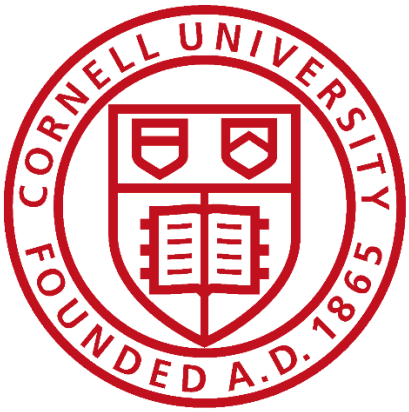
Implementation of Baseline on CPU, GPU for Binarized Neural Network (BNN) Acceleration



A Project Report

Presented to Professor Zhiru Zhang

Submitted by

Wentao Zhang (wz298)

Advisor: Prof. Zhiru Zhang

August, 2016

Abstract

This documents describes my part of the work in CIFAR-10 binarized neural network (BNN) hardware accelerator project. To set up the baseline for hardware accelerator, I did some test of BNN implemented with Theano framework [1] on different common-used platforms. Furthermore, I implement BNN on my own using Halide [2], a language and compiler for optimizing parallelism, to achieve better performance.

Executive Summary

[3] proposed BNN, a method which consists in training a DNN with binary weights during the forward and backward propagations, while retaining precision of the stored weights in which gradients are accumulated. They obtain near state-of-the-art results with BNN on the permutation-invariant MNIST, CIFAR-10 and SVHN. In BNN design, an important boost in inference can be obtained by utilize the inherent parallelism, the reduced data type and the properties of devices such as GPUs and FPGAs. For my work, I tested the inference part of CIFAR-10 implemented with Theano deep learning framework on different platforms, both on x86 CPU on a server and ARM CPU on a board, with and without the cooperation of GPU, to set up the baseline for hardware accelerator which is the essential goal of the whole project. In addition, I implemented a new version of CIFAR-10 inference in C++ using Halide, a language and compiler for optimizing parallelism, which has a better performance. In this version, each layer contains the definition which specifies the calculation, and schedule function which specifies the optimization, so it's also reusable.

Acknowledgements

I would like to thank Professor Zhiru Zhang and the rest of Zhang Group for their help and advice during the execution of this project. Special thanks to Ritchie Zhao for his collaboration in the development of some of the algorithms for BNN, preprocessed dataset and for their code.

I would also like to thank Steve Dai for environment set up on Jetson board.

Table of Contents

1 Introduction

1.1 Motivation

There are several ways to accelerate the inference of binarized neural network (BNN). First of all, the algorithm can be parallelized easily due to lots of repetitive computation on different parts of data. For another, the weights in each layer are expressed in less bits than float number. Furthermore, connected layers can be pipelined to improve the throughput of the whole network. We utilized these features in common platforms like CPUs and GPUs, to see how fast it can reach.

1.2 Overview

The BNN architecture is nearly identical to that found in Binaryconnect [4].

The BNN inference code is written by Ritchie Zhao in Python with Theano and Lasagne deep learning framework. First, I test this code on a server with high performance CPU and GPU. After that, I set up the running environment on Jetson TK1, a board has an ARM core and an embedded GPU.

Moreover, I implement a version of CIFAR-10 inference with Halide, using the feature of parallelization, vectorization and pipelining.

2 Architechture of Binarized Neural Network

The CIFAR-10 dataset we use contains a test set of 10000 32x32 3-channel images, each images should be classified to a label from all 10 labels.

The BNN architecture use six 3x3 convolutional (conv) layers and three fully-connected (FC) layers. After each of these layers there is a batch normalization (batch-norm) layer and a binary activation layer. In addition, there are three 2x2 max-pooling(pool) layers, which are placed after the 2nd, 4th, and 6th conv layer before the batch-norm. The architecture, size and needed computation for multiplication of each layer is documented in Table 1.

The first conv layer is different from the rest. Each conv layer takes in a set of binary feature maps and produces non-binary output feature maps. The first conv layer, however, takes in the 32-bit fixed-point pixels of the input image directly as input. Each pool layer reduces a image width and height by a factor of two. The input images are 32×32 so the feature maps at the end of the last conv layer is 4×4. This gets fattened into an array of 512×4×4=8192 inputs for the FC layer.

Layer	Input Maps	Dim	Output Maps	Computation (multiplication)	
Conv1	3	32x32	128	3x128x32x32x3x3	3538944
BatchNorm	128	32x32	128		
Conv2	128	32x32	128	128x128x32x32x3x3	150994944
Pool	128	16x16	128		
BatchNorm	128	16x16	128		
Conv3	128	16x16	256	128x256x16x16x3x3	75497472
BatchNorm	256	16x16	256		
Conv4	256	16x16	256	256x256x16x16x3x3	150994944
Pool	256	8x8	256		
BatchNorm	256	8x8	256		
Conv5	256	8x8	512	256x512x8x8x3x3	75497472
BatchNorm	512	8x8	512		
Conv6	512	8x8	512	512x512x8x8x3x3	150994944
Pool	512	4x4	512		
BatchNorm	512	4x4	512		
FC1	8192	1	1024	8192x1024	8388608
BatchNorm	1024	1	1024		
FC2	1024	1	1024	1024x1024	1048576
BatchNorm	1024	1	1024		
FC3	1024	1	10	1024x10	10240
BatchNorm	10	1	10		

Table 1

3 Baseline with Theano framework

3.1 on server

Environment setup on server is quite easy while using pip, a python package manager, to install external library like Theano, Lasagne, Pylearn2 and so force. Thanks to Theano framework, we can test the code not only on CPU but also with the cooperation of GPU, which can be specified by one line instruction in ".theanorc" file.

On the other side, the memory resource is sufficient to hold the whole test data in BNN model, so I do not need to change the input size. However, things are different in Jetson TK1 board, so I introduce the variable "batch size" to compare the performance on these two platforms. Batch size stands for the number of input images processed in BNN at one time. We can reasonably guess large batch size will lead to a less run time due to the increasing parallelism. The test result proved this guess was partially correct.

3.2 on Jetson TK1 board

On Jetson TK1, many modules had to be installed manually instead of using package manager. I downloaded several packages including BLAS and CUDA, then installed them and set up the environmental variables manually. After that, with all dependencies solved, I could use package manager to install gfortran and SciPy. Finally, I successfully installed Theano from source code.

A problem came out when I tried to run the code. It was due to the memory limitation on the board. Therefore, I changed something in the code. The first one was dividing 10000 input images into small batches. Secondly, I changed the Theano settings to keep the embedded GPU occupy no more than 60% of its Video RAM (VRAM).

3.3 Test result

Environment:

Server:

CPU: Intel® Xeon® CPU E5-2640 v3 @ 2.60GHz

GPU: NVIDIA Tesla K40c

Jetson TK1 board:

Tegra K1 SOC

GPU: NVIDIA Kepler GPU with 192 CUDA Cores

CPU: NVIDIA 4-Plus-1™ Quad-Core ARM® Cortex™-A15 CPU

Figure 1 shows the runtime for per image in different environments.

Figure 2 shows the number of images that can be processed in one second in different environments. To be more clear, Figure 3 excludes the line of “server GPU”. The numerical data can be seen in Table 2.

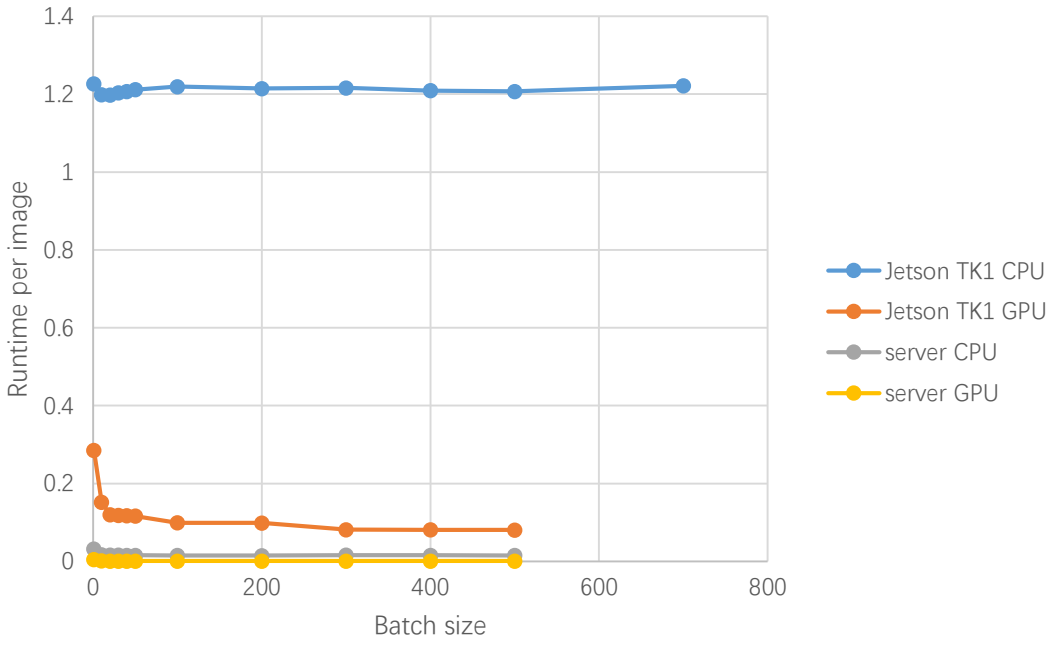


Figure 1

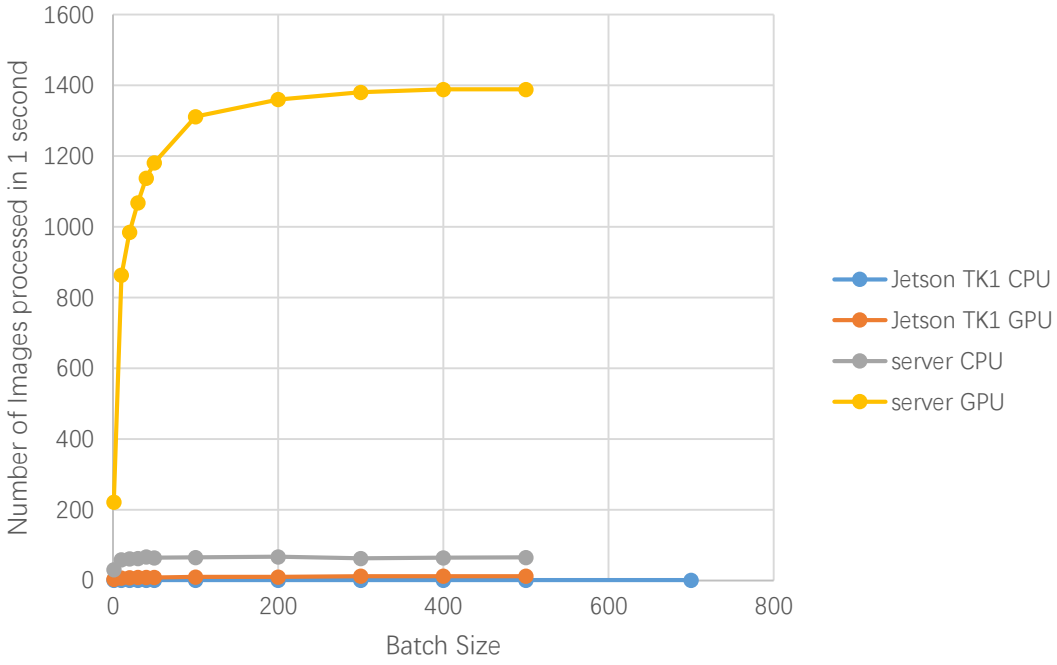


Figure 2

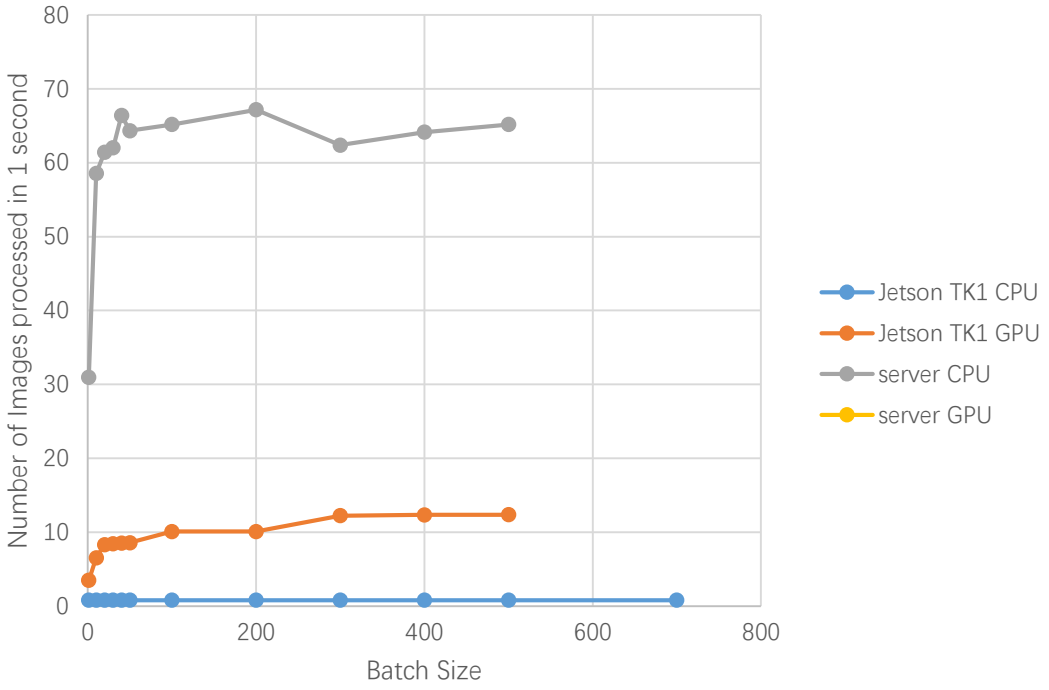


Figure 3

Batch size	1	10	20	30	40	50	100	200	300	400	500
Jetson TK1 CPU	1.227	1.198	1.198	1.204	1.206	1.211	1.219	1.214	1.215	1.209	1.206
Jetson TK1 GPU	0.285	0.152	0.12	0.118	0.117	0.116	0.098	0.098s	0.081	0.08	0.08
server CPU	0.032	0.017	0.016	0.016	0.015	0.015	0.015	0.015	0.016	0.016	0.015
server GPU	0.0045	0.0012	0.001	0.0009	0.001	0.0008	0.0008	0.0007	0.0007	0.0007	0.0007

Table 2. Runtime(seconds) in different environments

4 Baseline implemented by Halide

4.1 Approach

To do this part of work, we can focus on the optimization of one conv layer. Take the second conv layer for example. It has 128 input feature maps and 128 output feature maps. We need to know all input feature maps to compute one output feature map, this computation is called 3D-convolution. See the pseudo code.

algorithm 1 3D-Convolution

input: input feature maps, convolution kernel
output: convolution result

```
1: function CONVOLUTION(input, kernel)
2:     result ← 0
3:     for i = 0 → N − 1 do
4:         for x = 0 → 31 do
5:             for y = 0 → 31 do
6:                 for r.z = 0 → M − 1 do
7:                     for r.x = 0 → 2 do
8:                         for r.y = 0 → 2 do
9:                             output[i][x][y] += kernel[i][j][r.x][r.y] * input(x + 1 − r.x, y + 1 − r.y, r.z)
10:                        end for
11:                    end for
12:                end for
13:            end for
14:        end for
15:    end for
16:    return output
17: end function
```

Figure 4. Naive 3D-Convolution

algorithm 2 3D-Convolution optimized

input: input feature maps, convolution kernel
output: convolution result

```
1: function CONVOLUTION(input, kernel)
2:     result ← 0
3:     for i = 0 → N − 1 do in parallel
4:         for x = 0 → 31 do 16 bits vectorize
5:             for y = 0 → 31 do
6:                 for r.z = 0 → M − 1 do
7:                     for r.x = 0 → 2 do
8:                         for r.y = 0 → 2 do
9:                             output[i][x][y] += kernel[i][j][r.x][r.y] * input(x + 1 − r.x, y + 1 − r.y, r.z)
10:                        end for
11:                    end for
12:                end for
13:            end for
14:        end for
15:    end for
16:    return output
17: end function
```

Figure 5. Optimized 3D-Convolution

To speed it up, I parallelize the outmost loop and vectorize image's X coordinate, both two optimizations are well supported by Halide schedule function. In dense layers, I also apply the similar optimization. Besides, I use library function compute_root where we need to store the intermediate result. Moreover, due to the binarized parameters, we can use 16-bit-integer rather than float point in calculation. These schedule can be seen in my source code[1].

[1] https://github.com/nsknojj/bnn-halide

GPU version has not been completed yet, but it can be found in branch “GPU” at the same Github website. At present, the speed with the GPU schedule function is slower than that with CPU optimization.

4.2 Modules

- data: contains script to download data
- params: contains script to download parameters
- cpp: Just-In-Time compilation version
- compiled_cpp: Ahead-of-Time compilation version, to run it, you should first run the JIT version’s code, and that will generate the obj file “compiled_network.o” and header “compiled_network.h”. Move them to directory “baseline” and then we can compile the AOT version.

4.3 Result

To do this test, I modify the data type of the variables used for storing intermediate result and the output of each layer, as well as the layer definition, whether using schedule function or not.

Compared with the runtime without schedule, we can see a significant speed-up while using schedule function. Also, data type could influence the runtime a little.

Data type	Use schedule	No schedule
16-bit-int	0.009s	0.356s
32-bit-int	0.013s	0.332s
32-bit-float	0.013s	1.164s

Figure 6. Runtime (seconds) per image

5 Conclusion

Parallelization of inference algorithms can give machine learning applications a noticeable performance boost. The computation of convolution and matrix multiplication are both highly parallelizable, and using the efficiency of multi-core CPU for parallel operations, a speed-up can be obtained even for relatively small models. Moreover, more speed-up can be obtained if we implement it on GPUs.

Although CPU and GPU run times are not directly comparable (they are different devices), some configurations of the GPU algorithms are slower than the serial CPU implementations, which shows that the appropriate use of parallelism can indeed improve the execution time.

Future research could reveal whether further improvements over the baselines can be achieved. Unexplored alternatives include different parallelization schemes and a more thorough optimization of the training kernel, which can yield even higher speed-ups in the inference of BNN. The use of the same schemes on larger models might also result in a more noticeable acceleration of the training process.

6 Reference

[1] Al-Rfou, R., Alain, G., Almahairi, A., & Angermueller, C. (2016). Theano: A Python framework for fast computation of mathematical expressions. arXiv eprints abs/1605.02688 (May 2016). url: http://arxiv. org/abs/1605.02688.

[2] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., & Amarasinghe, S. (2013). Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. ACM SIGPLAN Notices, 48(6), 519-530.

[3] Courbariaux, M., Hubara, I., Soudry, C. D., El-Yaniv, R., & Bengio, Y. Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 or −1.

[4] Courbariaux, M., Bengio, Y., & David, J. P. (2015). Binaryconnect: Training deep neural networks with binary weights during propagations. In Advances in Neural Information Processing Systems (pp. 3123-3131).

[5] Qiu, J., Wang, J., Yao, S., Guo, K., Li, B., Zhou, E., ... & Wang, Y. (2016, February). Going deeper with embedded fpga platform for convolutional neural network. In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (pp. 26-35). ACM.