Towards Automatic Function Call Generation for Deep Learning

Shizhi Tang
tsz19@mails.tsinghua.edu.cn
Tsinghua University

Jidong Zhai
zhaijidong@tsinghua.edu.cn
Tsinghua University

ABSTRACT

Mainstream deep learning frameworks are commonly implemented by invoking underlying high performance tensor libraries on various architectures. However, as these libraries provide increasingly complex semantics including operator fusions, in-place operations, and various memory layouts, the gap between mathematical deep learning models and the underlying libraries becomes larger. In this paper, inspired by the classic problem of Instruction Selection, we design a theorem solver guided exhaustive search algorithm to select functions for complex tensor computations. Preliminary results with some micro-benchmarks and a real model show that our approach can outperform both Tensorflow and Tensor Comprehensions at run time.

CCS CONCEPTS

• **Computer systems organization** → Neural networks; • **Software and its engineering** → Retargetable compilers.

KEYWORDS

Tensor Computing, Tensor Compiler, Instruction Selection

ACM Reference Format:
Shizhi Tang and Jidong Zhai. 2019. Towards Automatic Function Call Generation for Deep Learning. In *Proceedings of Supercomputing '19 (SC'19).* ACM, New York, NY, USA, 3 pages.

1 INTRODUCTION

Currently, deep learning models are running on various architectures, including hardware accelerators such as GPUs, FPGAs, and ASICs, or end devices such as mobile phones and embedded devices. As a result, to mitigate the programming complexity of various underlying accelerators, typical programming frameworks, such as TensorFlow, Caffe and PyTorch, normally decouple the algorithm description and the computation of deep learning models. From a high-level view, a deep learning model is normally represented as a Directed Acyclic Graph (DAG) in these programming frameworks, where tensor operations are represented by nodes or edges. After specific compilation, tensor operations are transformed into underlying high performance tensor libraries such as Intel MKL-DNN, Eigen, NVIDIA cuBLAS and/or NVIDIA cuDNN.

However, as these libraries pursue higher performance, the semantics they provide are extraordinarily complex. Typical programming frameworks need to balance different factors to adapt to underlying tensor libraries, and sometimes additional tensor transformations are needed. Due to complex library features and numerous tensor operation types, existing frameworks usually incur large performance lost. This complex adaption also hinders the decoupling between frameworks and libraries. We identify three typical types for such complex semantics as follows:

In-place operations. For some tensor operations such as ReLU, computation can be directly applied to the input buffer, without copying the data to a separated output buffer, as long as the input tensor will not be used in the future. This saves memory allocation and data copying overhead. Libraries like cuDNN do provide in-place operations, but it is not trivial to determine whether an in-place operation can be applied for a specific model due to complex analysis over the DAG. A report from TensorFlow admitted that it is tricky to implement it correctly[1], while PyTorch just leaves this decision to end users[2].

Memory layouts. There are various ways to serialize a multi-dimensional tensor to a 1-D memory buffer, which are called different memory layouts. Three typical categories of layouts are listed below. (1) *Transposed layouts.* A typical batched multi-channel image in CNN can use either NCHW or NHWC layout[3], and cuDNN prefers the former. Based on previous studies, performance for different layouts are not the same[4]. (2) *Lazy computations enabling layouts.* When performing a broadcasting followed by an addition like in `cudnnAddTensor`, the broadcasting can be simply viewed as changing the layout of a specific tensor[5], so that the addition can be performed directly without an explicit broadcast. (3) *Complex optimized layouts.* Some libraries support blocked layouts, such as "NC/32HW32" layout in cuDNN, which is friendly to cache locality and vector instructions [4]. Different memory layouts represent different performance on various architectures. It remains an open problem to find an optimal layout for complex tensor operations on a given platform.

Fused operations. Fusing the computation of multiple tensor operations into one single function call can significantly improve the performance, especially for GPUs [6], but may incur more restrictions. For instance, cuDNN provides `cudnnConvolutionBiasActivationForward` to compute convolution, addition, and activation together, but with a limited convolution algorithm choice. Therefore, we need to balance the benefit of fusion.

Therefore, an automatic and effective approach to transform the DAG into function calls of the underlying libraries is increasingly important. In a nutshell, this problem can be defined as converting a DAG into a sequence of function calls, while meeting some constraints such as memory layout and data type of the functions, and keeping as high overall performance as possible. We consider this

SC'19, November 17-22, 2019, Denver, CO, USA
2019.

[1] https://ai.google/research/pubs/pub48051
[2] See the PyTorch API at https://pytorch.org/docs/stable/index.html. All functions that end with "_" are in-place.
[3] NCHW means the tensor is saved as a 4-D row-major array where the 4 dimensions represent samples, channels, heights and weights respectively. NHCW is alike.
[4] Convolution algorithms on different layouts are different in cuDNN. See the API (https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html) for details.
[5] These special layouts can be modeled with the "stride" vector, which is implemented in most frameworks.

problem a variant of the traditional instruction selection problem in compilers, which covers a DAG representing the Intermediate Representation (IR) with instructions. However, optimal DAG covering is an NP-complete problem, and traditional compilers often fall back to macro expansion or tree covering, or to use sub-optimal DAG covering algorithms [2]. In this paper, we demonstrate our preliminary results using an exhaustive search approach, guided by Z3 [5] theorem solver, which trades DAG covering speed for better final performance.

2 RELATED WORK

Several previous studies including TVM [3], PlaidML[6], TensorFlow XLA[7] and Tensor Comprehensions [8] have witnessed the difficulties of converting descriptions of deep learning models to low-level executables. These studies try to compile models or operations directly to underlying hardware instructions, to bypass tensor libraries in between. However, as for the best-optimized operations in common libraries such as matrix multiplication and convolution, which are also the most commonly used operations, these compiling approaches still fall behind in performance [1].

3 ALGORITHM

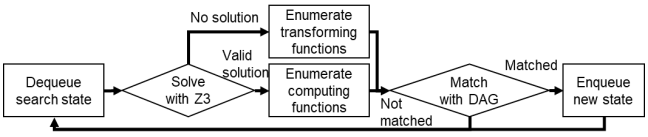

Figure 1: High-level work flow of our approach.

Figure 1 is an overview of our approach. To perform a search, we must firstly be able to check the constraints between functions. These constraints include data location, memory layout, data type, etc. Some constraints may be complex. For example, Level 1 functions in BLAS accept vector with strides other than 1, i.e., two adjacent items in the vector may not be adjacent in the memory, BUT it requires strides not equal to 0. Therefore, we turn to logic equations because it is the most general form. We set every property of the tensors as an unknown predicate, and use a first-order logic (in)equation set to describe the constraint of each function in the libraries. Given a sequence of function calls, if the unknown predicates can be solved using Z3, the constraints must be satisfied. E.g., the constraint above may be represented as $STRIDE(INPUT(f, 0), 0) \neq 0$, meaning the stride of the 0-th dimension of the 0-th input of function f cannot be 0.

Given the library description including the constraints, and a DAG to describe the deep learning model, we can search for the call sequence, where each call matches part of the DAG. We divided all functions into two categories, *computing functions* and *transforming functions*, where the former actually performs operations, while the latter only transforms the tensor from one layout to another. Suppose we only need transforming functions when a computing function does not fit, in each iteration, we pick a search state with the lowest cost, and try to add computing functions into the sequence, followed by a constraint checking. If the checking fails, we then try to add transforming functions. Finally, the sequence that covers the entire DAG will be the solution. However, the search process is very time-consuming, so we divide the DAG into sub-graphs before searching, although the solution is sub-optimal.

[6]https://ai.intel.com/plaidml
[7]https://www.tensorflow.org/xla

4 EVALUATION

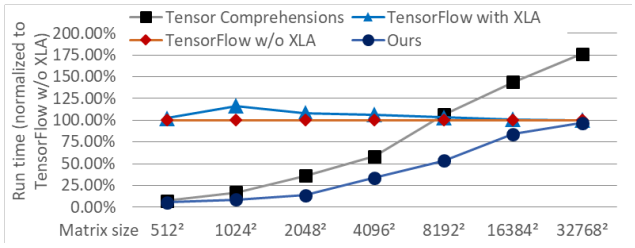

Figure 2: Fully Connected Layers normalized performance

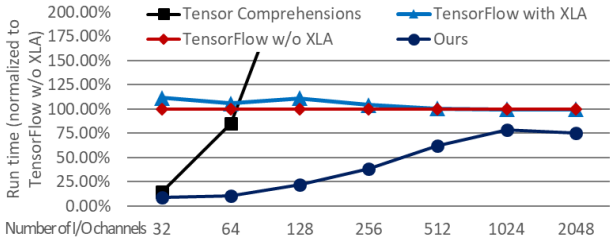

Figure 3: Convolution Layers normalized performance

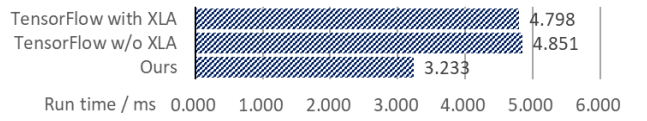

Figure 4: MobileNets performance

We measured the performance on Fully Connected Layers[8] (Figure 2) and Convolution Layers[9] (Figure 3) as micro-benchmarks and a real model, called MobileNets [7] (Figure 4[10]). We compare our approach with TensorFlow (including its XLA accelerated version[11]) as a mainstream framework and Tensor Comprehensions [8] as a compile-to-device approach. Batch sizes in all experiments are 1.

Results show that our approach outperforms other frameworks in experiments. Performance gains in small micro-benchmarks are more significant, which means we can successfully reduce the overhead outside libraries, instead of optimizing libraries themselves. To make our algorithm applicable to real industrial frameworks, further optimization to lower the call sequence generating time will be our future task.

[8]Consisting of a matrix multiplication, a bias and a ReLU.
[9]Consisting of a convolution, a bias and a ReLU. Input heights and widths fixed to 56.
[10]No data for Tensor Comprehensions because it cannot compute padded convolutions.
[11]XLA works in JIT mode.

REFERENCES

[1] Paul Barham and Michael Isard. 2019. Machine Learning Systems Are Stuck in a Rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*. ACM, New York, NY, USA, 177–183. https://doi.org/10.1145/3317550.3321441

[2] Gabriel Hjort Blindell. 2016. *Instruction selection: principles, methods, and applications*. Springer.

[3] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI 18*. USENIX Association, 578–594.

[4] Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey. 2016. Distributed deep learning using synchronous stochastic gradient descent. *arXiv preprint arXiv:1602.06709* (2016).

[5] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *TACAS 2008*. Springer, New York City, USA, 337–340.

[6] Jiří Filipovič, Matúš Madzin, Jan Fousek, and Luděk Matyska. 2015. Optimizing CUDA code by kernel fusion: application on BLAS. *The Journal of Supercomputing* 71, 10 (2015), 3934–3957.

[7] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[8] 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. *arXiv preprint arXiv:1802.04730* (2018).