computation is pretty time-consuming as the quantization needs to read and write the whole data. In order to reduce the overhead that quantization brings, we fuse the quantization process with the convolution computation (quantization-convolution fused kernel). In Figure 4, we can see that the combination of quantization and convolution could avoid one extra global memory read and write effectively. Thus we rewrite the INT8 forward and backward process using this quantization-convolution fused kernel and achieve a significant speed-up.

In our implementation, we transpose the data layout into NC4HW so that we can use the DP4A instruction to conduct the convolution computation. We use the $prmt$ instruction in Parallel Thread Execution and Instruction Set Architecture (PTX ISA) [4] to transpose the data efficiently. This $prmt$ instruction picks four arbitrary bytes from two 32-bit registers, and reassembles them into a 32-bit destination register. Figure 5 shows that one thread can transpose data



(a) the accuracy curve (b) the loss curve

Figure 2. Comparison of INT8 training and FP32 training on CIFAR-10 using MobileNetV2.



(a) the accuracy curve (b) the loss curve

Figure 3. Comparison of INT8 training and FP32 training on ImageNet using ResNet-50.

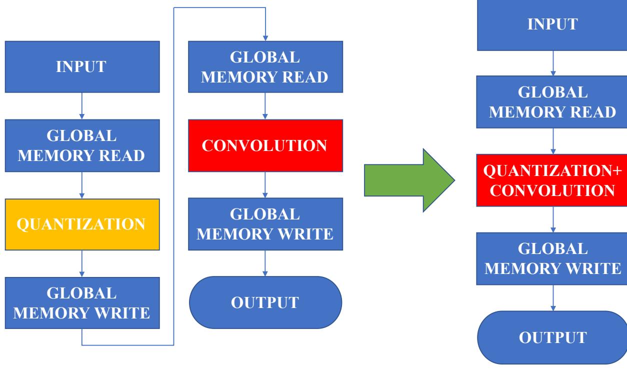


Figure 4. Quantization-convolution fused kernel avoids one extra global memory read and write.

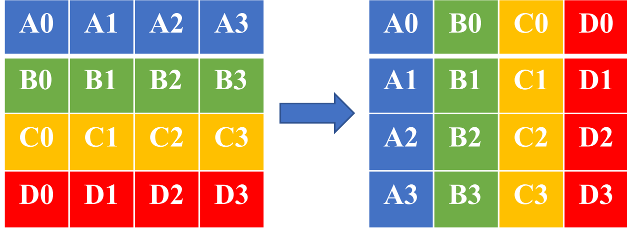


Figure 5. 4×4 8-bit integer block transpose in a thread using *prmt* instruction.

in 4×4 8-bit integer block by using 12 *prmt* instructions with shared memory. The transpose implementation code is listed below.

```
int regLDG[4]; int4 regPRMT; int tmp;
asm volatile("prmt.b32 %0, %1, %2,
    0x0040;" : "=r"(regPRMT.x) :
    "r"(regLDG[0]), "r"(regLDG[1]));
asm volatile("prmt.b32 %0, %1, %2,
    0x0040;" : "=r"(tmp) : "r"(regLDG[2]),
    "r"(regLDG[3]));
asm volatile("prmt.b32 %0, %1, %2,
    0x5410;" : "=r"(regPRMT.x) :
    "r"(regPRMT.x), "r"(tmp));
asm volatile("prmt.b32 %0, %1, %2,
    0x0051;" : "=r"(regPRMT.y) :
    "r"(regLDG[0]), "r"(regLDG[1]));
asm volatile("prmt.b32 %0, %1, %2,
    0x0051;" : "=r"(tmp) : "r"(regLDG[2]),
    "r"(regLDG[3]));
asm volatile("prmt.b32 %0, %1, %2,
    0x5410;" : "=r"(regPRMT.y) :
    "r"(regPRMT.y), "r"(tmp));
asm volatile("prmt.b32 %0, %1, %2,
    0x0062;" : "=r"(regPRMT.z) :
    "r"(regLDG[0]), "r"(regLDG[1]));
asm volatile("prmt.b32 %0, %1, %2,
    0x0062;" : "=r"(tmp) : "r"(regLDG[2]),
    "r"(regLDG[3]));
asm volatile("prmt.b32 %0, %1, %2,
    0x5410;" : "=r"(regPRMT.z) :
    "r"(regPRMT.z), "r"(tmp));
asm volatile("prmt.b32 %0, %1, %2,
    0x0073;" : "=r"(regPRMT.w) :
    "r"(regLDG[0]), "r"(regLDG[1]));
asm volatile("prmt.b32 %0, %1, %2,
    0x0073;" : "=r"(tmp) : "r"(regLDG[2]),
    "r"(regLDG[3]));
asm volatile("prmt.b32 %0, %1, %2,
    0x5410;" : "=r"(regPRMT.w) :
    "r"(regPRMT.w), "r"(tmp));
```

After transposition, we use two kinds of algorithms im2col plus GEMM [1, 2] and implicit GEMM [3] to implement convolution, and choose a faster algorithm for each convolution layer before training. Through these two algorithms, we convert the original convolution into dot product. Then we use one float load instruction to load four INT8 data and one DP4A instruction to compute four INT8 dot product operations. This can speed up the INT8 convolution significantly.

C.2. Stochastic Rounding

Due to the use of stochastic rounding in quantizing gradients, we need to generate uniform random numbers during the backward process. One way to generate random numbers is using $curandGenerator$, but this instruction needs extra global memory access, which will significantly degrade our INT8 convolution performance, with time consumption increasing over 100%. Another method is to use $curand_uniform$, and we need to set a unique $curandState$ for each thread to get different random numbers, which requires a large amount of gpu memory. Worse still, this method runs as slow as the first method. Considering both disadvantages above, we use Linear Congruential Generator (LCG) [5] to yield a sequence of pseudo-randomized numbers instead.

The generator is defined by recurrence relation,

$$X_{n+1} = (aX_n + c) \bmod m, \quad (12)$$

where X is the sequence of pseudo-random values, m is the modules, a is the multiplier, c is the increment, and X_0 is the random seed. The parameters a, c and m are set to constants.

In order to get different random seeds in each thread, we set the random seed X_0 to first input data and add the thread index to X_0. With above settings, each thread can get a unique random seed. The LCG method generates random numbers quickly and brings slight time consumption to INT8 convolution.

C.3. Detailed Speed Result

We test the speed of each convolutional layer in ResNet-50 int8 training on ImageNet dataset, and compare with full

precision training. The speedup of each convolutional layer in ResNet-50 is shown in Figure 6. We find that speedup of different layers is different, and some layers are more difficult to optimize. We will spend more effort to optimize int8 convolutional kernel in the future.

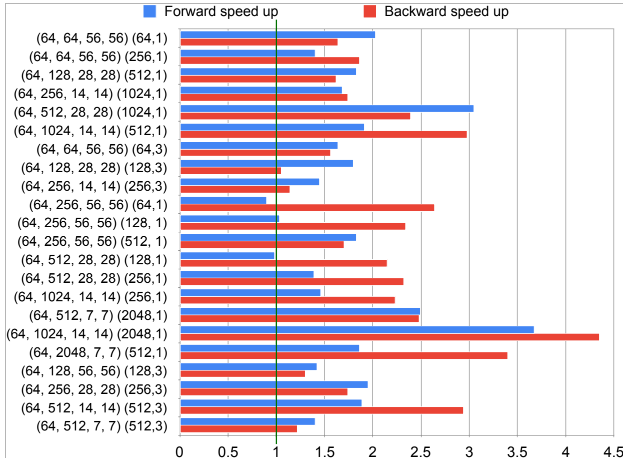


Figure 6. INT8 convolution speedup on GPU, where Y-axis indicates (input shape), (kernel number, kernel size) of convolution.

References

[1] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, New York, NY, USA, 2014. ACM. 3

[2] Patrice Simard Kumar Chellapilla, Sidd Puri. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*, La Baule, France, 2006. 3

[3] NVIDIA Corporation. cuDNN Documentation. `https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html`. 2, 3

[4] NVIDIA Corporation. PTX ISA. `https://docs.nvidia.com/cuda/parallel-thread-execution/`. 2

[5] S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *Commun. ACM*. 3