Intel® nGraph™

An Intermediate Representation, Compiler, and Executor for Deep Learning

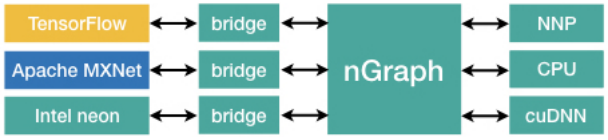
Scott Cyphers
Arjun K. Bansal
Anahita Bhiwandiwalla
Jayaram Bobba
Matthew Brookhart
Avijit Chakraborty
Will Constable
scott.cyphers@intel.com
arjun.bansal@intel.com
anahita.bhiwandiwalla@intel.com
jayaram.bobba@intel.com
matthew.i.brookhart@intel.com
avijit.chakraborty@intel.com
will.h.constable@intel.com
Intel

Christian Convey
Leona Cook
Omar Kanawi
Robert Kimball
Jason Knight
Nikolay Korovaiko
Varun Kumar
christian.convey@intel.com
leona.cook@intel.com
omar.kanawi@intel.com
robert.kimball@intel.com
jason.knight@intel.com
nikolay.korovaiko@intel.com
varun.v.kumar@intel.com
Intel

Yixing Lao
Christopher R. Lishka
Jaikrishnan Menon
Jennifer Myers
Sandeep Aswath Narayana
Adam Procter
Tristan J. Webb
yixing.lao@intel.com
christopher.r.lishka@intel.com
jaikrishnan.menon@intel.com
jennifer.myers@intel.com
sandeep.aswath.narayana@intel.com
adam.m.procter@intel.com
tristan.webb@intel.com
Intel

ABSTRACT

The Deep Learning (DL) community sees many novel topologies published each year. Achieving high performance on each new topology remains challenging, as each requires some level of manual effort. This issue is compounded by the proliferation of frameworks and hardware platforms. The current approach, which we call "direct optimization", requires deep changes within each framework to improve the training performance for each hardware backend (CPUs, GPUs, FPGAs, ASICs) and requires $O(fp)$ effort; where f is the number of frameworks and p is the number of platforms. While optimized kernels for deep-learning primitives are provided via libraries like Intel® Math Kernel Library for Deep Neural Networks (MKL-DNN), there are several compiler-inspired ways in which performance can be further optimized. Building on our experience creating neon (a fast deep learning library on GPUs), we developed Intel nGraph, a soon to be open-sourced C++ library to simplify the realization of optimized deep learning performance across frameworks and hardware platforms. Initially-supported frameworks include TensorFlow, MXNet, and Intel® neon framework. Initial backends are Intel Architecture CPUs (CPU), the Intel® Nervana Neural Network Processor™ (NNP), and NVIDIA GPUs. Currently supported compiler optimizations include efficient memory management and data layout abstraction. In this paper, we describe our overall architecture and its core components. In the future, we envision extending nGraph API support to a wider range of frameworks, hardware (including FPGAs and ASICs), and compiler optimizations (training versus inference optimizations, multi-node and multi-device scaling via efficient sub-graph partitioning, and HW-specific compounding of operations).



1 INTRODUCTION

Deep learning frameworks are libraries that provide domain-specific languages for defining deep learning computations and APIs for managing data and executing computations. Backends' implementations are encapsulated so that the same computation can execute on multiple backends, such as CPUs and GPUs. We adopt the organization of compilers such as LLVM[7] by converting framework-specific computation definitions into a framework-independent intermediate representation (IR) that we compile into a form that can execute on the backend. An nGraph *framework bridge* acts as a framework backend. Each nGraph backend has a *transformer* that compiles or interprets the IR and provides an allocation and execution API that the framework bridges use to implement the framework's API. A key difference between compilers for languages like C++ and compilers for deep learning frameworks is that with deep learning, the data being operated on is large and variable-sized, but highly amenable to parallelization.

1.1 Related Work

Google's accelerated linear algebra (XLA)[6] compiler acts as an experimental backend for TensorFlow[4]. Unlike nGraph, the XLA project has made no public comments about support of frameworks other than TensorFlow.

SYSML 2018, Stanford University

DOI:

The DMLC group announced the NNVM[8] project as a lightweight graph optimization library for deep learning and later announced TVM[5] as an ahead-of-time compiler that supports multiple hardware platforms and interoperates with NNVM. NNVM leaves operator set unspecified, which makes different frontends and backends incompatible. nGraph, XLA, and LLVM use a fixed, but extensible, IR operation set.

DLVM [9] is a project of the University of Illinois Urbana-Champaign. DLVM proposes an LLVM inspired modular IR with full control flow and a side-effect free representation. It remains to be seen if the more flexible IR is capable of supporting the performance optimizations enabled by simpler data flow graph IRs like those of nGraph, XLA, and NNVM.

ONNX [2] is a recent cross-industry effort, which we participate in, to standardize an IR for inference. The nGraph IR has a richer feature set, including support for training and a rich set of optimization passes and backends for execution. We will aim for ONNX interoperability.

The activity in the space of deep learning compilers and IRs highlights their need and we look forward to a healthy exchange of ideas as the field moves forward on these complementary efforts. We believe nGraph can interoperate with developing standards through framework bridges and nGraph backends.

2 INTERMEDIATE REPRESENTATION

An nGraph IR is a directed acyclic graph of stateless operation nodes. Each node has zero or more inputs and zero or more outputs. Nodes may have additional constant attributes that affect their behavior, such as which axes to sum over. The inputs and attributes of a node determine the shape and element types of the outputs.

Nodes operate on multi-dimensional arrays, called tensors. Most frameworks associate semantics with particular axis positions. For example, images might be stored in tensors ordered by mini-batch size, channels, height and width. The framework-specific ordering is usually a reflection of op implementation and tensor element layout. When dealing with video or high-dimensional time series datasets, rank restrictions from framework ops require explicit tensor reshaping and axis reordering. With the exception of tensors directly accessible to framework users, nGraph, *does not* have a fixed relationship between axis order and tensor element layout.

3 FRAMEWORK BRIDGES

Frameworks use a framework-specific symbolic representation of their computations, called a *computational graph*. Backends use the graph to interpret or compile computations. The graph is also used in the implementation of some form of the autodiff algorithm for the computation of derivatives, either by computing the derivative directly on the graph, or by computing the graph for a derivative computation from an existing graph. Framework bridges belonging to the nGraph library use the graph to construct the nGraph IR.

Apache MXNet[1] is a core C++ library and a C API for interacting with several frontend languages. Machine learning models may be defined imperatively through Gluon or symbolically through the standard MXNet frontend. Models' operations are represented as nodes in the NNVM graph. The MXNet-nGraph bridge translates the NNVM inference graph into nGraph IR; it selects the largest possible computation for the respective backend and uses autodiff on the nGraph IR for the derivative. Compiled nGraph functions can then interface with the standard MXNet execution engines.

TensorFlow's[4] XLA framework enables compilation and execution of TensorFlow graphs on novel hardware such as NNP. During the execution of a computation via XLA, an HLO IR of the TensorFlow computation is sent to a device such as a CPU, GPU or novel hardware. Our bridge plugin registers itself as a new XLA device, maps HLO IR to nGraph IR, and returns a compiled function. During the execution of the TensorFlow computation, the function is invoked by TensorFlow on the input data and the resulting nGraph output is returned.

For neon, we are creating a Python binding for the nGraph API, which we hope to also use with other Python-based frameworks.

4 TRANSFORMERS

The IR generated by the nGraph library is passed to a transformer for the generation of code optimized specifically for the selected backend. These newly-optimized backends provide facilities for pattern matching, liveness analysis, memory management, and the combining of tensor-element layout and shape management with backend kernel selection.

The CPU transformer makes use of MKL-DNN, which produces optimized sequences of calls to highly optimized kernels. Optimizations provided by MKL-DNN are at a finer granularity than those provided by nGraph. The CPU transformer will also be used by other transformers for sub-graphs that use operations not supported by their backend.

Intel's NNP processor is tailored for deep learning workloads. Its transformer lets us make the fullest use of the hardware, falling back on the CPU transformer for unsupported operations.

The cuDNN[3] transformer dynamically generates code that links to the NVIDIA CUDA Deep Neural Network (cuDNN) library for common kernels, such as convolution or softmax. The nGraph library then compiles portions of the graph into LLVM IR representation, and uses the PTX-emitting backend of LLVM to generate the GPU assembly language PTX.

nGraph will natively support collective communication primitives (AllReduce, Gather, Broadcast), as well as point-to-point primitives as core graph ops. Transformers will generate the corresponding communication library calls. Transformers will support vanilla MPI, or provide optimized communication methods that impose restrictions. An example of such a restriction would be: operate only across a homogeneous cluster of a certain topology.

REFERENCES

[1] 2017. Apache MXNet. (2017). Retrieved January 4, 2018 from https://mxnet.apache.org/
[2] 2017. ONNX. (2017). Retrieved January 4, 2018 from http://onnx.ai
[3] 2018. NVIDIA cuDNN. (2018). Retrieved January 4, 2018 from https://developer.nvidia.com/cudnn
[4] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng.

2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). https://www.tensorflow.org/ Software available from tensorflow.org.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, and Haichen Shen. 2017. TVM: An End to End IR Stack for Deploying Deep Learning Workloads on Hardware Platforms. (Aug 2017). http://tvmlang.org/2017/08/17/tvm-release-announcement.html

[6] Google. 2017. XLA Overview. (2017). Retrieved January 5, 2018 from https://www.tensorflow.org/performance/xla/

[7] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California.

[8] Amazon Web Service AI team. 2017. (Oct 2017). http://tvmlang.org/2017/10/06/nnvm-compiler-announcement.html

[9] Richard Wei, Vikram S. Adve, and Lane Schwartz. 2017. DLVM: A modern compiler infrastructure for deep learning systems. *CoRR* abs/1711.03016 (2017). arXiv:1711.03016 http://arxiv.org/abs/1711.03016