Summary of Ansor: Generating High-Performance Tensor Programs for Deep Learning

Yabin Dong (dyabin), Hanchi Zhang (tonyzhc), Haofeng Zhu (hfzhu)

Problem and Motivation

The deep learning system stack contains high-level frameworks (e.g., Pytorch, TensorFlow, mxnet) and low-level hardware (e.g., CPU, GPU), as well as compilers that receive the compute declaration from high-level frameworks and produce optimized code to the low-level hardware. In deep learning workloads, efficient execution of deep learning models requires optimizing, high-performance tensor programs. Current solutions (e.g., TVM, Halide's auto-scheduler) are not fully automated, and huge engineering efforts and required to develop compilers for various hardware platforms. Besides, one single compute declaration may have significant amounts of implementations making the optimization challenge. Also, current solutions have limited search space. This work is motivated to automatically generate a large search space and search efficiently; thus, the tensor program generation framework, Ansor, is developed in this paper.

Hypothesis

This work aims to tackle three challenges. The first challenge is that how to automatically develop a large search space. This work proposed a hierarchical search space to solve this challenge. The second challenge is how to search efficiently in this large space. This work considers first sample the complete programs and then do the fine-tuning process. The third challenge is regarding the work scheduling which is tackled by a task scheduler to reduce time wasted on unimportant subgraphs. It is based on the hypothesis that tuning some layers of DNN models may not improve the end-to-end performance significantly.

Solution Overview

Figure 1 displays the overview of the Ansor system. Each step of Ansor is corresponding to a challenge mentioned before. Overall, Ansor takes deep learning models as the input and uses the operator fusion algorithm to partition into subgraphs. In the current iteration, a task scheduler is used for all subgraphs to allocate tasks. After the task scheduler, each subgraph is sent to a program sampler where high-level sketches and low-level annotations are generated. It should note that randomly generated program samples are not necessarily efficient. Hence after the program sampler, a batch of initial programs is sent to the performance tuner to produce optimized programs. Finally, optimized programs are measured in hardware and the execution time of the program is fed back to the task scheduler, program sampler, and performance tuner for the next iteration. This process will run until the system converges or the profiling budget time is running out.

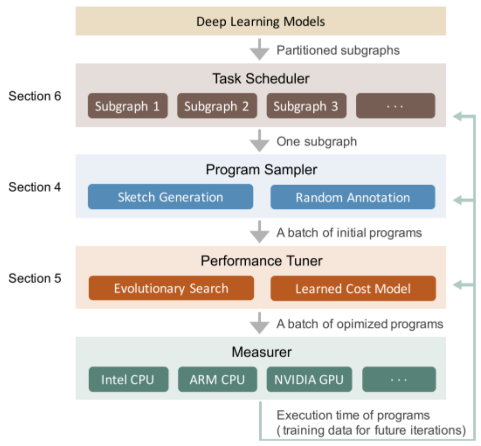

Figure 1: System overview of Ansor

(i) Program Sampler: The program sampler will automatically build a two-level large search space: sketch and annotation, where the sketch will determine the graph structures while annotation includes detail factors. In the sketch generation process, the paper proposed a derivation-based enumeration approach to automatically generate sketches by applying basic rules (e.g., Skip, Always Inline). After the list of sketches is generated, a sketch will be randomly selected and randomly annotate with computing details (e.g., loop annotations, tile sizes, parallel, unroll) to make it a complete program. Then, Ansor will uniformly sample from the space and send it to the Performance Tuner. The advantage of using the two-level hierarchical structure is that separating the sketch and annotation makes it flexible to enumerate them for efficient sampling in a large space. Besides, as the sketch will be randomly generated, Ansor avoids pre-defining graph structures that limit the searching space.

(ii) Performance Tuner: The objective of the performance tuner is to optimize programs coming from the program sampler as the program sampler does not guarantee the quality of programs. The performance tuner will use the evolutionary search at each iteration to generate a small batch of promising programs based on the learned cost model. At the end of the current iteration, the execution time cost of the program will be measured on hardware and sent as feedback to retrain the cost model. For one iteration, in the evolutionary search, the mutation rule will randomly mutate tile size, parallel, pragma, computation locations, and the crossover rule will merge to two or more parents computational graphs to generate the offspring.

(iii) Task Scheduler: When fine-tuning the deep learning models, one problem is that training some part of the model may not necessarily improve the end-to-end model performance. Hence, the task scheduler essentially aims to avoid wasting time on tuning parameters of unimportant subgraphs. The task scheduler of Ansor forms a minimization problem and reschedule the time budget after a time window based on the subgraph latency. The task scheduler of Ansor slices the task time and prioritizing important subgraphs.

Limitations and Possible Improvements

Firstly, Ansor cannot optimize dynamic computational graphs. The Ansor takes the DNN models as the input; hence it requires static and known computational graphs. Secondly, Ansor has difficulties on sparse neural networks where a large subset of model parameters are zero. The paper mentioned that some ideas of Ansor can still be used but the search space needs to be re-designed. Finally, it is challenging for Ansor to perform platform-dependent optimizations as Ansor optimizes the program at a high level and does not use many special instructions (e.g., NVIDIA Tensor Core).

Summary of Class Discussion

(i) The Halide auto-scheduler method is the very early idea in the field of optimizing the tensor program compilers. TVM is partially motivated by the Halide method and TVM is popularly used. The impacts of optimizing the tensor program compilers (TVM, Ansor) may gradually show stronger influences in the future.

(ii) How to make sure the crossover in the evolutionary search will produce valid results?
Answer: The results of the crossover may not be necessarily compatible. In the worst case when the output code is not runnable, the crossover will merely take nodes from one parent graph.

(iii): Can Ansor be applied to the combination of operators? For instance, can it figure out that $A \times B \times B^{-1}$ will just be A and skip the calculation?
Answer: No. Ansor is a static analysis and it has the limitation that the computational graph must be known in advance. Hence, it does not have the ability to skip part of the computation.

(iv): Is it possible to hybrid Ansor and TASO?
Answer: Ansor runs a model, executes on hardware, and predicts future executions. As Ansor will randomly generate a large space of computational graphs and then fine-tune them. There is a possibility to integrate the idea of TASO which optimizes the computational graphs.

Summary of TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions

Yabin Dong (dyabin), Hanchi Zhang (tonyzhc), Haofeng Zhu (hfzhu)

Problem and Motivation

Deep neural network frameworks represent a neural architecture as a computation graph, where each node is a mathematical tensor operator. To improve the runtime performance of a computation graph, the most common form of optimization is graph substitutions that replace a subgraph matching a specific pattern with a functionally equivalent subgraph with improved performance. However, existing DNN frameworks optimize a computation graph by applying graph substitutions that are manually designed, and they have several shortcomings: low maintainability, poor optimization with regards to data layout and no guarantee for correctness. As a result, TASO aims to tackle these problems and build an automatic graph substitution optimizer.

Hypothesis

The paper proposes the following hypotheses: First, the system could automatically generate graph substitutions instead of depending on manual effort that creates maintainability and correctness issues; second, the system could use formal verification to ensure correctness; third, the systems could jointly optimize with regards to both graph substitution and data layout, thus achieving maximum efficiency.

Solution Overview

As illustrated in figure 2, the TASO system consists of the four following major steps:

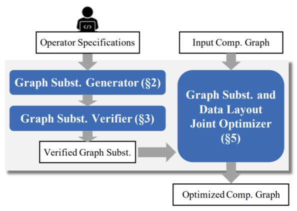
Operator Specifications
Input Comp. Graph
Graph Subst. Generator (§2)
Graph Subst. Verifier (§3)
Verified Graph Subst.
Graph Subst. and Data Layout Joint Optimizer (§5)
Optimized Comp. Graph

Figure 2: System overview of TASO

(i) Graph Substitution Generator: The generator begins by enumerating all potential graphs and calculating their fingerprints, which is a hash of the output tensors obtained by evaluating the graph on random integer input tensors and a number of constants e.g. the identity matrix. To construct a graph, the generator iteratively adds an operator in the current graph by enumerating the type of the operator and the input tensors to the operator. It then performs a depth-first search algorithm to construct all acyclic computation graphs that do not contain duplicated computation (i.e. no two operators perform the same computation on the same input tensors). After computing all the fingerprints, candidate pairs of graphs are further tested on their equivalences by a set of test cases that involves floating point input tensors. These tests set a small threshold value to determine the equivalence of output tensors. Eventually, this step produces a set of candidate graph substitution mappings.

(ii) Graph Substitution Verifier: The research team first manually wrote and validated a small set of operator properties expressed in first-order logic for each operator defined by TASO. The validation process involves testing these properties for all combinations of parameter values and tensor sizes up to a small bound of $4 \times 4 \times 4 \times 4$. Additionally, the verifier also checks that the set of operator properties is consistent and does not contain redundancies. Then, given these properties, the team used a first-order theorem prover, which in this case is their implementation of Microsoft Research's Z3 theorem prover. This step thus further ensures the correctness of candidate pairs generated from the last step.

(iii) Pruning Redundant Substitutions: The pruning is done in two situations:

- Renaming input tensors to avoid identical substitutions e.g. $A \times (B \times A) \Leftrightarrow (A \times B) \times A$ and $A \times (B \times C) \Leftrightarrow (A \times B) \times C$
- Eliminating substitutions whose source and target graphs have a common graph

It was shown that these pruning techniques were able to reduce the number of substitutions to consider by 39×.

(iv) Joint Optimizer: The optimizer uses a cost-based backtracking search algorithm to search for an optimized computation graph by applying verified substitutions, which is based on previous work from MetaFlow. TASO extends MetaFlow's search algorithm to also consider possible layout optimization opportunities when performing substitutions. When TASO searches for the optimal graph substitution, multiple graphs with identical graph structure but different data layouts may need to be searched. Due to the fact that DNN operators perform dense linear algebra with no branches, and their performance on hardware is highly consistent, TASO simply measures the execution time of a DNN operator once for each configuration and data layout; the system can then estimate the performance of the whole graph.

Built on top of MetaFlow and reusing its cost-based backtracking search, TASO is designed and implemented as a generic and extensible computation graph optimizer for tensor computations such that new tensor operators can be easily added. The current implementation of TASO supports up to 12 tensor operators and 4 constant tensors that are useful in substitutions. TASO uses operator properties specified by user to verify the generated graph substitutions. In order to include new tensor operators, TASO requires reference implementations for the operator and specifications of operator properties. TASO also boasts portability on TensorRT and TVM, enabling them to use TASO's optimization to improve performance.

The paper compares the end-to-end inference performance among several popular deep learning frameworks and TASO on variants of ResNet and NasNet. TASO's performance with both the cuDNN and TVM backends are evaluated. TASO achieves on par performance with existing frameworks for ResNet, and outperforms existing frameworks on the four remaining DNNs with new operators and graph structures.

Limitations and Possible Improvements

One limitation of TASO is that it relies on user provided operator properties. It is desirable to eliminate the effort on operator properties altogether. One possible approach is to automatically verify the substitutions directly against the implementations of operators.

TASO also has limited scalability for generator, which requires saving the fingerprints of all computation graphs up to a fixed size. I would suggest implementing a distributed generator to overcome the scalability bottleneck.

Summary of Class Discussion

(i) Is graph substitution basically run equivalent computation graph that is faster on hardware?
Answer: Yes.
(ii) During fingerprinting, how often do you come across the scenarios where you cannot remove all the "lucky cases" even with efficient and strong pruning?
Answer: The probability is very low.
(iii) Won't casting to ints affect accuracy?
Answer: Probably should.
(iv) Is it 28744 possible substitutions for any graph?
Answer: 28744 is the number of possible substitutions for their method for the 4op combination from the N-op they considered for any graph.
(v) How to choose which part of the graph to do tensor renaming?
Answer: Only the input part.