QPyTorch: A Low-Precision Arithmetic Simulation Framework

Tianyi Zhang, Zhiqiu Lin, Guandao Yang, Christopher De Sa
Cornell University, Department of Computer Science
{tz58,zl279,gy46,cmd353}@cornell.edu

Abstract

Low-precision training reduces computational cost and produces efficient models. Recent research in developing new low-precision training algorithms often relies on simulation to empirically evaluate the statistical effects of quantization while avoiding the substantial overhead of building specific hardware. To support this empirical research, we introduce QPyTorch, a low-precision arithmetic simulation framework. Built natively in PyTorch, QPyTorch provides a convenient interface that minimizes the efforts needed to reliably convert existing codes to study low-precision training. QPyTorch is general, and supports a variety of combinations of precisions, number formats, and rounding options. Additionally, it leverages an efficient fused-kernel approach to reduce simulator overhead, which enables simulation of large-scale, realistic problems. QPyTorch is publicly available at `https://github.com/Tiiiger/QPyTorch`.

1 Introduction

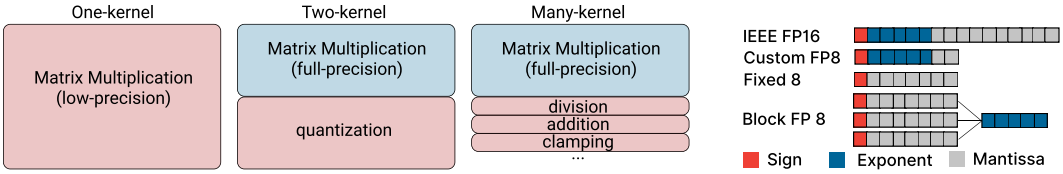
Low-precision arithmetic is one of the most successful techniques in compressing and accelerating Deep Neural Networks (DNNs). To obtain high-performance low-precision models, many works study low-precision training algorithms [7, 3, 15]. Along with the advent of low-precision training techniques, there has been a growing interest in software that simulates low-precision computation in order to understand its statistical effects on training. This simulation circumvents the overhead of building hardware while enabling investigations into the numerical behaviors of different designs.

To facilitate empirical research on low-precision training, we introduce QPyTorch, a low-precision arithmetic simulation framework. In designing QPyTorch, we have the following objectives. First, we aim to support broad and general research into low-precision numeric formats. Popular machine learning frameworks like Tensorflow [2] now support simulating fixed-point computation, in addition to the 16-bit low-precision floating point numbers that are often supported directly on hardware. However, recent algorithms often propose low-precision number formats that are more general than these options [12, 14], including block floating point and non-standard-width floating point numbers. To better support this sort of research, we support a diverse range of numeric design choices with our framework, including allowing arbitrary mix-and-match of different precisions, number formats, and rounding options.

Second, we value efficiency on commodity hardware. The current state of low-precision training research often requires validations on large-scale, realistic problems (e.g. ImageNet [13]). Efficient simulation enables fast iterations in empirical research. Unfortunately there is inherent trade-off between speed and generality—in some cases, we eschew support for a low-precision operation that would greatly diminish simulator efficiency. In this report, we document the scope of the features QPyTorch supports and evaluate our design choices regarding efficiency.

Third, we develop a convenient front-end interface. Low-precision simulation usually "hijacks" a regular computation graph in deep learning training, and quantization code can easily be buried

Preprint. Under review.



(a) Illustration of QPyTorch's two-kernel approach.

(b) Example supported formats

Figure 1

within a complex machine learning codebase. QPyTorch provides proper abstractions to elucidate the high-level designs in low-precision training, which helps understanding and reproducibility. Additionally, our interface allows users to adjust the settings for individual layer or number categories independently. QPyTorch also implements a common set of low-precision training techniques from the literature[1] [3, 15, 14, 16], which can serve as a set of building blocks for future exploration.

Finally, we focus on the reliability of QPyTorch. Naive implementation of low-precision simulation can be error-prone. For example, high-precision numbers can be accidentally leaked into the training process. In the past, we have found quantization bugs in low-precision codebases that could potentially invalidate some conclusions in the published literature. It is our hope that QPyTorch can help prevent this class of error, both because QPyTorch contains standard test suites to validate the implementation, and because potential errors may be exposed more quickly by a growing open-source community working on a single codebase.

2 Related Works

Low-level low-precision packages. Tensorflow-Lite [8] supports dynamically quantizing a trained model into 8 bits at inference time and store computation results in floating point. QNNPACK [11] provides a mobile-optimized 8-bit fixed point implementation. FBGEMM [4] provides low-level matrix-multiplication and convolution support for half precision floating point and 8-bit fixed point.

Low-precision simulation frameworks. TensorQuant [10] is one of the first software frameworks that supports fixed point computation simulation. Tensorflow [2] supports simulating fixed point computation of arbitrary bits with nearest rounding. In comparison, our framework supports a wider range of number formats and rounding options. QPyTorch also allows the users to configure individual layers and number categories independently.

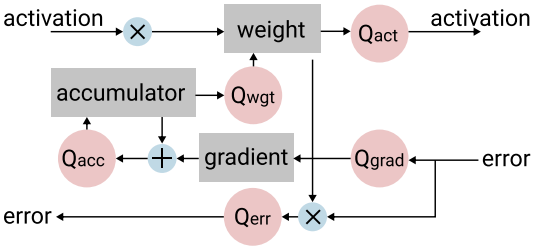
3 Design

At its core, QPyTorch operates by (1) representing low-precision numbers as their corresponding floating-point number, and (2) simulating low-precision computation by running single-precision floating-point computation and then removing the extra precision through quantization. For individual base operations (e.g. scalar $+$, $\times$), this approach is equivalent to low-precision computation. However, for composite operations (e.g. matrix multiplication), this corresponds to using a high-precision accumulator rather than accumulating in the same low-precision format: this choice is standard in the literature [15, 16]. For QPyTorch, we are motivated by efficiency: quantizing after each base operation would significantly slow down the simulator (see discussion in Loroch et al. [10]).

3.1 Features of QPyTorch

Number Formats. QPyTorch can simulate a diverse set of low-precision number formats (Fig 1b), including floating point, fixed point, and block floating point numbers. For floating point numbers, QPyTorch can simulate any floating point format that uses fewer bits than single precision (i.e. those that use fewer than 8 bits for exponent and fewer than 23 bits for mantissa). Among these are the Brain Floating Point (bfloat16) and the effective 8-bit and 16-bit format proposed in Wang et al. [14]. In the interest of efficiency, we do not simulate denormals, NaN, and Inf, as these numbers are expected to occur rarely in training and are often not all supported in low-precision hardware [1].

[1]For instance, many training algorithms keep a higher-precision model copy for gradient accumulation.



(a) Categories of numbers in low-precision training

```
prec = BlockFloatingPoint(wl=8)
accumulate_prec = BlockFloatingPoint(wl=16)

model = net()
model = lower(model, layer_types=["conv", "linear"],
              forward_number=prec, backward_number=prec,
              forward_rounding="nearest",
              backward_rounding="stochastic")

optim = SGD(model.parameters(), lr=0.01)
optimizer = OptimLP(optimizer, weight_quant=prec,
                    grad_quant=prec,
                    acc_quant=accumulate_prec)
```

(b) Example QPytorch implementation

Figure 2

QPyTorch also supports fixed point, which is a popular format that has been used in many algorithms [6, 15]. QPyTorch can simulate fixed point numbers with arbitrary precision although the precision is again limited by the underlying single precision computation (restricting us to fixed-point numbers with at most 24 bits of precision). Block floating point numbers reduce the bit usage of floating point numbers by sharing the exponent bits across a block [16]. QPyTorch can simulate any block floating point format that allocates fewer than 8 bits for exponent and fewer than 23 bits for mantissa. The assignment of numbers to blocks appears to be an important design decision in low-precision training algorithms that use block floating point [17, 16]. QPyTorch supports assigning the block over an arbitrary dimension of a tensor or treating the tensor as an single block. Finally, QPyTorch provides a unified API to combine these primitive quantizations with additional PyTorch code to define more specialized formats [15, 17].

Rounding mode. Stochastic rounding has been shown to be beneficial for neural network training [6]. QPyTorch implements both stochastic and nearest-neighbor rounding. For handling the middle-points with nearest-neighbor rounding, QPyTorch supports round-away-from-zero, round-toward-zero, and round-to-nearest-even.

Front-end interface. Neural network training involves distinctive groups of numbers [5]. Figure 2a illustrates the different roles of weight, accumulator, gradient, activation, and error in a single layer. Different categories and different layers in a neural network may require distinct precision and quantization [6]. QPyTorch provides a convenient interface to handle different number categories independently, allowing the search for more effective combinations of different precision, formats, and rounding mode. Specifically, QPyTorch injects the quantization of activation and error as a separate PyTorch module into each layer and abstracts the quantization of weight, gradient, and accumulators into a low-precision optimizer. In addition, we provide a useful tool to automatically inject the quantization modules so that low-precision networks do not require a separate model definition. Using these convenient designs, it only takes around 10 lines of code to convert an existing full-precision training codebase into a low-precision one (Figure 2b).

3.2 Two-Kernel Approach

One can imagine several potential designs for QPyTorch. The most straightforward is to utilize existing PyTorch operations directly. The caveat is that each PyTorch operation would launch a separate CUDA kernel and many separate launches are inefficient. Moreover, due to the lack of support for bitwise operations in PyTorch, this approach cannot simulate low-precision floating point numbers. From now on, we refer to this strategy as the *many-kernel approach*. On the other hand, we could implement a customized kernel for each common operation, such as matrix multiplication, convolution, nonlinearities, etc. This *one-kernel design* is a popular choice when efficiency is of major concern, e.g., in software packages designed to leverage low-precision hardware [11, 4]. Although fast, this approach is cumbersome for general research purposes: as any new operations arising in frontier research would require a separate kernel, this adds significant maintenance overhead. QPyTorch adopts an intermediate approach. We fuse instructions specific to quantization into a single kernel and then append it to a regular full-precision operation run in a separate kernel. Figure 1a illustrates this *two-kernel approach*.

Algorithm	PyTorch	QPyTorch
FP16 SGD	6.88	6.89
Block8 SGD	8.24	8.4
WAGE	6.96	6.95
SWALP	6.7	6.67

Table 1: Error rate of four models trained with PyTorch and QPyTorch on CIFAR10.

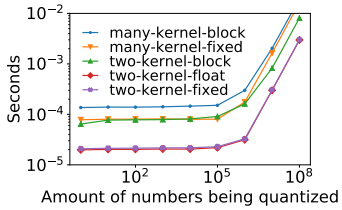


Figure 3: Wall Clock Time of Quantization Kernels.

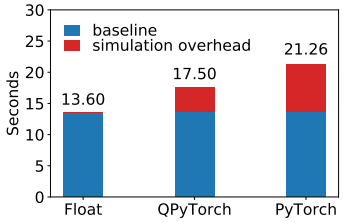


Figure 4: Wall Clock Time of training WAGE for one Epoch.

4 Validation

To validate the correctness of our framework, we train various low-precision models. We use the CIFAR10 [9] dataset to benchmark the system performance. We first study applying stochastic gradient descent to train VGG16 models with two precision settings, half-precision floating point and 8-bit block floating point where each tensor is treated as a block. We apply standard data augmentation and borrow other training hyperparameters from Yang et al. [16]. Then, we implement two state-of-the-art low-precision training algorithms [16, 15] in QPyTorch and compare to the reported performance. For SWALP [16], we train a VGG16 model using the reported 8-bit small-block block floating point format. For WAGE [15], we implement the customized VGG model and the specialized number format, which can be viewed as a variant of fixed point. Table 1 shows that QPyTorch performs as expected.

We also evaluate QPyTorch's two-kernel approach. We cannot fairly compare to a one-kernel approach by modifying existing optimized full-precision kernels because source code for these kernels is not publicly available: therefore, we restrict our evaluation to a comparison between the two-kernel and the many-kernel approaches. We implemented the many-kernel approach directly through PyTorch operations, and compare it with our CUDA implementation of the two-kernel approach used in QPyTorch. Since PyTorch does not support bitwise shift operations, a many-kernel implementation of floating point quantization is not possible, so we restrict our comparison to the fixed point and block floating point formats. We denote our PyTorch implementation of fixed point quantization `many-fixed` and block floating point quantization `many-block`. Similarly, denote the fused kernels of QPyTorch `two-kernel-fixed`, `two-kernel-block`. We refer to the floating point quantization in QPyTorch as `two-kernel-float`. In Figure 3, we plot the wall clock time spent quantizing tensors of different sizes. Figure 3 demonstrates the efficiency of our two-kernel approach. Notably, `two-kernel-fixed` with nearest rounding on GPU is about 5x faster than `many-kernel-fixed`. In addition, for Block Floating Point which is most expensive to simulate, `two-kernel-block` is about 2.6x faster than `many-kernel-block`. We observe similar results for stochastic rounding and quantizing on CPUs. Additionally, we compare the run time of training WAGE for one epoch. Figure 4 shows that QPyTorch effectively halves the end-to-end simulation overhead (3.9s vs. 7.56s).

5 Conclusion and Future Work

In this report we introduce QPyTorch, a low-precision arithmetic simulation framework. QPyTorch targets low-precision training research, facilitating studies on various number formats, rounding choices. QPyTorch adopts a two-kernel approach for efficient simulation and offers a convenient interface that is tailored for recent algorithms. We validate QPyTorch by training low-precision neural networks in four diverse settings and demonstrate its efficency empirically.

References

[1] Bfloat16 – hardware numerics definition. URL https://software.intel.com/en-us/download/bfloat16-hardware-numerics-definition.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zhang. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.

[3] 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*, pages 3123–3131, 2015.

[4] Jianyu Huang James Reed Daya Khudia, Jongsoo Park. Fbgemm. 2018. URL https://github.com/dskhudia/FBGEMM.

[5] Christopher De Sa, Matthew Feldman, Christopher Ré, and Kunle Olukotun. Understanding and optimizing asynchronous low-precision stochastic gradient descent. *SIGARCH Comput. Archit. News*, pages 561–574, 2017. ISSN 0163-5964.

[6] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746, 2015.

[7] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 4107–4115. 2016.

[8] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *CoRR*, abs/1712.05877, 2017.

[9] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[10] Dominik Marek Loroch, Norbert Wehn, Franz-Josef Pfreundt, and Janis Keuper. Tensorquant - a simulation toolbox for deep neural network quantization. In *MLHPC@SC*, 2017.

[11] Hao Lu Marat Dukhan, Yiming Wu and Bert Maher. Qnnpack. 2018. URL https://github.com/pytorch/QNNPACK.

[12] Naveen Mellempudi, Abhisek Kundu, Dipankar Das, Dheevatsa Mudigere, and Bharat Kaul. Mixed low-precision deep learning inference using dynamic fixed point. *arXiv preprint arXiv:1701.08978*, 2017.

[13] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Fei-Fei Li. Imagenet large scale visual recognition challenge. *arXiv*, abs/1409.0575, 2014.

[14] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 7685–7694. Curran Associates, Inc., 2018.

[15] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and inference with integers in deep neural networks. In *ICLR*, 2018.

[16] Guandao Yang, Tianyi Zhang, Polina Kirichenko, Junwen Bai, Andrew Gordon Wilson, and Christopher De Sa. Swalp : Stochastic weight averaging in low-precision training. In *ICML*, 2019.

[17] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.