Accelerating Recurrent Neural Networks through Compiler Techniques and Quantization

Li-Wen Chang, Yang Chen, Wenlei Bao, Amit Agarwal, Eldar Akchurin, Ke Deng, Emad Barsoum

Microsoft

{LiWen.Chang, yanchen, Wenlei.Bao, amitaga, eldak, kedeng, ebarsoum}@microsoft.com

Abstract

Deep learning inference lies in the core of trained neural networks. The performance of the inference engine can determine the success of trained neural network services. Deep learning inference often runs in data centers or at edge devices, which require quick service response time and the flexibility of supporting a variety of hardware targets. Recently, a number of machine learning frameworks have emerged to apply compiler and quantization techniques to accelerate inference engines and showed great effectiveness. In these frameworks, generic optimizations such as graph partitioning and operator fusion are applicable to all hardware targets, whereas target-dependent optimizations and code generation remain isolated and can be easily plugged into the frameworks. However, none of these existing frameworks are specialized to optimize recurrent inference. The low latency requirement of recurrent applications such as speech recognition and natural language processing (NLP) and the common long-tail dependency aspect of recurrent models impose interesting challenges in optimizing such models for inference. In this paper, we present a high-performance inference engine, ModelCompiler, which utilizes various compiler techniques as well as quantizations to accelerate recurrent models, and show its extensibility of targeting diverse hardware. ModelCompiler is used for runtime inference of several large recurrent models in Microsoft production services, which require strict real-time latency constraints. Our experiments showed that ModelCompiler can achieve up to 6.24X speedups over CNTK for real production models.

1 Introduction

Deep learning inference is critical to the success of trained neural network services, which often run in cloud platforms such as Azure, AWS and Google Cloud, or at edge devices such as smartphones and autonomous vehicle devices. A well-designed inference engine needs to meet a number of criteria. First of all, inference must be fast. Fast inference reduces service response time and thus improves user experience. Reduced service response time can increase the throughput of the AI service running in the cloud and improve its scalability to serve millions of users. Furthermore, the ability to quickly produce answers is important to time-critical tasks such as image or speech recognition in a running autonomous vehicle. Second, the inference engine should be flexible enough to support a variety of hardware targets. The machine learning industry has been exploring a wide range of AI hardware solutions, not only GPUs or CPUs, but including ASICs such as Google TPUs (Tensor Processing Unit) [1], Graphcore IPU (Intelligence Processing Unit) [2], Habana AIP (AI Processor) [3] and Microsoft HPU (Holographic Processing Unit) [4] and FPGAs such as Microsoft Brainwave [5, 6]. Consequently, the inference engine needs to be easily extended to support emerging targets and ready to run in a heterogeneous environment.

Recently, a number of machine learning frameworks [7, 8, 9, 10, 11, 12, 13] have emerged to utilize compiler, high-performance computing, and quantization techniques to accelerate inference and work with the diversity of underlying hardware. They demonstrate the effectiveness of compiler techniques and quantization in optimizing neural networks. These frameworks often use AST-like (abstract syntax tree) intermediate representations (IRs) for graph optimization and code generation. Common graph-level optimizations such as operator fusion and constant folding can be separated from target-dependent ones. Moreover, existing compiler frameworks such as LLVM [14] can also be integrated to leverage the ability for re-targeting different hardware and optimizing code generation.

Most current compiler-based machine learning frameworks demonstrate their achievements through vision-based neural networks. Thus far, none of these frameworks are specialized to optimize for recurrent neural network inference. Typical recurrent model inferencing requires low latency to make the service possible on real products. Furthermore, recurrent models often suffers long-tail dependency because of the loop-dependent structures, which make it difficult to perform parallelization. These characteristics and requirements impose interesting challenges in optimizing such models for inference.

In this paper, we present ModelCompiler, a high-performance inference engine that exploits compiler and quantization techniques to accelerate inference and shows extensibility of various hardware targets. We further customize ModelCompiler to serve recurrent models for an x64 CPU with a number of optimizations and quantization techniques.

2 Related Work

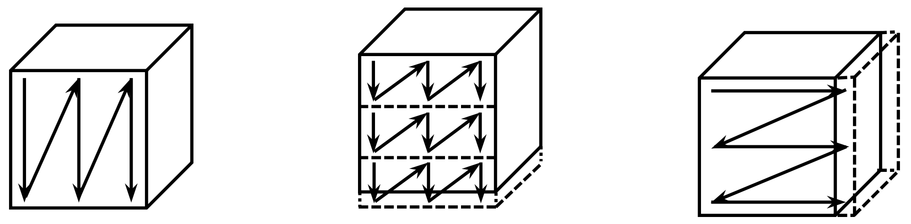
XLA (Accelerated linear algebra) [13] acts as a compiler backend for TensorFlow [15], performing optimizations and generates target-specific machine instructions. TVM [12] is an end-to-end deep learning stack that compiles deep learning models for different hardware targets. It uses Halide [16] IR for lowering and arithmetic optimizations. Tensor Comprehensions [10] applies a similar approach like TVM, and also uses Halide IR. Glow [8] is a graph-lowering compiler, which lowers a dataflow graph into two-level IRs. Glow separates high-level graph optimizations from low-level machine-dependent optimizations for retargeting diverse hardware. ELL [7] is very similar to Glow, but specifically designed for resource-constraint environments. Particularly, XLA, TVM and Glow support quantizations with various bit-widths.

Besides the compiler-inspired frameworks described above, commonly-used deep learning primitives are implemented as optimized kernels and provided via libraries such as cuDNN [17], Eigen [18] and MKL-DNN [19]. These kernels are target-specific and their performance is well-tuned. Most of these libraries support integer kernels for quantization. On top of libraries, nGraph [9] and TensorRT [11] apply certain compiler techniques that transform the computation graph to call underneath libraries.

3 Design Overview

ModelCompiler has two levels of IRs, a high-level Microsoft Cognitive Toolkit (CNTK) IR [20] and a low-level Halide IR [16]. The high-level CNTK IR is a dataflow node-based graph representation that contains domain-specific attributes for nodes. Its graph representation allows ModelCompiler to perform graph-level optimizations for neural networks. The low-level Halide IR is an AST with rich utilities for constructing optimizations and index iterations. This low-level IR enables ModelCompiler to utilize the Halide infrastructure for common loop transformations without writing LLVM lowering passes. Moreover, it also provides the ability to re-target different hardware systems.

During the inference process, ModelCompiler first builds a CNTK IR from a CNTK model, and then performs various graph optimizations such as constant folding and quantization to transform the model into the most compact form. It further partitions the transformed model into sub-graphs called execution blocks. Each execution block is then lowered to a Halide IR, where optimizations such as tiling, fusion, unrolling, vectorization and data layout transformations are all realized.When the quantization decision is given, the relevant quantization optimization and code generation also happen at the low-level optimization stage. Furthermore, external function calls are provided to facilitate existing efforts from third parties. After generating all execution blocks, ModelCompiler produces an executor for the runtime based on the execution block graph. Our executor can run in parallel or pipeline mode.



(a) Native Layout (b) Tiled Layout for $k = 0$ (c) Permuted-and-Padded Layout for $k = 1$

Figure 1: Data Layout Examples

4 Compiler Optimization

In ModelCompiler, optimizations are categorized to high- and low-level based on the IRs that they depend on. High-level optimizations take place at the CNTK IR level (i.e. high-level) and typically require to collect metadata from the model, which include shape information, node reference counts, tensor data types, node types and attributes. Low-level optimizations, which occur at the Halide IR level, require the metadata collected from both of the high- and low-level IRs. The metadata from the Halide IR typically contain data types and loop structures.

4.1 High-level Optimizations

Graph Optimizations in ModelCompiler include dead node elimination, inference-irrelevant node elimination, and constant folding. We also perform quantization-related graph transformations at this level. More details of quantization are described in Sec 5.

Graph optimizers within ModelCompiler are capable of serializing a transformed model back into a new CNTK-format model. This capability makes ModelCompiler be able to serve as an offline graph optimizer or quantizer. After graph transformation, ModelCompiler starts to analyze the transformed graph and collects necessary metadata for performance-critical decisions later, such as data layouts and execution plans.

Data Layouts are determined in the high-level IR by the tensor's shapes and data types, and the node's types and attributes. ModelCompiler currently supports three categories of data layouts: *native*, *tiled*, and *permuted-and-padded*. The tiled layout tiles a specific dimension, k, to the lowest dimension and moves the number of tiles to the highest dimension, while the permuted-and-padded layout moves a specific dimension, k, to the lowest dimension and pads to a certain size. In our context, we follow the CNTK C's notation that the lowest dimension locates at consecutive addresses in memory. For a tensor with multiple dimensions, there might be multiple tiled and permuted-and-padded layouts. Figure 1 illustrates one example for each layout. Moreover, tiling or padding sizes of the last two layouts guarantee alignment for vectorization and cache lines. Data marshaling for layout transformation of constant tensors such as weights occurs at compile-time. Because data layouts underneath a tensor become manageable, some layout marshaling nodes, such as Transpose, Pad, or transpose attribute of MatMul, could also be eliminated with proper layouts.

Execution Plans define how to iterate dynamic axes within one graph or sub-graph. In our context, dynamic axes are sequence and batch iterations, whose length can vary and is typically unknown before runtime. ModelCompiler has a centralized decision making mechanism to determine an execution plan, which can impact graph partition, data layouts, and executors in runtime. For each dynamic axis, ModelCompiler can have *sequential*, *parallel*, and *pipeline* execution. Figure 2 and 3 illustrate the looping behavior for each execution. In the figures, $F1$ and $F2$ are two fused functions, each of which is a sub-graph containing multiple nodes. In particular, parallel execution is only allowed in a dynamic axis without any dependency, such as a batch axis or a sequence axis in non-recurrent nodes. The parallel loop in parallel execution can potentially map to a batched execution. For example, multiple `gemv`'s can be grouped into a single `gemm`. Meanwhile, our pipeline execution can potentially have good data reuse since the weight of each node can be kept in its private cache. This pipeline design is similar to [21], but can be considered as an optimized version for shared memory model by replacing the message between two workers with an address rather than a state itself. The `Release` and `Acquire` pair can be implemented through lightweight task queues or callback mechanisms. ModelCompiler uses Intel TBB [22] task graphs. In ModelCompiler, an

```
for (i=0:N)
  F1(i)

for (i=0:N)
  F2(i)
```

(a) Sequential

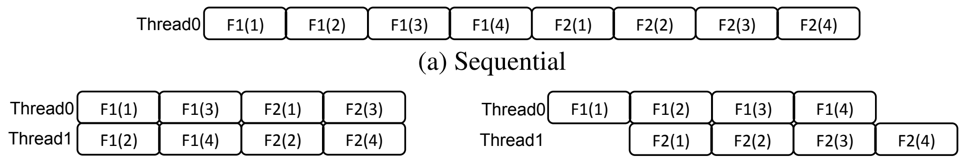
```
parallel_for (i=0:N)
  F1(i)

parallel_for (i=0:N)
  F2(i)
```

(b) Parallel

```
for (i=0:N) //Thread0
  F1(i)
  Release(i)
for (i=0:N) //Thread1
  Acquire(i)
  F2(i)
```

(c) Pipeline

Figure 2: Pseudo Code of Loop Pattern in Execution Plans for Two Fused Functions (Sub-graphs)



(a) Sequential

(b) Parallel

(c) Pipeline

Figure 3: Illustration of Loop Patterns in Execution Plans for Two Fused Functions (Sub-graphs)

execution plan is determined by a profiling heuristic. Pipeline execution often takes place in a model with multi-layer LSTMs (Long short-term memory), parallel execution can run in a sub-graph without any independent dynamic axis, and sequential execution is mainly for debug purpose.

Graph Partitioner splits a transformed graph into several sub-graphs called execution blocks. In ModelCompiler, each execution block has its own execution plan, but all execution plans are determined together without contradicting each other. Graph partition is driven by the size of the partition, reference counts of tensors, total weights of tensors, types of nodes, recurrent structure of nodes and profiling data. There are several guidelines in designing our graph partitioner. First, the partitioner avoids too small partitions, which could cause too much function call overhead. Second, we calculate the computation cost for each node by estimating the weights of tensors and the types of nodes based on profiling, and make sure the total computation cost of nodes in each partition below a predefined threshold to balance partitions. Third, the partitioner only puts nodes with compatible loop structures into the same partition. Nodes with different sequence axes are placed into different partitions. Forth, the partitioner prefers not to split a recurrent structure unless the structure is too large, to avoid increasing the complexity and overhead for executors at runtime.

After graph partitioning, ModelCompiler lowers each partition with its own execution plan to a Halide IR, and then generates an executable function for each partition by applying various low-level optimizations. After code generation of all partitions, ModelCompiler constructs an executor based on the dependencies among partitions and execution plans.

4.2 Low-level Optimization

ModelCompiler's low-level optimizations are built on top of the Halide infrastructure, which provides rich utilities for constructing basic loop transformations. How to apply those transformation utilities impacts inference performance.

Loop Transformations in ModelCompiler, such as fusion, vectorization, unrolling and tiling, are controlled by both node-level metadata collected previously and low-level metadata collected in the Halide IR. Those node-level metadata allow ModelCompiler to optimize loop structures without losing information from tensors or nodes. Our loop fusion relies on tensor reference counts and node types. For example, a group of element-wise nodes with single reference are simply fused with preceding or succeeding nodes. Tensor reference counts are used to avoid redundant computation during fusion. For instance, if an output tensor of a node A is consumed by two nodes B and C, ModelCompiler currently avoids to fuse A with B or C. Vectorization is typically applicable with predication and alignment guaranteed by data layout transformation. Unrolling and tiling is mainly controlled by the tensor's data type, loop body, and a heuristic threshold for each given architecture. ModelCompiler also allows to bind a specific optimization with a specific node type or a node satisfying specific conditions so that a sophisticated loop transformation can be easily adapted.

Data Layout Transformation in ModelCompiler is applied by introducing a tensor container that converts a memory access on the native layout to the corresponding access in the real data layout. This index calculation can be further optimized by index simplification. This design allows ModelCompiler to decouple data layout transformation from CNTK-to-Halide lowering, and further simplifies our low-level optimizers.

Shape Manipulation and Data Marshaling in ModelCompiler include Reshape, Pad, Crop, Transpose, Slice and Concat. As mentioned in Sec 4.1, some of these nodes could be eliminated by high-level optimizations. The remaining nodes can be either physically computed or simplified through fusion and index simplification . For example, Pad can either physically be computed as moving data to a new address, or be simplified as an *inline condition* fused with its consumer nodes. ModelCompiler prefers the latter unless it is suppressed by another node-specific optimization.

External Function Calls, such as library calls using Eigen [18] and MKL [19], are allowed in ModelCompiler. This strategy has an advantage of directly utilizing existing libraries optimized by third-party experts without building a compiler with sophisticated optimizations. However, one disadvantage is that existing libraries could be lack of necessary information about specific nodes or sub-graphs to enable proper optimizations. In the following paper, particularly for the results in Sec 7, we *disable* any external library, such as MKL, in order to demonstrate robustness of our compiler optimizations. We only use a set of pre-built custom instructions, which is also generated offline by us, for integer BLAS working on certain bit-widths and tensor shapes.

5 Quantization

ModelCompiler supports multiple linear quantization schema and both *symmetric* and *asymmetric* quantization methods [23]. Furthermore, ModelCompiler has multiple granularity levels for quantization, *whole-tensor*-based, *column*-based, *row*-based, and *channel*-based ones. Fine-grained quantization typically generates multiple-dimension coefficients (e.g. scale). ModelCompiler allows arbitrary quantization bit-widths less than or equal to 16 bits, and arbitrary accumulation bit-widths less than or equal to 32 bits. To utilize native 8-, 16-, 32-bit arithmetic computation, we currently cast bit-widths less than 8 to 8 bits, bit-widths more than 8 but less than 16 to 16 bits, bit-widths more than 16 but less than 32 to 32 bit. The native code generation support for bit-widths less than 8 without casting is considered future work. ModelCompiler supports both *dynamic* and *static* quantization coefficient calculation. The term static implies that coefficients of quantization are determined offline, while dynamic means that coefficients are calculated through input tensors on the fly. Moreover, ModelCompiler allows more than one kinds of quantization methods, bit-widths, granularity levels, and coefficient calculation applied in different nodes within a single model to obtain the best performance without sacrificing final accuracy.

5.1 Optimizations for Quantization

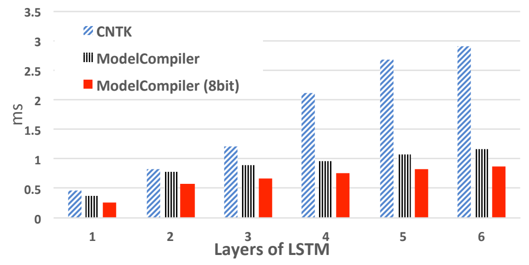
Quantization in code generation changes data types and the sizes of tensors. Changes to data types can affect preferred data layouts, tiling or padding sizes of data layouts, widths of vectorization, sizes of tiling and unrolling transformation and external library calls. The reduction of the size of all tensors can also change graph partition. However, most optimizations described in Sec 4 are generic and applicable regardless of the changes caused by quantization.

Although ModelCompiler casts quantization and accumulation bit-widths to 8, 16, or 32 bits to utilize native arithmetic computation, given a specific IR, it still tracks its original effective bit-width to use a minimal bit-width for an intermediate result without causing *additional*[1] overflow or underflow. For example, $c = a * b + c$, where a, b, and c are 4-bit integers, is converted into a multiplication of two 8-bit integers to an 8-bit saturated output and an add of two 8-bit integers to a 16-bit integer output, instead of a multiplication of two 8-bit integers to a 16-bit output and an add with two 16-bit integers to a 32-bit integer output.

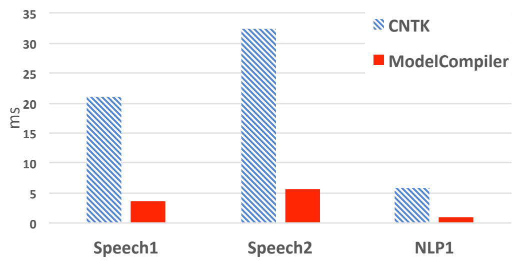
5.2 Debugging Numerical Computation for Quantization

Since quantization could lose precision and potentially have overflow or underflow with an improper accumulation bit-width, ModelCompiler has a special mode for debugging numerical computation in

[1]Since the accumulation bit-width is specified by the model, overflow or underflow could still occur when the result exceeds the representation of the accumulation bit-width.



(a) Multiple-layered Stack LSTMs



(b) Production Models

Figure 4: Performance Comparison between CNTK and ModelCompiler

quantization. In this mode, ModelCompiler generates special code to detect and record overflow or underflow for each numerical expression with its original bit-width.

At a higher debug level, for each quantized node or sub-graph, ModelCompiler further generates two code variants for the same computation, one with full-precision computation and one with quantization, and records the difference of each tensor element for later analyses. Data scientists can use this information to adjust the quantization schema to deliver more accurate results.

6 Runtime Design

ModelCompiler generates a runtime executor based on execution plans and dependencies among execution blocks at compile-time as described in previous sections. It also introduces barriers to guarantee dependencies. Particularly, a barrier is inserted in between a pipeline execution and its following non-pipeline execution or pipeline execution with an incompatible dynamic axis, e.g. a different dynamic axis or the same dynamic axis with a different direction. Although the value of dynamic axis is unknown, most neural network formats such as CNTK or ONNX [24] support a symbolic representation for a dynamic axis. ModelCompiler uses this information to differentiate dynamic axes from each other. Each dynamic axis becomes a parameter in an executor, and the value of the dynamic axis is determined and passed as an argument of the executor at runtime.

7 Evaluation

We evaluated ModelCompiler by comparing performance with CNTK 2.6 on a machine with a 6-core Intel Xeon E5-1650V4 CPU, 16 GB RAM and Windows 10 OS. Both ModelCompiler and CNTK were compiled by Microsoft Visual Studio Enterprise 2017 15.8.4 and the performance numbers were obtained using Google benchmark [25]. Furthermore, CNTK used MKL, and ModelCompiler did not call any external library.

Figure 4 (a) compares ModelCompiler with CNTK using synthetic models of multiple-layered stack LSTMs, in which each LSTM layer has both input and output sizes of 128, with sequence length of 17 and batch size of 1. ModelCompiler demonstrated up to 3.37X and 2.51X speedups over CNTK with and without quantization, respectively. Moreover, ModelCompiler can scale with the number of layers.

In Figure 4 (b), we use real production models, a 4-layered customized LSTMs (`Speech1`) with more than 45-MByte parameters, a 6-layered customized LSTMs (`Speech2`) with near 65-MByte parameters, and a 4-layered bidirectional LSTMs (`NLP1`) with near 25-MByte parameters. All models have a sequence length of 16 and a batch size of 1, and are carefully quantized to achieve the same final accuracy. ModelCompiler delivered 5.81-6.24X speedups over CNTK for these models.

8 Conclusion

We presented ModelCompiler, a high-performance inference engine that exploits compiler and quantization techniques. ModeCompiler utilizes two-level IRs to apply various generic optimizations as well as specialized ones to solve the challenges for recurrent neural networks. Our evaluation on production models validated our inference framework, obtaining significant speedups over CNTK.

References

[1] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017.

[2] Simon Knowles. How to Build a Processor for Machine Intelligence (Part 2). `https://www.graphcore.ai/posts/how-to-build-a-processor-for-machine-intelligence-part-2`.

[3] Habana, AI processor. `https://habana.ai`.

[4] Second version of HoloLens HPU will incorporate AI coprocessor for implementing DNNs. `https://www.microsoft.com/en-us/research/blog/second-version-hololens-hpu-will-incorporate-ai-coprocessor-implementing-dnns/`.

[5] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Mahdi Ghandi, Daniel Lo, Steve Reinhardt, Shlomi Alkalay, Hari Angepat, Derek Chiou, Alessandro Forin, Doug Burger, Lisa Woods, Gabriel Weisz, Michael Haselman, and Dan Zhang. Serving DNNs in real time at datacenter scale with project brainwave. *IEEE Micro*, 38(2):8–20, March 2018.

[6] Jeremy Fowers et al. A configurable cloud-scale DNN processor for real-time ai. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 1–14, 2018.

[7] The embedded learning library (ELL). `https://github.com/Microsoft/ELL`.

[8] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

[9] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. Intel nGraph: An intermediate representation, compiler, and executor for deep learning. *arXiv preprint arXiv:1801.08058*, 2018.

[10] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

[11] NVIDIA TensorRT. `https://developer.nvidia.com/tensorrt`.

[12] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: End-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799*, 2018.

[13] XLA: Domain-specific compiler for linear algebra to optimizes tensorflow computations. `https://www.tensorflow.org/performance/xla`, 2008.

[14] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[15] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, pages 265–283, 2016.

[16] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[17] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[18] Eigen. `http://eigen.tuxfamily.org`.

[19] Intel MKL. `https://software.intel.com/en-us/mkl`.

[20] Frank Seide and Amit Agarwal. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2135–2135. ACM, 2016.

[21] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, Michael Isard, Manjunath Kudlur, Rajat Monga, Derek Murray, and Xiaoqiang Zheng. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference*, pages 18:1–18:15, 2018.

[22] Intel Threading Building Blocks. `https://software.intel.com/en-us/intel-tbb`.

[23] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *arXiv preprint arXiv:1712.05877*, 2017.

[24] Open Neural Network Exchange (ONNX). `https://onnx.ai/`.

[25] Google Benchmark. `https://github.com/google/benchmark`.