Towards Transparent Neural Network Acceleration

Anonymous Author(s)
Affiliation
Address
email

Abstract

1 Deep learning has found numerous applications thanks to its versatility and accu-
2 racy on pattern recognition problems such as visual object detection. Learning and
3 inference in deep neural networks, however, are memory and compute intensive
4 and so improving efficiency is one of the major challenges for frameworks such
5 as PyTorch, Tensorflow, and Caffe. While the efficiency problem can be partially
6 addressed with specialized hardware and its corresponding proprietary libraries,
7 we believe that neural network acceleration should be transparent to the user and
8 should support all hardware platforms and deep learning libraries.

9 To this end, we introduce a transparent middleware layer for neural network
10 acceleration. The system is built around a compiler for deep learning, allowing one
11 to combine device-specific libraries and custom optimizations while supporting
12 numerous hardware devices. In contrast to other projects, we explicitly target
13 the optimization of both prediction and training of neural networks. We present
14 the current development status and some preliminary but encouraging results:
15 on a standard x86 server, using CPUs our system achieves a 11.8x speed-up for
16 inference and a 8.0x for batched-prediction (128); on GPUs we achieve a 1.7x and
17 2.3x speed-up respectively.

18 ## 1 Introduction

19 The limitations of today's general purpose hardware and the extreme parallelism that neural network
20 processing can exploit has led to a large range of specialized hardware from manufacturers such as
21 NVIDIA [13], Google [7], ARM [1] and PowerVR [15], to name but a few. Most of these platforms
22 come with their own proprietary development environment or a specialized extension to some deep
23 learning frameworks such as TensorFlow [8], PyTorch [5], CNTK [12], and Caffe [4]. Often, such
24 hardware makes it necessary to transform neural network models from one framework to another in
25 order to utilize different hardware architectures. While standardized formats [6, 11] try to bridge this
26 gap, they cannot guarantee that an exported network behaves identically in all frameworks.

27 In addition to the hardware support for deep learning frameworks, the usage model itself can differ.
28 For example, PyTorch is known to be very flexible thanks to its dynamic graph structure, while
29 TensorFlow uses a static graph that is more restricted, but usually yields better performance. These
30 differences are dealt with through different strategies. The big hardware manufacturers such as Intel
31 [9] or NVIDIA [3] provide optimized libraries for the most performance-critical functionality. As an
32 example, PyTorch introduced the so called Tensor Comprehensions [16], which is somewhat similar
33 to Vertex.ai's PlaidML [17]. Both require the neural network layers to be programmed in a tensor
34 mathematical notation, which is then compiled into a specialized implementation. However, they are
35 only capable of optimizing the functionality inside a single layer, not across multiple layers. Other
36 approaches such as TensorRT [14] or TVM [2] compile optimized implementations for NN prediction
37 deployment, which means that they cannot be used to optimize training. As training can take up to
38 several days or weeks of computation, even small improvements are often meaningful.

Submitted to 32nd Conference on Neural Information Processing Systems (NIPS 2018). Do not distribute.

To go beyond these limitations, we propose a modular middleware for NN processing, designed to optimize not only prediction but also training computations. It interfaces seamlessly with existing frameworks and accelerates neural networks on various types of hardware. The system performs such work transparently, allowing data scientists to concentrate on the design of neural networks without having to deal with framework or hardware specific issues. To use our system, the user simply adds a line of code of the form `optimizedNN = optimize(myNN)`. Finally, our middleware can be easily extended to interface with other AI frameworks and hardware platforms.

In the following we will introduce our optimization cycle, followed by our system architecture, some preliminary results and close with a description of our future development plans.

2 Transparent Neural Network Optimization

The proposed middleware consists of multiple stages. The first stage directly operates on the neural network structure and applies several optimization heuristics. Tensor concatenation operations, for instance, are common and provide multiple opportunities for improvements such as moving layers in front of the concat layer to reduce the amount of data needed to be processed (if a pooling layer is moved), and the merging of multiple concat layers. Moreover, if we generate code for the preceding layers (see below), data can be directly written into the destination memory without an explicit memcopy. An additional optimization step merges consecutive MaxPooling and ReLU layers.

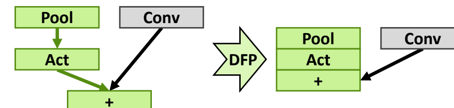

Figure 1: Illustration of the depth-first parallelization strategy.

In the next stage, the system attempts to fuse multiple layers. To this end, it analyzes the network structure and detects blocks of layers that have similar limitations. We distinguish between (1) I/O memory bound (e.g., pooling), (2) parameter memory bound (e.g., fully connected) or (3) compute bound (e.g., convolutions) layers. We group consecutive layers with the same limitations. Element-wise layers (e.g., ReLU) do not impose a specific limitation and can be assigned to any group. Each of these groups is then optimized separately. For now we use optimized vendor libraries for compute and parameter memory bound layers, e.g., Intel's MKL-DNN [9] or NVIDIA's cuDNN [3] (in the following referred to as DNN) as these operations benefit from specialized algorithms.

Blocks of I/O memory-bound layers are optimized using a principled merging strategy based on the notion of depth-first parallelism (DFP) [18]. Instead of processing the networks layer-by-layer, DFP generates specialized code, merging several layers into one and improving cache utilization. More specifically, in this stage the system optimizes each I/O memory bound group of layers in several steps. First, a computation graph for all operations is constructed. With this graph we generate a naïve plan of nested loops that would be necessary to compute the graph. From this we apply loop transformations to merge these nested loops; this step is generic and identical for all target devices. Next, we use hardware characteristics (e.g., number of cores, SIMD units per core and cache sizes) to generate specific mappings of loops onto compute resources. Figure 1 illustrates a merging operation for a small neural network.

Depending on the hardware, we further exploit device-specific characteristics (shared memory, approximate mathematical functions, OpenMP flags, etc.). After all groups of layers are optimized and an implementation tailored to the target hardware is compiled, we return a new executable neural network representation specific to the deep learning framework to the user. This optimized network behaves identical to the original network.

3 Architecture

Our architecture (Figure 2) is modular and, therefore, highly extensible. We use deep learning framework-specific frontends for translating the NN representation of the framework to that of our middleware and provide an optimized NN representation to the user. In addition, a runtime component bridges framework specific functionality such as memory (de-)allocation. Our system applies different optimizations depending on the performance bottleneck (e.g., CPU bound) of each layer. Finally, the device backends implement these optimizations and apply device-specific optimizations; for example,

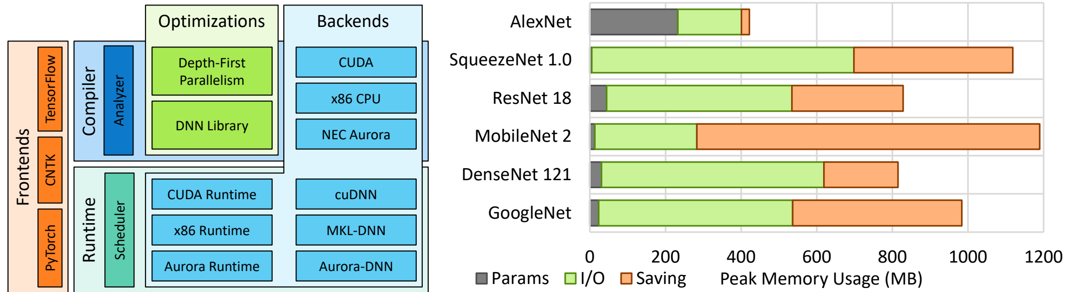

Figure 2: (**Left**) Our architecture can easily be extended with more frontend, optimizations and backend modules. (**Right**) Peak parameter and I/O memory data consumption (Batch size: 128).

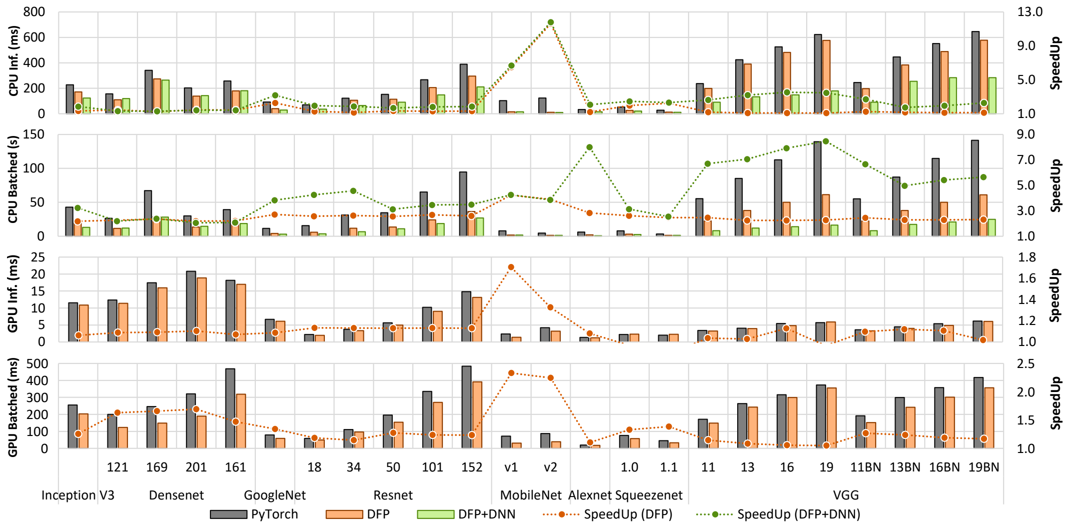

Figure 3: Execution times and speed ups for Inference and Batched-prediction (128) for 8 different neural networks in 24 variants.

our x86 backend uses OpenMP and the ISPC [10] compiler for the I/O memory bound layers, and the Intel MKL-DNN library for other layers.

4 Preliminary Results

Currently, our system can run prediction tasks on both CPUs and GPUs. To test its performance, we use a server with 2x Intel E5-2637 v4 CPUs, 128GB DDR4, an NVIDIA GTX 1080 Ti card, Debian 9.5 (Kernel 4.9.0-3), ISPC 1.9.2, GCC 8.2.0, CUDA 9.2.148, cuDNN 7.2 and PyTorch 0.4.1. We run each test 20 times and show the best result (Figure 3). For all layers not optimized by the DFP method we use the default PyTorch implementation. As PyTorch uses cuDNN by default, we do not show results for (DFP+DNN) on GPUs. We applied our system to a set of typical networks for inference and batched prediction. We can see that depending on the network structure, DFP or DNN yield the highest performance gains, e.g., DFP in the MobileNets and DNN in AlexNet and the VGGs. Overall, we achieve a peak improvement of 11.8x for inference and 8.0x for batched-prediction (128) on CPUs; and a 1.7x and 2.3x speed-up respectively on GPUs. Figure 2 further shows that the DFP method can significantly reduce neural network peak memory consumption.

5 Status and Future Work

We plan to provide support for PyTorch, TensorFlow and CNTK as frontends; x86 CPUs, NVIDIA GPUs and NEC Aurora as backends; and applying the previously mentioned optimizations for both prediction and training. Further, we plan to add more optimization features targeting other types of network layers, such as those found in recurrent neural networks.

References

[1] ARM. ARM project trillium. arm.com/products/processors/machine-learning.

[2] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018. arxiv.org/abs/1802.04799.

[3] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cuDNN: efficient primitives for deep learning. *arXiv*, 2014. arxiv.org/abs/1410.0759.

[4] Facebook. Caffe2. caffe2.ai.

[5] Facebook. PyTorch. pytorch.org.

[6] Facebook and Microsoft. Open neural network exchange format. onnx.ai.

[7] Google. Tensor processing unit. cloud.google.com/tpu.

[8] Google. TensorFlow. tensorflow.org.

[9] Intel. Intel Math Kernel Library for Deep Neural Networks. github.com/intel/mkl-dnn.

[10] Intel. Intel SPMD program compiler. ispc.github.io.

[11] Khronos. Neural network exchange format. khronos.org/nnef.

[12] Microsoft. Cognitive toolkit. microsoft.com/en-us/cognitive-toolkit.

[13] NVIDIA. Tensor cores in NVIDIA volta. nvidia.com/en-us/data-center/tensorcore.

[14] NVIDIA. TensorRT. developer.nvidia.com/tensorrt.

[15] Power VR. PowerVR Series2NX. imgtec.com/powervr-2nx-neural-network-accelerator.

[16] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. 2018. research.fb.com/announcing-tensor-comprehensions/.

[17] Vertex.ai. PlaidML: open source deep learning for every platform, 2017. vertex.ai/blog/announcing-plaidml.

[18] N. Weber, F. Schmidt, M. Niepert, and F. Huici. BrainSlug: Transparent Acceleration of Deep Learning Through Depth-First Parallelism. *arXiv*, 2018.