Backwards Operator Fusion in TVM

Andrew McMullin
University of Michigan
mcmullin@umich.edu

Morgan Ruffner
University of Michigan
mruffner@umich.edu

Yuan Zeng
University of Michigan
zengyuan@umich.edu

1. Introduction

TVM is an open source machine learning compiler framework intended for compiling multiple different framework models on any type of machine, and optimizing for that machine. One of the optimizations which TVM employs is operation fusion. Operator fusion combines multiple operations into a single kernel operation without saving the intermediate results in memory, reducing the number of operations being executed by the machine. Typically, the fusion process is done in a forward direction. The goal of our project was to reverse this process and implement backward operator fusion in TVM, and subsequently compare the results obtained by both methods. The purpose of this report is to summarize the methods, challenges, and results of our project and discuss the viability of backward operator fusion as an alternative to forward operator fusion.

2. Related Works

In the paper "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning", the authors introduce TVM as a compiler that solves optimization challenges specific to deep learning (2).

The key challenges identified by the authors are leveraging specific hardware features and abstractions and the large search space for optimization, which TVM addresses by introducing a tensor expression language, an automated program optimization framework, and a graph rewriter (2).

TVM makes use of a computational graph representation to apply high-level optimizations, in which a node represents an operation on tensors or program inputs, and edges represent data dependencies between operations (2).

Some of the graph-level optimizations employed include constant-folding, data layout transformations, a static memory planning pass, and operator fusion, which we chose as the focus for our project (2).

We used the Tensorflow documentation as a resource to understand the purpose of operator fusion and the steps taken to implement it (1). The main benefit of operator fusion is that fused operations maximize the performance of their underlying kernel implementations, by optimizing the overall computation and reducing memory footprint (1).

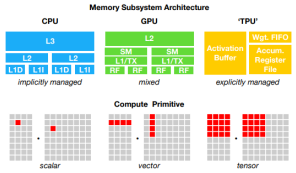
However, fusing operations can be challenging due to the fact that it can be very challenging to identify (e.g. via pattern matching) the sub-graph corresponding a composite operation, which may be represented as a set of primitive operations without a well defined boundary (1).

We utilized the TVM documentation in order to gain an understanding of how to install TVM and use it to compile and run a model (3).

3. Overview of TVM

TVM was motivated by the need to solve the growing demand to use deep learning on a wide spectrum of devices such as cloud servers, self-driving cars, embedded devices, etc. Not only did the creators of TVM attempt to make custom compilations for different devices but also include optimizations for each device's physical memory design. As we can see in Fig. 1, memory is structured differently on different devices such as CPUs, GPUs and TPU-like accelerators.

TVM's solutions to the proposed problems are exposed graph-level and operator-level optimizations, and different memory organization on different devices.



. Fig. 1

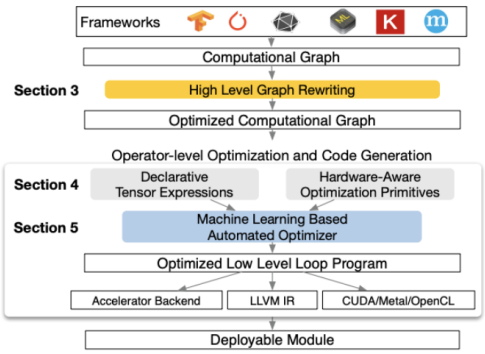
Another problem addressed by TVM is ***producing efficient code without manually tuning operators***, or building Dynamic Cost Models. As mentioned in the related works section, TVM approaches this problem using the following:

Tensor Expression Language provides program transformations that generate different versions of the program with various optimizations.

Automated Program Optimization Framework is a ML-based cost model that adapts with new data on different hardware backends.

Graph Rewriter adds high-level and operator-level optimizations.

TVM provides builds for existing frameworks such as: *keras, tensorflow,* and *pytorch*. It also supports multiple deployment back-ends in languages such as *C++, Java* and *Python*. The steps for converting a framework's model to a TVM optimized model are shown in Fig. 2:



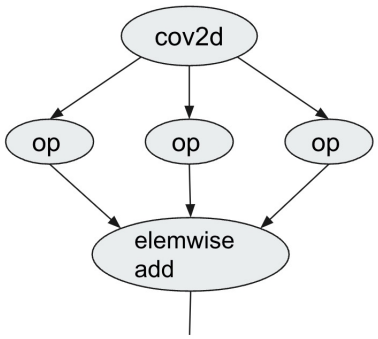
. Fig. 2

- Takes the input model from the existing framework
- Transforms it into a computational graph representation
- Identifies possible code optimizations
- Uses ML to find optimized operators
- Packs generated code into deployable model for the different devices

4. Operator Fusion

Operator fusion combines multiple operators into a single kernel without saving the intermediate results in memory. This optimization can greatly reduce execution time, particularly in GPUs and specialized accelerators. Usually, it achieves 1.2x -2.0x speed up through reducing memory accessing. We classified graph operators to 4 categories: (1) injective (one-to-one map, e.g., add), (2) reduction (e.g., sum), (3) complexout-fusable (can fuse element-wise map to output, e.g.,conv2d), and (4) opaque (cannot be fused, e.g., sort). We also define some fusing rules, and the later fusing algorithm will follow these rules. Multiple injective operators can be fused into another injective operator. A reduction operator can be fused with input injective operators (e.g., fuse scale and sum). For complexout-fusable operator such as cov2d, we can fuse element-wise operators to its output.

After we define those categories and fusing rules, we can construct a forward fusion algorithm. The forward fusion algorithm fuses nodes to their immediate post-dominator. For example, in figure 3, cov2d can be fused to element-wise add. However, at the point of conv2d we do not necessarily know that all the future paths will merge at the elemwise add. To solve this, the fusion algorithm applies post-dominator analysis.



. Fig. 3 Operator fusion example

The general algorithm is as follows:

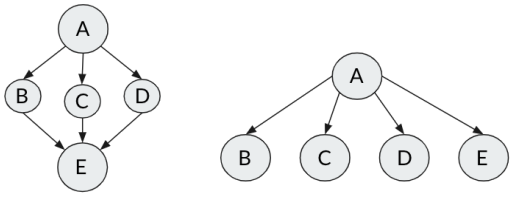
1. Construct a directed acyclic graph of dataflow graph for dominator analysis
2. Construct a post-dominator tree which gives immediate post dominator of each node.
3. Run fusion algorithm with the given post-dominator information.

Note that, because we run analysis on a DAG, we use a single pass post-dominator tree construction algorithm via LCA(more details in next section), which is simpler than the full version that handles cycles. The fusion algorithm traverses from each node and checks if it can be fused to its immediate post dominator. The condition to be considered is whether all the paths between a node and its immediate post-dominator satisfy the fusing rules. In the next section, we will implement a backward fusion algorithm.

5. Experimentation

In order to implement backward operator fusion, our approach was to modify the source code of TVM to fuse nodes in the computational graph to their immediate pre-dominators rather than post-dominators. A node x's immediate pre-dominator is defined as the closest node y for which all paths from the entry/root must go through through y in order to reach x. In order to do so, we added a member variable to the node class of the dataflow graph to store the inputs, or parents, of each node in the graph. Then, we created a function to construct a pre-dominator tree from the directed

acyclic graph (DAG) by looping over each node in the DAG, creating a new node in the dominator tree corresponding to this node, and using the Least Common Ancestor algorithm to set its parent in the tree. The Least Common Ancestor algorithm would take in the node's list of inputs in the original DAG and find their shared ancestor in the dominator tree that is furthest from the root, returning this node. The node that was returned would then be assigned as the new tree node's parent in the dominator tree.



. Fig. 4

The graphs shown above illustrate the construction of the pre-dominator tree (on the right) from the original dataflow graph (on the left). Since node A is the immediate pre-dominator of nodes B, C, D, and E in the DAG, it will be the parent of those nodes in the dominator tree due to the Least Common Ancestor algorithm.

The fusion algorithm then traverses through the nodes in the original graph and checks whether each node can be fused to its pre-dominator, using the information from the pre-dominator tree.

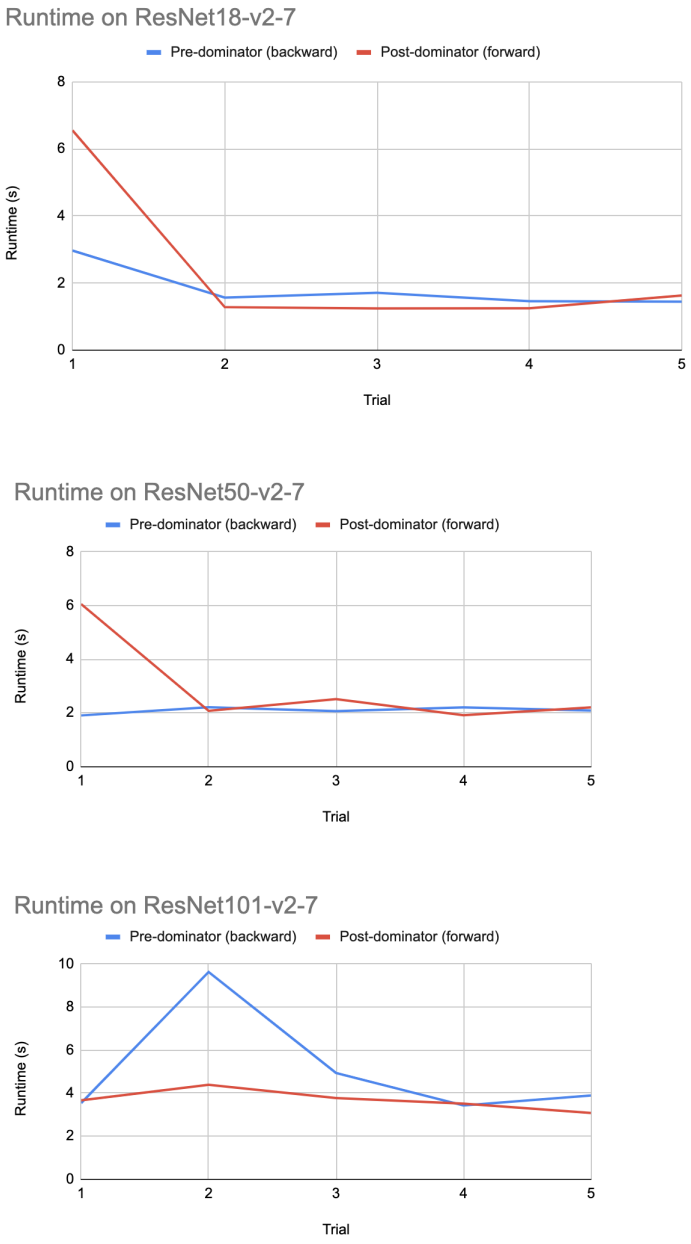
6. Results

We evaluated our method by comparing the compile time and run time using forwards and backwards fusion on three models: ResNet-18 v2, ResNet-50 v2, and ResNet-101 v2. ResNet is a convolutional neural network that is designed to classify images, and is pretrained on over 1 million images with 1000 different possible classifications (3). The three models we used for testing were of different sizes; ResNet-18 is made up of 18 layers, ResNet-50 is 50 layers deep, and ResNet-101 is 101 layers deep. Below is a table displaying the speedup in compile time by using the backward fusion method for each model over 10 trials.

Compile Time Speedup Using Backwards Fusion	
ResNet-18 v2	6.88%
ResNet-50 v2	0.98%
ResNet-101 v2	22.46%

As shown in the table, we observed significant speedups in compile time when using the backward fusion method. However, when looking into the actual numbers of operators fused, we found that the backward method was fusing significantly fewer operators than the forward method, which may have been a factor in the reduced compile time.

The graphs below display the differences in runtime we observed on the three models between the forward and backward methods.



. Fig. 5

We can see from the graphs that the methods performed quite similarly when considering runtime, with a few spikes in execution time on individual trials that may have been due to other processes running. In general it seems that the original forward fusion method may have had slightly better runtime, which we would hypothesize is due to the fact that it committed more fusions. The accuracy of the predictions made did not change between the two methods.

7. Future Work

Given the chance to pursue this problem further in the future, one area we would want to spend more time on is deeper analysis of the new operators that backward fusion gives us which do not appear in forward fusion. The following are potential ideas we would have liked to pursue in this project given a longer time frame:

1. Improve the backward fusing algorithm to allow more operator fusion. We are currently using the framework from forward fusion. By altering the fusing rules, we may be able to find fusing opportunities that do not exist in forward fusion algorithm.
2. Run forward and backward fusion simultaneously to allow for increased fusion opportunities.
3. Test the performance of backward fusion on a wider variety of models, as currently we have only tested on ResNet models.

References

[1] Tensorflow operation fusion nbsp;: nbsp; tensorflow lite.

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

[3] Leandro Nunes, Chris Hoge, and Matthew Barrett. Compiling and optimizing a model with tvmc.