GPU enabled deep learning framework

Evgeniy Zheltonozhskiy
https://github.com/Randl
https://www.linkedin.com/in/zheltonozhskiy/

14th May 2017

Abstract

GPU are very suitable for deep learning (DL) purposes, providing significant (sometimes almost two orders of magnitude) speedups [1] compared to CPU. Currently, most of deep learning frameworks support GPU both for training and inference. However, those frameworks are pretty heavy and have many dependencies. I propose to add GPU support to a simple DL framework with a small amount of dependencies.

1 Motivation

Neural networks (NN) have become a very hot topic lately and lots of people are interested in trying to apply NNs to their problems. However, most of modern NN and DL frameworks are not that simple to use, which might be a dealbreaker for some users outside of machine learning (ML) field.

tiny-dnn[1] is a C++11 header-only DL library. Its main advantages are ease of integration and usage, dependency-freedom and portability. However, as for now, tiny-dnn doesn't support GPUs, which is a standard for the modern DL frameworks.

Due to lack of GPU support, tiny-dnn can't provide competitive performance, especially in training. Although the library is pretty easy to use and dependency-free, most of users and researchers need higher performance. As a result, usefulness of the library is somewhat limited, since the people who are seeking for a simple and easy-to-add library for DL are forced to choose between simplicity and speed.

I believe that the addition of GPU support will greatly increase the usage of tiny-dnn by researches and engineers, providing simple and powerful instrument for DL. The possibility of now unrealistic (unless you are ready to wait couple of months) training of big models will attract new users and encourage the current ones for a wider usage of the library. This feature is pretty often requested by users [2]

[1] https://github.com/tiny-dnn/tiny-dnn
[2] Issues about GPU in tiny-dnn GitHub repository

2 Proposal

As for now, other tiny-dnn team members and me are working on Tensor class implementation which is supposed to help us to introduce GPU support. The structures which are currently used are less suitable for GPU computations, so this refactoring is a necessary step before future GPU support introduction and is estimated to be done before summer.

We intended to use CLCudaAPI[3] to allow users to use both OpenCL and CUDA. Most of frameworks support only CUDA, which significantly limits people who haven't NVIDIA GPU or want to use FOSS in their work.

Basic elements of deep neural networks (DNNs) are convolutions, poolings and fully connected (FC) layers. For convolution and poolings, libdnn[4], which supports both CUDA and OpenCL, might be used. As for fully connected, one possible simple solution is to consider fully connected layer as a convolution with filter of size 1×1[5] [2]. This will allow relatively simple implementation of main elements of DNNs with as few dependencies as possible. In this project, I want to exclusively use existing kernels and focus on high level integration instead of low-level DL kernel development. In addition, I believe it would be better to contribute to existing kernels instead of rewriting them from scratch, unless that's unavoidable.

All new code should be covered with tests and be tested in current CI systems (Travis CI).

3 Approximate timetable

Date	Task
1st April to 31st May 2017	Finishing Tensor implementation
1st to 7th June 2017	Revising, fixing and refactoring existing code
7th June to 1st July 2017	Integrating libdnn with Tensor
1st July to 1st August 2017	Implementing basic layers: convolutions, poolings, fully connected
1st to 14th August 2017	Benchmarking, writing tests and optimization
14th August to 1st September 2017	Finishing the work, fixing bugs and testing everything

Finishing Tensor implementation First of all, since the current code structure is less suitable for work with GPU, we decided to refactor it, implementing a new class, called Tensor, similarly to most other DL frameworks. Another option is to use one of existing tensor implementation, for example, xtensor [6].

[3] https://github.com/CNugteren/CLCudaAPI
[4] https://github.com/naibaf7/libdnn
[5] Yann LeCun on facebook about FC being convolutions with 1×1 filter
[6] https://github.com/QuantStack/xtensor

On the one hand, this will introduce the first hard dependency in the library, but on the other hand this will allow us to focus on developing the library itself and not to work on Tensor development, which is non-trivial itself.

Revising, fixing and refactoring existing code There already were some attempts to implement OpenCL support for tiny-dnn, so first of all I'm going to adapt the existing code to the new code model if possible (and removing it if not), make sure tests are passed and code works as planned.

Integrating libdnn with Tensor The next step is to integrate libdnn with the chosen Tensor implementation, i.e. implementing interfacess to pass our data to libdnn and GPU in a way it understands it. After this stage is finished, it will already be possible to call GPU functions from tiny-dnn.

Implementing basic layers: convolutions, poolings, fully connected At this stage I will actually implement neural networks layers for GPU. In terms of tiny-dnn, this means implementing backend which is responsible for performing all the calculations (or, in our case, passing them to the GPU).

Benchmarking, writing tests and optimization After initial implementation is ready I'll proceed to benchmarking and testing. For benchmark I am going to use a number of different GPUs and CPUs available for me, both mobile and desctop, including latest Nvidia's GPUs like GTX1080. Unfortunately, I don't have any AMD production available, but, hopefully, someone with kindly run benchmarks and tests on AMD hardware for me.

Another important part of benchmarking is profiling and optimization of performance. I believe it's possible to achieve much higher performance compared to current CPU implementation even without optimizations, but high performance is always important for DL frameworks users, which means we need to dedicate some time to optimizations.

Finishing the work, fixing bugs and testing everything Last stage of the work is making sure everything works as intended, there are no bugs that disturb users, all tests are passed and coverage is high. After rechecking this couple of times and getting feedback from users, we can consider the work done.

4 Biographical Information

I'm a first year undergraduate in Technion – Israel Institute of Technology, participant of Technion's excellence program. I'm a research assistant in a professor Avi Mendelson[7] group working on neural networks for an embedded hardware. The team also works with GPUs as a main instrument of modern neural networks research.

[7] https://www.linkedin.com/in/avi-mendelson-04a0a51

As for now, I'm working on a project which is focused on Binarized Neural Networks and their implementation on embedded hardware.

As for programming, my main relevant experience is currently being one of contributors of tiny-dnn.

References

[1] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking state-of-the-art deep learning software tools," *CoRR*, vol. abs/1608.07249, 2016.

[2] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. A. Riedmiller, "Striving for simplicity: The all convolutional net," *CoRR*, vol. abs/1412.6806, 2014.