MIXED PRECISION TRAINING OF CONVOLUTIONAL NEURAL NETWORKS USING INTEGER OPERATIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

The state-of-the-art (SOTA) for mixed precision training is dominated by variants of low precision floating point operations, and in particular FP16 accumulating into FP32 Micikevicius et al. (2017). On the other hand, while a lot of research has also happened in the domain of low and mixed-precision Integer training, these works either present results for non-SOTA networks (for instance only AlexNet for ImageNet-1K), or relatively small datasets (like CIFAR-10). In this work, we train state-of-the-art visual understanding neural networks on ImageNet-1K dataset, with Integer operations on General Purpose (GP) hardware. In particular, we focus on Integer Fused-Multiply-and-Accumulate (FMA) operations which take two pairs of INT16 operands and accumulate results into an INT32 output.We propose a shared exponent representation of tensors, and develop a Dynamic Fixed Point (DFP) scheme suitable for common neural network operations. The nuances of developing an efficient integer convolution kernel is examined, including methods to handle overflow of the INT32 accumulator. We implement CNN training for ResNet-50, GoogLeNet-v1, VGG-16 and AlexNet; and these networks achieve or exceed SOTA accuracy within the same number of iterations as their FP32 counterparts without any change in hyper-parameters and with a 1.8X improvement in end-to-end training throughput. To the best of our knowledge these results represent the first INT16 training results on GP hardware for ImageNet-1K dataset using SOTA CNNs and achieve highest reported accuracy using half precision representation.

1 INTRODUCTION

While single precision floating point (FP32) representation has been the mainstay for deep learning training, half-precision and sub-half-precision arithmetic has recently captured interest of the academic and industrial research community. Primarily this interest stems from the ability to attain potentially upto 2X or more speedup of training as compared to FP32, when using half-precision fused-multiply and accumulate operations. For instance NVIDIA Volta NVIDIA (2017) provides 8X more half-precision Flops as compared to FP32.

Unlike single precision floating point, which is a unanimous choice for *32b* training, half-precision training can either use half-precision floating point (FP16), or integers (INT16). These two options offer varying degrees of precision and range; with INT16 having higher precision but lower dynamic range as compared to FP16. This also leads to residues between half-precision representation and single precision to be fundamentally different – with integer representations contributing lower residual errors for larger (and possibly more important) elements of a tensor. Beyond this first order distinction in data types, there are multiple algorithmic and semantic differences (for example FP16 multiply-and-accumulate operation accumulating into FP32 results) for each of these data types. Hence, when discussing half-precision training, the whole gamut of tensor representation, semantics of multiply-and-accumulate operation, down-conversion scheme (if the accumulation is to a higher precision), scaling and normalization techniques, and overflow management methods must be considered in totality to achieve SOTA accuracy. Indeed, unless the right combination of the aforesaid vectors are selected, half precision training is likely to fail. Conversely, drawing conclusions on the efficacy of a method by not selecting all vectors properly can lead to inaccurate conclusions.

In this work we describe a mixed-precision training setup which uses:

- INT16 tensors with shared tensor-wide exponent, with a potential to extend to sub-tensor wide exponents.
- An instruction which multiplies two INT16 numbers and stores the output into a INT32 accumulator.
- A down-convert scheme based on the maximum value of the output tensor in the current iteration using multiple rounding methods like nearest, stochastic, and biased rounding.
- An overflow management scheme which accumulates partial INT32 results into FP32, along with trading off input precision with length of accumulate chain to gain performance.

The compute for neural network training is dominated by GEMM-like, convolution, or dot-product operations. These are amenable to speedup via specialized low-precision instructions for fused-multiply-and-accumulate (FMA), like QVNNI16 [1]. However, this does not necessarily mean using half-precision representation for all tensors, or using only half-precision operations. In fact performance speedups by migrating the FPROP, BPROP and WTGRAD steps is often close to the maximum achievable speedup obtained by replacing all operations (for instance SGD) in half-precision. In cases where it is not, performance degradation typically happens due to limitations of memory bandwidth, and other architectural reasons. Hence on a balanced general purpose machine, a mixed-precision strategy of keeping precision critical operations (like SGD and some normalizations) in single precision and compute intensive operations in half precision can be employed. The proposed integer-16 based mixed-precision training follows this template.

Using the aforesaid method, we train multiple visual understanding CNNs and achieve Top-1 accuracies Russakovsky et al. (2015)on the ImageNet-1K dataset Deng et al. (2009) which match or exceed single precision results. These results are obtained **without changing any hyper-parameters, and in as many iterations as the baseline FP32 training**. We achieve 75.77% Top-1 accuracy for ResNet-50 which, to the best of our knowledge, significantly exceeds any result published for half-precision training, for example Micikevicius et al. (2017); Ginsburg et al. (2017). Further, we also demonstrate our methodology achieves state-of-the-art accuracy (comparable to FP32 baseline) with int16 training on GoogLeNet-v1, VGG-16 and AlexNet networks. To the best of our knowledge, these are first such results using int16 training.

The rest of the paper is organized as follows: Section 2 discusses the literature pertaining to various aspects of half-precision training. The dynamic fixed point format for representing half-precision tensors is described in Section 3. Dynamic fixed point kernels and neural network training operations are described in Section 4, and experimental results are presented in Section 5. Finally, we conclude this work in Section 6.

2 Related Work

Using reduced precision for Deep learning has been an active topic of research. As a result there are a number of different reduced precision data representations, the more standard floating-point based Micikevicius et al. (2017); Ginsburg et al. (2017); Dettmers (2015) and custom fixed point schemes Vanhoucke et al. (2011); Courbariaux et al. (2014); Gupta et al. (2015); Hubara et al. (2016b); Bansal et al. (2017).

The recently published mixed precision training work from Micikevicius et al. (2017) uses 16-bit floating point storage for activations, weights and gradients. The forward, back propagation computation uses FP16 computation with results accumulating into FP32 and a master-copy of the full precision (FP32) weights are retained for the update operation. They demonstrate a broad variety of deep learning training applications involving deep networks and larger data-sets (ILSVRC-class problems) with minimal loss compared to baseline FP32 results. Further, this shows that FP16 mixed precision requires loss scaling Ginsburg et al. (2017) to achieve near-SOTA accuracy. This ensures back-propagated gradient values are shifted into FP16 representable range and the small magnitude (negative exponent) values, which are critical for accuracy are captured. Such scaling is inherent with fixed point representations, making it more amenable and accurate for deep learning training.

Custom fixed point representations offer more flexibility - in terms of both increased precision and dynamic range. This allows for better mapping of the representation to the underlying application,

[1] https://www.anandtech.com/show/11741/hot-chips-intel-knights-mill-live-blog-445pm-pt-1145pm-utc

thus making it more robust and accurate than floating-point based schemes. Vanhoucke et al. (2011) have shown that the dynamically scaled fixed point representation proposed by Williamson (1991) can be very effective for convolution neural networks - demonstrating upto to $4\times$ improvement over an aggressively tuned floating point implementation on general purpose CPU hardware. Gupta et al. (2015) have done a comprehensive study on the effect of low precision fixed point computation for deep learning and have successfully trained smaller networks using 16-bit fixed point on specialized hardware. With further reduced bit-widths, such fixed point data representations are more attractive - offering increased capacity for precision with larger mantissa bits and dynamically scaled shared exponents. There have been several publications with $<$16-bit precision and almost all of them use such custom fixed point schemes. Courbariaux et al. (2014) use a dynamical fixed point format (DFXP), with low precision multiplications with upto 12-bit operations. Building on this Courbariaux et al. (2015) proposed training with only binary weights while all other tensors and operations are in full precision. Hubara et al. (2016a) further extended this to use binary activations as well, but with gradients and weights still retained in full precision. Hubara et al. (2016b) proposed training with activations and weights quantized upto 6-bits and gradients in full precision. Rastegari et al. (2016) use binary representation for all components including gradients. However, all of the aforementioned use smaller benchmark model/data-sets and results in a non-trivial drop in accuracy with larger ImageNet data-set Deng et al. (2009) and classification task Russakovsky et al. (2015). Bansal et al. (2017) have shown that a fixed point numerical format designed for deep neural networks (Flexpoint), out-performs FP16 and achieves numerical parity with FP32 across a wide set of applications. However, this is designed specifically for specialized hardware and the published results are with software emulation. Here we propose a more general dynamic fixed point representation and associated compute primitives, which can leverage general purpose hardware using the integer-compute pipeline. Further we provide actual accuracy and performance for training large networks for the ILSVRC classification task, measured on available hardware.

3 THE DYNAMIC FIXED POINT FORMAT

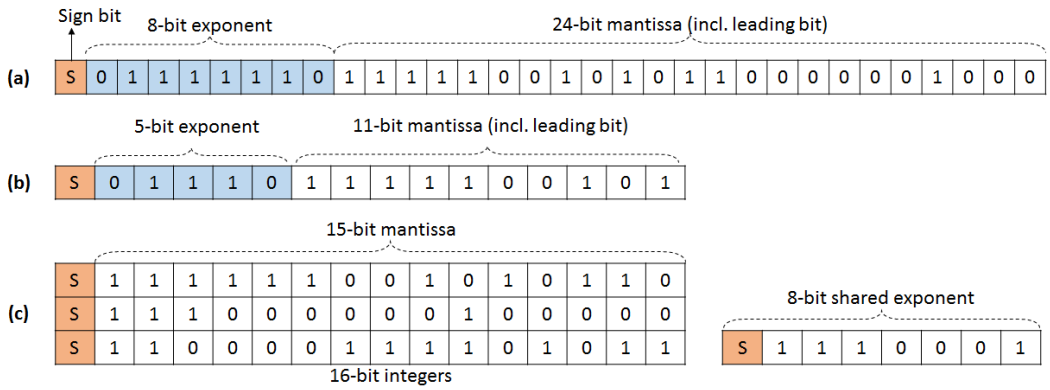

Figure 1: Snapshot of precision and dynamic range capabilities of a) IEEE-754 float b) IEEE-754 half-float, and c) Dynamic Fixed Point (DFP-16) data formats.

Dynamic Fixed Point (DFP) tensors are represented by a combination of an integer tensor I and an exponent E_s, shared across all the integer elements. For the sake of convenience, the DFP tensor can be denoted as DFP-P = $\langle I, E_s \rangle$, where P represents the number of bits used by the integer elements in I (ex: DFP-16 contains 16-bit integers).

Figure 1 illustrates the differences in data representation between IEEE-754 standard format float, half-float and DFP-16 data format. DFP-16 data type offers a trade-off between float and half-float in terms of precision and dynamic range. When compared to full-precision floats, DFP-16 can achieve higher compute density and can carry higher effective precision compared to half-floats because of larger 15-bit mantissa (compared to 11-bits for half-floats). Further, the effective dynamic range of DFP format can be increased by extending the data type to use Blocked-DFP representation. Blocked-DFP uses fine-grained quantization to assign multiple exponents per tensor with smaller blocks of

integers sharing a common exponent. Mellempudi et al. (2017) have demonstrated effectiveness of fine-grained quantization for low-precision inference tasks.

In this work, we use a single shared exponent for each tensor. The integers are stored in 2's compliment representation and the shared exponent is an 8-bit signed integer. We use standard commodity integer hardware to perform arithmetic operations on DFP tensors. This implies that the exponent handling and precision management of DFP is done in the software, which is covered in more detail in Section 4.3.

3.1 DFP Tensor Primitives

To facilitate end-to-end mixed-precision training using DFP, we have created primitives to perform arithmetic operations on DFP tensors and data conversions between DFP and float. When converting floating point tensors into to DFP data type, the shared exponent is derived from the exponent of absolute maximum value of the floating point tensor. If F is the floating point tensor, the exponent of the absolute maximum value is expressed as follows.

$$E_{fmax} = E(\max_{\forall f \in F} |f|) \tag{1}$$

The value of the shared exponent E_s is a function of E_{fmax} and the number of bits P used by the output integer tensor I.

$$E_s = E_{fmax} - (P - 2) \tag{2}$$

The relationship of the resulting DFP tensor $\langle I, E_s \rangle$ with the input floating point tensor F is expressed by Eq.3.

$$\forall i_n \in I, f_n = i_n \times 2^{E_s}, where f_n \in F \tag{3}$$

Extending this basic formulation Eq.3, we can define a set of common DFP primitives required for neural network training.

- Multiplying two DFP-16 tensors produces 32-bit I tensor with a new shared exponent expressed as follows.

$$i_{ab} = i_a \times i_b \textit{ and exponent, } E_s^{ab} = E_s^a + E_s^b \tag{4}$$

- Adding two DFP-16 tensors may result in a 32-bit I tensor and a new shared exponent.

$$i_{a+b} = \begin{cases} i_a + (i_b >> (E_s^a - E_s^b)), \textit{ when } E_s^a > E_s^b \\ i_b + (i_a >> (E_s^b - E_s^a)), \textit{ when } E_s^b > E_s^a \end{cases} \quad \text{and exponent, } E_s^{a+b} = \max_{E_s^a, E_s^b} \tag{5}$$

  Note that when a Fused Multiply and Add operation is performed, all products have the same shared exponent: $E_s^{ab} = E_s^a + E_s^b$, and hence the sum of such products also has the same shared exponent.

- Down-Conversion scales DFP-32 output of a layer to DFP-16 to be passed as input to the next layer. The 32-bit I tensor right-shifted R_s bits to fit into 16-bit tensor. The R_s value and the new shared exponent are expressed as follows.

$$\begin{gathered} R_s = P - LZC(\max_{\forall i_{ab} \in I^{32}} |i_{ab}|) \\ i_{ab}^d = i_{ab} >> R_s \textit{ and exponent, } E_s^{ab} += R_s \end{gathered} \tag{6}$$

4 Neural Network Training using Dynamic Fixed Point

Neural network training is an iterative process over minibatches of data points, with four main operations on a given minibatch: forward propagation, backpropagation, weight gradient computation, and the solver (typically stochastic gradient descent, or ADAM).

In a CNN, the three steps of forward-propagation, backpropagation, and weight-gradient computation are often the compute intensive steps, and consist of GEMM-like (General Matrix Multiply) convolution operations which dominate the compute, and additional element-wise operations like normalization, non-linear (ReLU) and element-wise addition. In this work we propse a method to use INT16 operations, for implementing kernels for the convolutions and GEMM. There kernels are

stitched with the rest of the operations in neural network training via *Dynamic Fixed Point* to floating point conersions described earlier in Section 3. In this section, we first describe the overall method for using dynamic fixed point in neural network training, and then explain the optimized kernel for convolutions.

4.1 TRAINING WITH DYNAMIC FIXED POINT

The mixed precision training scheme used in this work is described in Figure 2. The core compute kernels in this scheme are the *FPROP, BPROP, and WTGRAD* functions which take two DFP-16 tensors as input and produces a FP32 tensor as output. For example *FPROP* takes two DFP-16 tensors, **Act[l], and Wt[l]** (activations and weights for layer-*l*), and produces a FP32 output tensor **Act_f32[l+1]**. The FPROP and BPROP operations are followed by quantization steps (*QuantizeFP32_DFP16*) which convert the FP32 tensors (Act_f32[l+1], delAct_f32[l]) to DFP-16 tensors (Act[l+1], delAct[l]) for operations in the next layer. The WTGRAD step is followed by the Stochastic Gradient Descent (SGD) step, which takes the FP32 tensor for weight-gradients (delWt_f32[l]) and a FP32 copy of the weights (Wt_f32[l]) as inputs, and produces an updated weight tensor as output. We follow the now established practice Micikevicius et al. (2017) of keeping a FP32 copy of weights as well as a low precision (DFP-16) copy of weights. Therefore SGD or other solvers are FP32 operations. In case a batch-norm layer is used, it operates on the intermediate FP32 result (for FPROP and BPROP), prior to the quantization step.

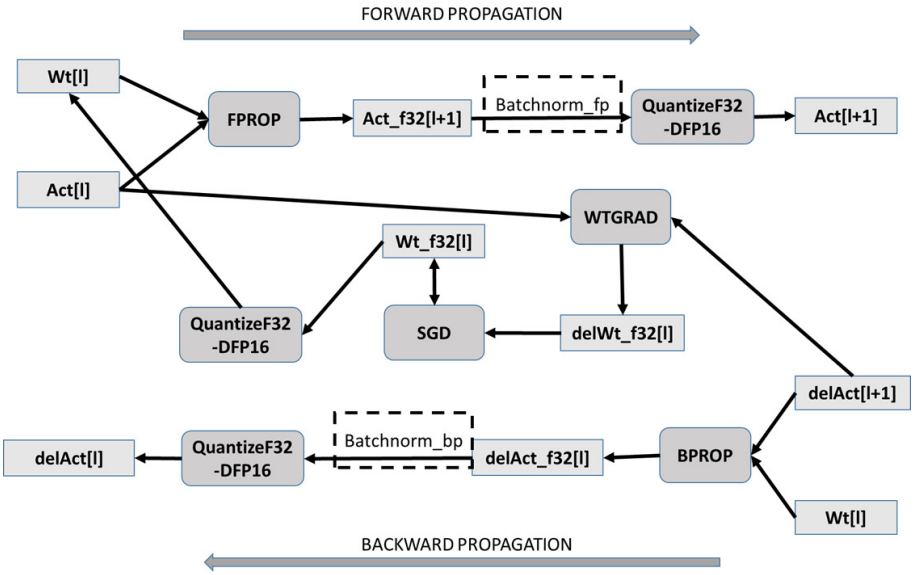

Figure 2: Overview of Dynamic Fixed Point Training Methodology

4.2 CORE COMPUTE KERNELS

In this section we delve into efficient implementations of core compute kernels written using Integer FMA instruction sequence; in particular the QVNNI16 instruction (described in Algorithm 1). This instruction takes a memory pointer as the first input and four vector registers as the second input and performs 8 multiply-add operations per output (16 Integer-OPs). For each 32b lane, the instruction takes two pairs of 16-bit Integers, performs a multiply followed by a horizontal add.

The *FPROP* convolution kernel is written using QVNNI-16 instruction in Algorithm 2. The data layout of the weights captures the 2-way horizontal accumulation operation in QVNNI16. Here the last dimension moves along consecutive input-feature maps. Hence the dimensions of activations is: N, C/16, H, W, 16, and that of weights is C/16, K/16, KH, KW, 8c, 16k, 2c (where C and K are input and output feature maps, H, W are input feature map height and width, and KH, KW are kernel height/width). Note that while we briefly touch upon data layout and blocking of the core kernel

Algorithm 1 Semantics of the QVNNI16 Instruction

```
K=4; SIMD_WIDTH=16;
QVNNI16(short *mem, _m512i vinp2[0..3], _m512i vout)
for v = 0 . . . K-1 do
    for o = 0 . . . SIMD_WIDTH-1 do
        vout[o] += vinp2[v][2*o]*mem[2*v] + vinp2[v][2*o+1]*mem[2*v+1]
    end for
end for
```

loops in Algorithm 2, detailed analysis of performance is not the objective of this work. These details are explored only to highlight different functional components of the kernel.

Algorithm 2 Example Forward Propagation Loop

```
1: fprop(DFP16 <input[IC/16][IH][IW][16], e_inp>, DFP16 <weights[IC/16][OC/16][KH][KW][8][16][2],
   e_wt>; FP32 output[OC/16][OH][OW][16] = 0)
2: _m512 vwt[0. . . 3], vout[RB_SIZE], vtemp, vscale;
3: vscale = VBROADCAST(2^(e_inp+e_wt))
4: for ofm = 0 . . . OC/16-1 do
5:     for ofh = 0 . . . OH-1 do
6:         for ofw = 0 . . . OW/RB_SIZE-1 do
7:             for ifm = 0 ... IC/ICBLK-1 do
8:                 for rb=0 . . . RB_SIZE-1 do
9:                     vout[rb] = SETZERO()
10:                end for
11:                for ifmb = 0 ... ICBLK/16-1 do
12:                    for kh = 0 . . . KH-1 do
13:                        for kw = 0 . . . KW-1 do
14:                            for ib = 0. . . 1 do
15:                                for v= 0. . . 3 do
16:                                    vwt[v] = LOAD(&weights[ifm][ofm][kh][kw][ib*4+v][0][0])
17:                                end for
18:                                for rb = 0 . . . RB_SIZE-1 do
19:                                    QVNNI16(&input[ifm*(IC/ICBLK)+icb][S*ofh+kh][S*ofw+kw][ib]),
                                       vwt[0. . . 3], vout[rb])
20:                                end for
21:                            end for
22:                        end for
23:                    end for
24:                end for
25:            end for
26:            for rb=0 . . . RB-1 do
27:                vtemp = LOAD(&output[ofm][ofmh][ofmw*RB_SIZE + rb][0])
28:                vout[rb] = VCVTINTFP32(vout[rb])
29:                vtemp = VFP32MADD(vtemp, vout[rb], vscale) //vtemp = vtemp + vout[rb]*vscale
30:                STORE(vtemp, &output[ofm][ofmh][ofmw*RB_SIZE + rb][0])
31:            end for
32:        end for
33:    end for
34: end for
```

4.3 Handling Overflows in INT16-INT32 FMAs

Multiplication of two INT16 numbers can result in a 30-bit outcome, and hence an accumulate chain of 3 products of INT16 multiplicative pairs can cause an overflow of the INT32 accumulator. In neural network training, accumulate chains can exceed a million in length (for example in the WTGRAD kernel).

One way to prevent overflows is to convert an INT32 intermediate output into FP32 before accumulation as described in lines 26-31 in Algorithm 2. Here we first convert the INT32 result to FP32 using the VCVTINTFP32 instruction, followed by a scale and accumulate into the final FP32 result using

the VFP32MADD instruction. The scale used is $2^{(E_{inp}+E_{wt})}$ (equation 3), which is broadcast and stored in the *vscale* vector register. The instruction sequence in lines 26-31 can be applied after every QVNNI instruction to prevent almost all overflows. However the overheads would be significant and hurt performance. Hence we pick the strategy of partial accumulations into INT32 for short accumulate chains, and subsequently converting the results into FP32.

Performance Impact: The difference between an ideal instruction sequence (with no overflow management) and Algorithm 2 is essentially the additional VCVTINTFP32 instruction (line 28). In the loop in lines 8-31, we have *(ICBLK/16)*KH*KW*2*RB* QVNNI instructions, and *RB*4 + (ICBLK/16)*KH*KW*4* non-QVNNI instructions, and *RB* VCVTINTFP32 instructions. The instruction overhead from overflow management therefore varies between $<1\%$ in most cases, to at most 3%.

The length of the accumulate chain (via sizing the input feature map blocking factor ICBLK in line 7) is selected to optimize instruction overheads and cache/instruction reuse. In this work we strive to keep the accumulate chain to more than 200 (which is empirically shown to be close to optimal). Often this accumulate chain also overflows, which we circumvent by shifting inputs. In this work, we shift both the inputs by 1-bit for all convolutions in all experiments. Hence effectively we have a DFP15 representation of all DFP tensors. It is notable that this shift value is largely dependent on this inner accumulate chain length, and by constraining it we can find a shift value applicable across all operations. The combination of input shift and conversion of outputs to FP32 allows us to prevent occurrence of any overflows and hence catastrophic errors during training.

5 Experiments and Results

We compare mixed precision DFP16 training with baseline full precision (FP32) for several ImageNet-class SOTA CNNs. Both baseline and DFP16 experiments are run using versions of the BVLC CAFFE framework Jia et al. (2014). For the baseline runs we use Intel's CAFFE branch[2]. For the mixed precision DFP16 experiments we use a private fork of this branch, where we have added DFP16 data-type support. The DFP16 compute primitives are supported through the prototype 16-bit integer kernels in Intel's MKL-DNN library[3] along with explicit exponent management as described in Section.4. Both baseline and mixed precision DFP16 experiments are run on the newly introduced Intel Knights-Mill hardware using upto 32 nodes for training. Overall we see an average 1.8X speedup in the training throughput compared to the the baseline FP32 performance on the same platform. For instance, the FP32 baseline run of ResNet-50 in Intel Caffe on KNM with the latest MKL-DNN library achieves a throughput of 153 images/s. With our DFP16 implementation on KNM achieves a throughput of 275 images/s for the same ResNet-50 topology.

5.1 Accuracy Results for CNNs

We trained several CNNs for the ImageNet-1K classification task using mixed precision DFP16: AlexNet Krizhevsky et al. (2012), VGG-16 Simonyan & Zisserman (2014), GoogLeNet-v1 Szegedy et al. (2015), ResNet-50 He et al. (2016). We use exactly the same batch-size and hyper-parameter configuration for both the baseline FP32 and DFP16 training runs (Table.1). In both cases, the models are trained from scratch using synchronous SGD on multiple nodes. In our experiments the first convolution layer (C1) and the fully connected layers are in FP32 (constituting about $5-10\%$ of compute for modern CNNs). Table.1 shows ImageNet-1K classification accuracies, training with DFP16 achieve SOTA accuracy for all four models and in several cases even better than the baseline full precision result.

To the best of our knowledge, *top-1 accuracy of **75.77%** and top-5 accuracy of **92.84%** for ResNet-50 with mixed precision DFP16 - is highest achieved accuracy on the ImageNet-1K classification task with any form of reduced precision training*.

It can be seen from Figure.3 that DFP16, closely tracks the full precision training. For some models like GoogLeNet-v1 and AlexNet, we observe the initially DFP16 training lags the baseline, however this gaps is closed with subsequent epochs especially after the learning rate changes. Further, we

[2]https://github.com/intel/caffe

[3]https://01.org/mkl-dnn

Table 1: Training configuration and ImageNet-1K classification accuracy

Models	Batch-size / Epochs	Baseline		Mixed precision DFP16	
		Top-1	Top-5	Top-1	Top-5
Resnet-50	1024 / 90	75.70%	92.78%	**75.77%**	**92.84%**
GoogLeNet-v1	1024/ 80	69.26%	89.31%	**69.34%**	**89.31%**
VGG-16	256 / 60	**68.23%**	**88.47%**	68.12%	88.18%
AlexNet	1024 / 88	**57.43%**	**80.65%**	56.94%	80.06%

observe that compared to baseline run - with DFP16 the validation/test loss tracks much closer to the training loss. We believe this is the effect of the additional noise introduced from reduced precision computation/storage, which is results in better generalization with reduced training-testing gap and better accuracies.

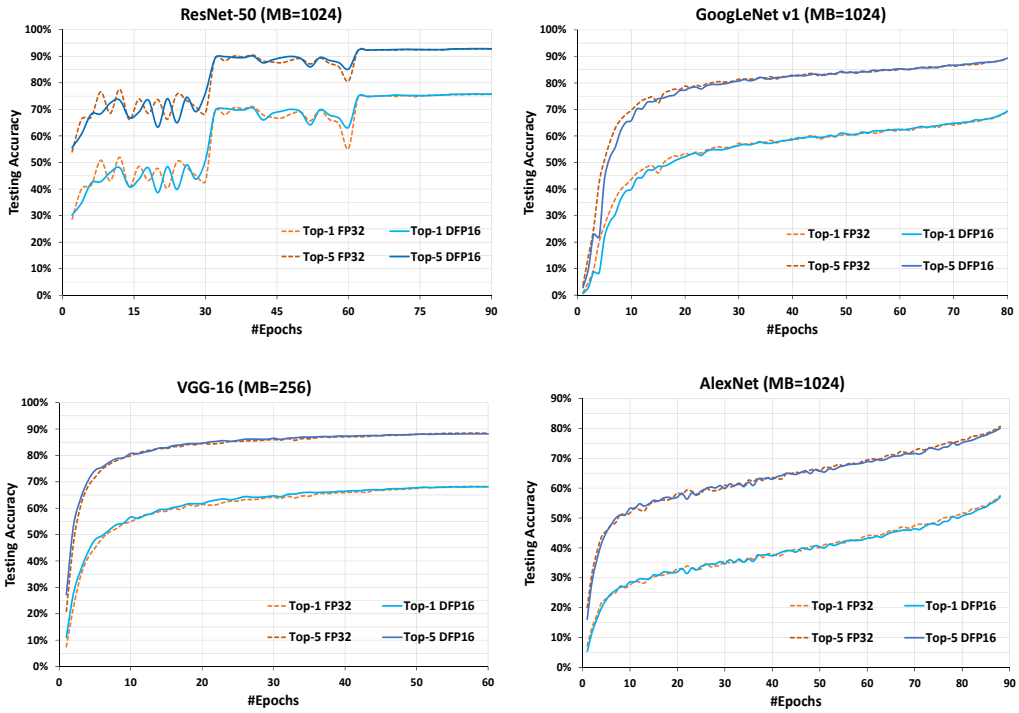

Figure 3: Convergence plots for DFP-16b training vs. reference baseline FP32 results for ResNet-50, GoogLeNet-v1, VGG-16 and AlexNet trained for ImageNet-1K classification task

6 Conclusions

We demonstrate industry-first reduced precision INT-based training result on large networks/data-sets. Showing on-par or better than FP32 baseline accuracies and potentially 2X savings in computation, communication and storage. Further, we propose a general dynamic fixed point representation scheme, with associated compute primitives and algorithm for the shared exponent management. This DFP solution can be used with general purpose hardware, leveraging the integer compute pipeline. We demonstrate this with implementation of CNN training for ResNet-50, GoogLeNet-v1, VGG-16 and AlexNet; training these networks with mixed precision DFP16 for the ImageNet-1K classification task. While this work focuses on visual understanding CNNs, in future we plan to demonstrate the efficacy of this method for other types of networks like RNNs, LSTM, GANs and extend this to wider set of applications.

REFERENCES

Arjun K Bansal, William Constable, Oguz Elibol, Stewart Hall, Luke Hornoff, Amir Khosrowshahi, Carey Kloss, Urs Koster, Marcel Nassar, Naveen Rao, Xin Wang, and Tristan Webb. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. *The Thirty-first Annual Conference on Neural Information Processing*, 2017.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*, pp. 3123–3131, 2015.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pp. 248–255. IEEE, 2009.

Tim Dettmers. 8-bit approximations for parallelism in deep learning. *arXiv preprint arXiv:1511.04561*, 2015.

Boris Ginsburg, Sergei Nikolaev, and Paulius Micikevicius. Training of deep networks with half-precision float. *NVidia GPU Technology Conference*, 2017.

Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *ICML*, pp. 1737–1746, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in neural information processing systems*, pp. 4107–4115, 2016a.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *arXiv preprint arXiv:1609.07061*, 2016b.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

Naveen Mellempudi, Abhisek Kundu, Dheevatsa Mudigere, Dipankar Das, Bharat Kaul, and Pradeep Dubey. Ternary neural networks with fine-grained quantization. *arXiv preprint arXiv:1705.01462*, 2017.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

NVIDIA. Nvidia tesla v100 gpu architecture - whitepaper. 2017. URL `http://www.nvidia.com/object/volta-architecture-whitepaper.html`.

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on cpus. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, volume 1, pp. 4. Citeseer, 2011.

Darrell Williamson. Dynamically scaled fixed point arithmetic. In *Communications, Computers and Signal Processing, 1991., IEEE Pacific Rim Conference on*, pp. 315–318. IEEE, 1991.