A Computation Kernel for Network Binarization on PyTorch

Xianda Xu
University of Electronic Science and Technology of China
Chengdu, China
xiandaxu@std.uestc.edu.cn

Marco Pedersoli
Ecole de Technologie Suprieure
Montreal, Canada
Marco.Pedersoli@etsmtl.ca

Abstract

Deep Neural Networks have now achieved state-of-the-art results in a wide range of tasks including image classification, object detection and so on. However, they are both computation consuming and memory intensive, making them difficult to deploy on low-power devices. Network binarization is one of the existing effective techniques for model compression and acceleration, but there is no computation kernel yet to support it on PyTorch. In this paper we developed a computation kernel supporting 1-bit xnor and bitcount computation on PyTorch. Experimental results show that our kernel could accelerate the inference of the binarized neural network by 3 times in GPU and by 4.5 times in CPU compared with the control group. [1]

1. Introduction

Today, deep neural networks have achieved remarkable results in real-world tasks such as image classification [4][6], object detection [11][12], semantic segmentation [3][8] and so on. However, energy efficiency has become the bottleneck for deploying deep neural networks on small devices like mobile phones and embedded devices since they are area-and-battery constrained.

Current deep neural networks consist of hundreds of millions of parameters, which brings about two following issues. (1) Energy consumption. One hidden layer can be extracted as $A = \sigma(X \cdot W^T + B)$ where X denotes the input feature, W denotes the parameter matrix, B denotes the offset matrix and σ denotes the activation function. We notice that the $Gemm$ (general matrix to matrix multiplication) operation takes most computation time in both training and testing. (2) Memory cost. Normally, parameters in the deep neural network are stored in the format of float-32. AlexNet [6] achieved a remarkable performance but with 60 million parameters, it takes totally 240 MB for storage.

[1]Our code is now available at https://github.com/brycexu/BNN_Kernel.

Network binarization is one of the existing techniques for model compression and acceleration. It quantifies the model to binary numbers. In [2], both weights and activations are binarized and the model could achieve a 89 % accuracy on CIFAR-10.

Unfortunately, at the time of this writing, PyTorch [9] does not support 1-bit bitwidth operations and available binarized neural networks released using PyTorch by now are more of a simulation because they still use 32-bit floating point representation for the tensors and still use the $Gemm$-$Accumulation$ operation in the convolutional layer. There is no actual acceleration and compression in these models.

It inspired us to develop a computation kernel for network binarization on PyTorch. Firstly, it should be able to encode tensors from 32-bit floating point representation to 1-bit representation. Secondly, it should replace the $Gemm$-$Accumulation$ operation with the $Xnor$-$Bitcount$ operation.

Contributions we have made in our work include:

- We build a computation kernel using C and CUDA. It supports $Xnor$-$Bitcount$ operations on the encoded input and weight to get the output result.
- Our computation kernel can be wrapped into a Python loadable shared library thus it can be used on PyTorch.
- Experimental results show that our kernel could accelerate the inference of a binarized neural network by 3 times in GPU and 4.5 times in CPU compared to the control group.

The rest of the paper is organized as follows: In section 2, we talk about how convolution is implemented on PyTorch. In section 3, we propose our computation kernel and explain how it works. In section 4, experimental results on GPU and CPU are presented. We draw our conclusions in section 5 and make a discussion in section 6.

2. Convolution in Practice

In this section, we would like to talk about how convolution is normally implemented in practice like on PyTorch.

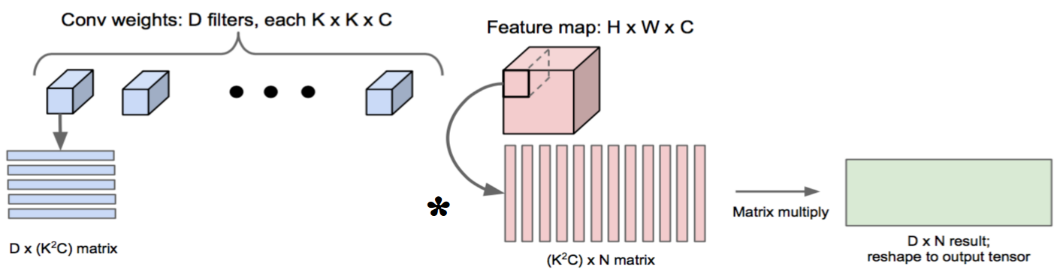
Conv weights: D filters, each K x K x C
Feature map: H x W x C
D x (K^{2}C) matrix
(K^{2}C) x N matrix
Matrix multiply
D x N result; reshape to output tensor

Figure 1. The $im2col$ Operation. The input feature map is $[H, W, C, 1]$ and is converted into a $[K^2C, N]$ matrix. The filter matrix is $[D, K, K, C]$ and is converted into a $[D, K^2C]$ matrix. It enables the $Gemm\text{-}Accumulation$ operation in the convolutional layer.

This may help understand what is changed in our computation kernel.

2.1. Im2col

For a convolutional layer, let X be the input feature map, W be the filter matrix and A be the output feature map. kH and kW represent the kernel height and the kernel width. C and D represent the input channel and the output channel.

The basic convolution operation to get each element $a_{i,j,d}$ in A is performed as:

$$a_{i,j,d} = \sum_{h=0}^{kH-1} \sum_{w=0}^{kW-1} \sum_{c} w_{d,h,w,c} \cdot x_{i+h,j+w,c}$$

We can see that this method is quite time-consuming. So, in practice, we convert the basic convolution operation to the $Gemm\text{-}Accumulation$ operation (general matrix to matrix multiplication) by using an operation named $im2col$ which arranges the data in a way that the output can be achieved by a simple matrix multiplication as shown in Figure 1.

Notice that the result here is a $[D, N, 1]$ matrix so we also need to reshape it to the output feature map. This process is the inverse process of $im2col$ which is called $col2im$.

2.2. Forward Graph

In this work, we only consider the acceleration in the inference of the binarized neural network. The forward graph used in PyTorch is shown in Figure 2.

Both the weight matrix and the input feature map are converted by the $im2col$ operation into matrixes which are then used to perform the $Gemm\text{-}Accumulation$ operation. The bias matrix also needs to be reshaped to perform the $addmm$ operation with the output of the $Gemm\text{-}Accumulation$ operation. The result shoud be converted by the $col2im$ operation before we get the output feature map.

3. Methology

3.1. Encoding

PyTorch normally uses the 32-bit floating points representation for the tensors, which is called $FloatTensor$. Available binarized neural networks implemented with PyTorch are more of a simulation because they are still using float-32 tensors in computation although they quantify these tensors to binary numbers with the $Sign$ function. So, the first thing we need to if we want to implement a "real" binarized neural network with PyTorch is to encode the tensors.

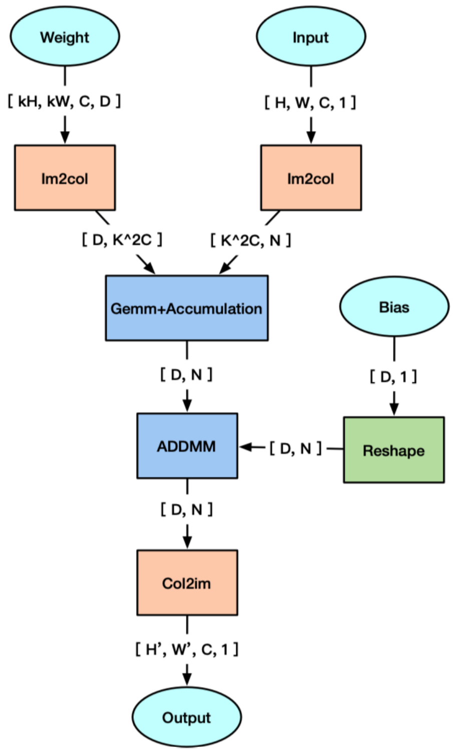

Figure 2. The Forward Graph for convolution computation utilized in most deep learning toolkits like PyTorch.

For the weight W, it manually skips the $im2col$ operation and is stored in a bitwise matrix. It is encoded in the direction of rows and each $w_{i,j}$ only takes 1 bit here. The dimention of this bitwise matrix is $[D, K^2C/32]$.

For the input X, it has to firstly pass the $im2col$ operation and gets reshaped into $[K^2C, N]$. Then, it is encoded in the direction of columns and also stored in a bitwise matrix. Each $x_{i,j}$ only takes 1 bit. The dimention of this bitwise matrix is $[K^2C/32, N]$.

In PyTorch, we use $IntTensor$ (int-32) for the weight and $FloatTensor$ (float-32) for the input and the output.

In our kernel, which is written in C, we use the datatype $uint32\_t$ for the bitwise weight matrix, the bitwise input matrix and the output matrix.

We are not saying that we are also running a simulation like other available binarized neural networks. We encode the input into 1-bit representation mannually and perform the bitwise $Xnor$-$Bitcount$ operation with the encoded weight. For each element in the bitwise weight matrix, it represents 32 1-bit weights. For each element in the bitwise input matrix, it represents 32 1-bit inputs. The forward graph in our kernel is shown in Figure 3.

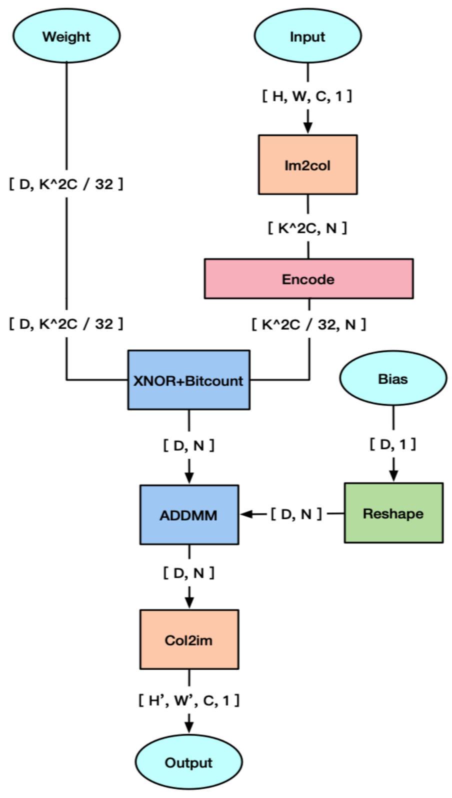

Figure 3. The Forward Graph for convolution computation utilized in our kernel. The input matrix has to be encoded into two bitwise matrixes and then with the encoded weight matrix, they perform the bitwise $Xnor$-$Bitcount$ operation.

Notice that both weights and activations are binarized to the binary values, which are either -1 or +1. These binary values are encoded with a 0 for -1 and a 1 for +1. To make it clear, we refer to the binary values -1 and +1 as binary ”values” and their encodings 0 and 1 as binary ”encodings”.

3.2. Xnor and Bitcount Operation

The reason why binarized neural networks can be accelerated is that they replace the $Gemm$-$Accumulation$ operation with the $Xnor$-$Bitcount$ operation, which is more hardware friendly and faster.

For the $Gemm$-$Accumulation$ operation between the float-32 $[D, K^2C]$ weight matrix and the float-32 $[K^2C, N]$ input matrix, each element in the output matrix is calculated as:

$$a_{i,j} = \sum_{k=0}^{K^2C-1} w_{i,k} \cdot x_{k,j}$$

Using an $Xnor$ operation on the binary encodings is equivalent to performing a multiplication operation on the binary values as seen in the following table:

Encoding (Value)	Encoding (Value)	Xnor (Multiply)
0 (-1)	0 (-1)	1 (+1)
0 (-1)	1 (+1)	0 (-1)
1 (+1)	0 (-1)	0 (-1)
1 (+1)	1 (+1)	1 (+1)

Table 1. It shows how the $Xnor$ operation on the binary encodings is equivalent to the multiplication operation on the binary values

Therefore, using an $Xnor$-$Bitcount$ operation on the binary encodings is equivalent to performing a $Gemm$-$Accumulation$ operation on the binary values. For the $Xnor$-$Bitcount$ operation between the $[D, K^2C/32]$ bitwise weight matrix and the $[K^2C/32, N]$ bitwise input matrix, each element in the output matrix is calculated as (since we use 0 to represent -1, it is different from what is calculated directly using -1):

$$a_{i,j} = \sum_{k=0}^{(K^2C/32)-1} 2Bitcount(\overline{w_{i,k} \otimes x_{k,j}}) - 32$$

3.3. Implementation

PyTorch provides the torch.nn module to help us create and train the neural network. THNN is a library that gathers torch.nn’s C implementations of neural network modules. It can be used in any application that has a C FFI. There is also a CUDA counterpart of THNN in PyTorch. It is the THCUNN library in the torch.cunn module, which provides a CUDA implementation for many functions in the torch.nn module.

The CPU implementation of our kernel for the forward graph is based on THNN_SpatialConvolutionMM_updateOutput in the THNN library. It is using the CPU tensor called THTensor. Besides, we add an encoding function and replace the $Gemm$-$Accumulation$ operation function with the $Xnor$-$Bitcount$ operation function. We keep the rest of its code unchanged.

The bitwise operator of $Xnor$ in C is $\sim (a \wedge b)$. We use the bit population count function for counting the number of one-bits from an external library named libpopcnt.h.

The GPU implementation of our kernel for the forward graph is based on THNN_SpatialConvolutionMM_updateOutput in the THCUNN library. It is using the GPU tensor called THCTensor. Like what we do in the CPU implementation, we add an encoding function and replace the $Gemm$-$Accumulation$ operation function with the $Xnor$-$Bitcount$ operation function. The rest of its code stays the same.

Both the encoding function and the $Xnor$-$Bitcount$ operation function are applied on CUDA. The bitwise operator of $Xnor$ is also $\sim (a \wedge b)$ while for the bit population count function, we use __popc() in the CUDA Toolkit.

3.4. Make an Extention for PyTorch

PyTorch provides a plethora of operations related to neural networks. But sometimes, a more customized operation is needed like you might want to use a novel activation function. To address such cases, PyTorch provides an easy way of writing the customed C and CUDA extensions.

After building the C and CUDA kernel, we need to connect the kernel with the Python backend in the THNN and THCUNN library. Then, in order to make it a Python loadable library, we can use the torch.utils.ffi.create_extension function in PyTorch. [2]

4. Experiments

4.1. Dataset

CIFAR-10 [5] is the dataset used in our experiments. It has totally 60,000 images within 10 classes. The size of each image is 32x32x3. It is divided into a training set with 50,000 images and a testing set with 10,000 images. In our experiments, we only use the testing set to test the speed of inference.

4.2. Model

Binarized Neural Network [2] is the network used in our experiments. It is the first network to quantify both weights and activations to binary numbers.

It uses Deterministic Binarization or the $Sign(x)$ function to binarize parameters, which is quite simple and fast. $Htanh(x)$ is the activation function in the Binarized Neural Network to deal with the gradient mismatch problem [7]. In backward propagation, gradients are not binary numbers and both weights and activations are updated with real-valued gradients.

Netscope is a tool to visualize neural network architectures. The visualized structure of the Binarized Neural Network used in our experiments can be found here.

[2]The tutorial for this is at https://github.com/chrischoy/pytorch-custom-cuda-tutorial.

4.3. Control Group

PyTorch has highly optimized implementations of its operations for CPU and GPU, powered by libraries such as NVIDIA cuDNN or Intel MKL. These libraries do not optimize the encoding operation and the $Xnor$-$Bitcount$ operation so we do not use them in our computation kernel.

We believe that it is unfair to compare our computation kernel directly with the PyTorch computation kernel highly optimized by those libraries so we introduce a control group for a fairer comparision.

Like our computation kernel, the kernel in the control group does not have any functions from NVIDIA cuDNN or Intel MKL, but it follows the forward graph used in PyTorch, as shown in Figure 2. It is a float-32 computation kernel so it does not encode the parameters to 1 bit and it performs the normal $Gemm$-$Accumulation$ operation between the weight matrix and the input matrix.

4.4. Results

We equip the Binarized Neural Network with our computation kernel. The computation kernel is only for convolution computation in the model since convolution computation takes up most of the time. We run our test on the inference of the model fed with the CIFAR-10 testing dataset. Results are shown in Table 4.4.

	CPU	GPU
PyTorch	301s	1.70s
Our Kernel	243s	3.57s
Control Group	1093s	11.23s

Table 2. Results

Our kernel is faster in the CPU inference of the Binarized Neural Network than both PyTorch and Control Group. It accelerates by about 4.5 times than the kernel in Control Group. Our kernel is faster than the kernel in the Control Group by about 3 times. The reason why our kernel is slower than the GPU computation kernel in PyTorch is that the later is highly optimized by NVIDIA cuDNN but ours not. The specifications of our testing environment is illustrated in the following Table 4.4.

CPU	Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
GPU	NVIDIA Geforce GTX 1080 Ti
RAM	11 GB

Table 3. The Testing Environment

5. Conclusion

In this work, we develop a computation kernel for network binarization on PyTorch. We make a C and CUDA extension that supports the encoding operation and the *Xnor-Bitcount* operation. Experimental results show that our kernel can accelerate the inference of the Binarized Neural Network by about 3 times in GPU and by about 4.5 times in CPU fed with CIFAR-10 testing dataset.

6. Discussion

It is known that the bitwise *Xnor-Bitcount* operation is much simpler than the float-point *Gemm-Accumulation* operation. This can lead to faster execution time. However, theorizing efficiency speedups is not always precise. For example, we have seen some papers like [10] using the total number of instructions in operations as a measure of execution time. We know that the 64-bit x86 instruction set allows a CPU to perform a bitwise *Xnor* operation within a single instruction while a float-32 *Gemm* operation takes 32 instructions. But it is not safe to conclude that the *Xnor* operations would have a 32x speed up over the *Gemm* operations because instruction and resource scheduling within a CPU is dynamic. We believe that instead of using the total number of instructions in operations as a measure of efficiency speed-ups in network binarization, it is better to look at the actual execution time.

As far as we know, the only available work testing the speed-ups in practice and releasing the code as well is from [2]. They observe a 23x speed up in GPU when binarizing the model. But they are using Theano [1] which is not highly optimized in GPU computation as PyTorch. They are also questioned by some people who try to reproduce their result in the speed up.

Our computation kernel is built and tested on PyTorch. Our kernel can be accelerated greatly both in CPU and GPU compared with the control group. It means that the bitwise operations could help binarized neural networks achieve a speed up in execution time. Although our kernel is a little bit faster in the CPU test than the CPU computation kernel in PyTorch, it is much slower than the GPU computation kernel in it. The reason is that the computation kernel in PyTorch is highly optimized by libraries like NVIDIA cuDNN or Intel MKL, especially the GPU computation kernel. Therefore, for the inference of binarized neural network on CPU, our kernel is faster, but on GPU, running the simulation on PyTorch seems a better idea unless NVIDIA cuDNN optimizes the 1-bit convolution computation in the future.

7. Acknowledgement

We would like to thank Mitacs Globalink Program for its funding to support Xianda to conduct the research internship.

References

[1] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4. Austin, TX, 2010.

[2] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[3] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[4] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[5] A. Krizhevsky and G. Hinton. Convolutional deep belief networks on cifar-10. *Unpublished manuscript*, 40(7):1–9, 2010.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[7] D. D. Lin and S. S. Talathi. Overcoming challenges in fixed point training of deep convolutional networks. *arXiv preprint arXiv:1607.02241*, 2016.

[8] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[9] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[10] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[11] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[12] K. Zhang, Z. Zhang, Z. Li, and Y. Qiao. Joint face detection and alignment using multitask cascaded convolutional networks. *IEEE Signal Processing Letters*, 23(10):1499–1503, 2016.