Sol: Transparent Neural Network Acceleration Platform

Nicolas Weber
NEC Laboratories Europe
nicolas.weber@neclab.eu

ABSTRACT

With the usage of neural networks in a wide range of application fields, the necessity to execute these efficiently on high performance hardware is one of the key problems for artificial intelligence (AI) framework providers. More and more new specialized hardware types and corresponding libraries appear from various manufacturers. The biggest problem arising is that these libraries usually are only supported by a very limited set of AI frameworks and interoperability can become an issue. In this extended abstract we present Sol, a transparent middleware for neural network acceleration. Sol comes with an optimizing compiler engine, allowing to use device specific libraries and to implement own optimizations, that can be leveraged on all target devices. In contrast to other projects Sol explicitly aims at optimizing prediction and training of neural networks.

CCS CONCEPTS

• **Computing methodologies** → *Neural networks*; • **Software and its engineering** → *Middleware*; *Just-in-time compilers*;

KEYWORDS

Neural Networks, Transparent Acceleration, CPU, GPU, NEC Aurora

ACM Reference Format:
Nicolas Weber. 2018. Sol: Transparent Neural Network Acceleration Platform. In *Proceedings of SuperComputing (SC'18).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Artificial intelligence (AI) and neural networks (NN) are present in nearly all fields of our daily life. Autonomous driving, fraud detection, automated stock exchange, and medical computing are prominent examples of AI applications today. With increasing popularity the number of frameworks to develop AI systems has risen. Due to the limitations of todays hardware and the extreme parallelism applicable in NN processing, specialized hardware architectures are developed by a wide range of manufacturers, such as NVIDIA [14], Google [7], ARM [1], PowerVR [16], and many more. Usually all of these come with their own development environment or as an extension to one of the more prominent frameworks, such as TensorFlow [8], PyTorch [5], CNTK [13] or Caffe [4]. This can make it necessary to transform neural network models from one framework to another, in order to utilize different hardware architectures. Formats such as ONNX [6] or NNEF [12] try to bridge this gap, but they do not guarantee that an exported network behaves identical in all frameworks.

SC'18, November 2018, Dallas, Texas USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Not only the hardware support can vary between frameworks, but also their usage model. PyTorch is known to be very flexible due to its dynamic graph structure, while TensorFlow uses a static graph that is more restricted, but usually yields in better performance. To increase the performance of these frameworks, different approaches are pursuit. The big hardware manufacturers such as Intel [9] or NVIDIA [3] provide optimized libraries for the most important functionality. PyTorch introduced the so called Tensor Comprehensions [17], which is somewhat similar to Vertex.ai's PlaidML [18]. Both require the neural network layers to be programed in a tensor mathematic notation, which is then compiled into a specialized implementation. However, they are only capable of optimizing the functionality inside a single layer, not across multiple layers. Other approaches such as TensorRT [15] or TVM [2] compile optimized implementations for NN prediction deployment. These cannot be used to optimize training. As training can take up several days or weeks of computation, even small improvements are important (10% of one week are 17h!). As to our knowledge, only Intel's NGraph [11] allows training. It tries to provide an abstraction layer between AI frameworks and optimized neural network libraries.

To get beyond these limitations, we propose Sol, which is a modular middleware for NN processing, designed to optimize not only prediction but also training computations. It interfaces seamlessly into frameworks and transparently accelerates neural networks on various types of hardware.

We designed it to work fully autonomous, so that data scientists can concentrate on the design of neural networks and do not need to bother with framework or hardware specific issues. Therefore Sol's user interface does not have any adjustable parameters. The user only needs to execute `optimizedNN = sol.optimize(myNN, [0, 3, 224, 224])`, where the first parameter is the neural network and the second the data input size (Values unknown at compile time can be specified with a 0). Sol can be easily extended to interface with 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 an outlook on our future development plans.

2 OPTIMIZATIONS

To optimize neural networks, we employ multiple optimization stages. The first directly operates on the network structure. Concat operations provide multiple ways of optimizations, e.g., moving layers in front of the Concat, to reduce the amount of data needed to be processed by the Concat (if a pooling layer is moved), merging of multiple Concat layers, or if we generate code for the preceding layers (see below), data can directly be written into the destination memory, without an explicit memcopy. Another optimization is to merge consecutive MaxPooling and ReLU layers.

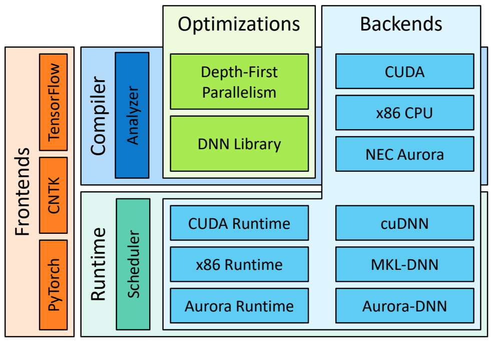

Figure 1: SoL's architecture can easily be extended with more frontend, optimization and backend modules.

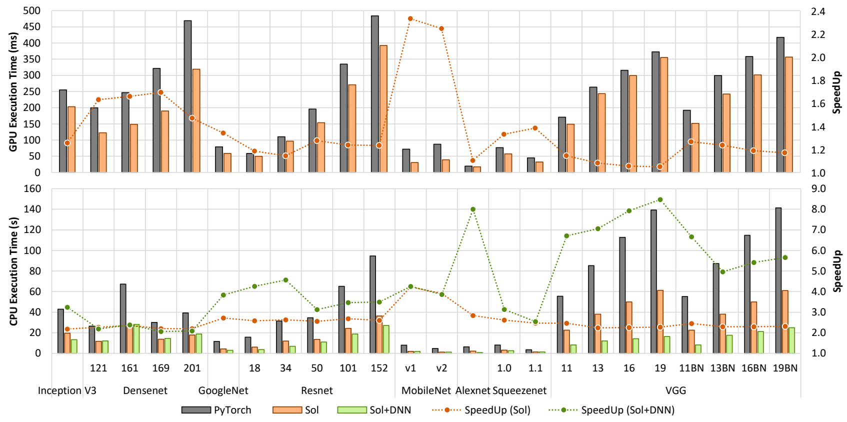

Figure 2: Batched-prediction (128) execution times and speed ups for different 8 neural networks in 24 variants.

In the next stage, we evaluate fusion of multiple layers. For this SoL analyzes the network structure and detects areas of layers that have similar limitations. We distinguish between I/O memory bound (e.g., pooling), parameter memory bound (e.g., fully connected) or 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 different groups.

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 [3] (in the following referred as DNN) as these operations mainly benefit from specialized algorithms. For each of these groups we compile a small control library with specialized code for all layers, that employs memory reuse, to reduce the number of allocations and control-flow instructions to an absolute minimum.

The I/O memory bound groups are optimized using the depth-first parallelism (DFP) method [19]. In contrast to default layer-by-layer processing, DFP generates specialized code, that for each data entry applies all layer operations before continuing to process the next data entry. This improves the cache utilization for the I/O data.

Our implementation optimizes each I/O memory bound group in several steps. First, we generate a computation graph of all operations, in an internal intermediate representation. With this graph we generate a naïve plan of nested loops that would be necessary to compute the graph. Then we apply loop transformations to merge these nested loops. This step is generic and identical for all target devices. Next, we use hardware characteristics (number of cores, SIMD units per core and cache sizes) to generate specific mappings of loops onto compute resources. Depending on the used hardware, specialized hardware functionality (shared memory, approximate mathematical functions, OpenMP flags, ...) is used.

After all groups have been optimized and specialized implementations have been compiled, we generate a new network description for the framework, that is returned to the user as executable neural network object. This optimized network behaves identical to the original, but uses our optimized implementations.

3 ARCHITECTURE

The architecture (Figure 1) of SoL is modular, which makes it easily extendable. To interface with AI frameworks, we use frontends that consist of an interface that is responsible to translate the NN from the framework to SoL and to provide an optimized NN description to the user. Further a runtime component bridges framework specific functionality to SoL, e.g., memory (de-)allocation. Our optimizations perform different optimizations depending on the performance limiting type of the layers. The device backends need to implement these optimizations and employ the device specific optimizations, i.e., our x86 backend uses OpenMP and the ISPC [10] compiler for the I/O memory bound layers and the Intel MKL-DNN library other layers.

4 PRELIMINARY RESULTS

At the current development state, we can run prediction tasks on CPUs and GPUs, which are shown in Figure 2. These have been run on a server with 2x Intel E5-2637 v4, 128GB DDR4, NVIDIA GTX 1080 Ti, 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. Each test was run 20 times and the best result is shown. In the SoL case, all layers that are not optimized by the DFP method use the default PyTorch implementation. As PyTorch uses cuDNN by default, we do not show results for (SoL+DNN) on GPUs. We evaluated a series of networks for inference and batched prediction. We can see that it depends on the network structure, if SoL or DNN contribute the higher performance gain, e.g., SoL in the MobileNets and DNN in AlexNet and the VGGs. Overall we achieve a peak improvement of 11.8x for inference (see poster), 8.0x for batched-prediction (128) on CPUs and 1.7x and 2.3x respectively on GPUs.

5 STATUS AND FUTURE WORK

For our next milestone we plan to complete the support for PyTorch, TensorFlow and CNTK as frontends; x86 CPUs, NVIDIA GPUs and NEC Aurora as backends; applying the previously mentioned optimizations, for prediction and training! Further, we plan to add more optimization features, especially targeting other types of network layers, e.g., recurrent neural networks.

REFERENCES

[1] ARM. ARM Project Trillium. arm.com/products/processors/machine-learning.
[2] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. *CoRR* abs/1802.04799 (2018). arxiv.org/abs/1802.04799.
[3] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. 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] Intel. 2018. Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning. *arXiv* (2018). arxiv.org/abs/1801.08058.
[12] Khronos. Neural Network Exchange Format. khronos.org/nnef.
[13] Microsoft. Cognitive Toolkit. microsoft.com/en-us/cognitive-toolkit.
[14] NVIDIA. Tensor Cores in NVIDIA Volta. nvidia.com/en-us/data-center/tensorcore.
[15] NVIDIA. TensorRT. developer.nvidia.com/tensorrt.
[16] Power VR. PowerVR Series2NX. imgtec.com/powervr-2nx-neural-network-accelerator.
[17] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. (2018). research.fb.com/announcing-tensor-comprehensions/.
[18] Vertex.ai. 2017. PlaidML: Open Source Deep Learning for Every Platform. vertex.ai/blog/announcing-plaidml.
[19] Nicolas Weber, Florian Schmidt, Mathias Niepert, and Felipe Huici. 2018. BrainSlug: Transparent Acceleration of Deep Learning Through Depth-First Parallelism. arxiv.org/abs/1804.08378. *arXiv* (2018).