Summary of "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning"

Jiaqing Ni (jiaqni), Hengjia Zhang (hengjia)

Problem and Motivation

Current Deep Learning frameworks such as MxNet, Tensorflow, and PyTorch optimize hardware utilization by computation graph, which is usually too high-level to handle hardware back-end-specific operator-level transformations. On the other hand, operator-level libraries can hardly be transported across hardware platforms, resulting in costly manual tuning.
An optimizing compiler is in the call to attack the above challenges. Meantime, it needs to handle novel tensor-computation primitives and exploit a potentially huge optimization space (combinatorial choices of memory access, threading pattern, and novel hardware primitives). In this paper, it proposes TVM, a compiler that takes a high-level specification of a deep learning program from existing frameworks and generates low-level optimized code for a diverse set of hardware back-ends, so that both graph-level and operator-level optimizations for different hardware back-ends (CPU, GPU, FPGA, Deep Learning Accelerator) can be achieved at the same time.

Hypothesis

TVM is able to solve optimization challenges specific to deep learning (e.g. high-level operator fusion, mapping to arbitrary hardware primitives and memory latency hiding) and automate optimizations of low-level programs to hardware characteristics. It's trying to solve the dilemma for machine learning researchers who are supposed to focus on novel ML model proposal but struggling to the workaround for deploying their models to back-end hardware like FPGA or GPU.

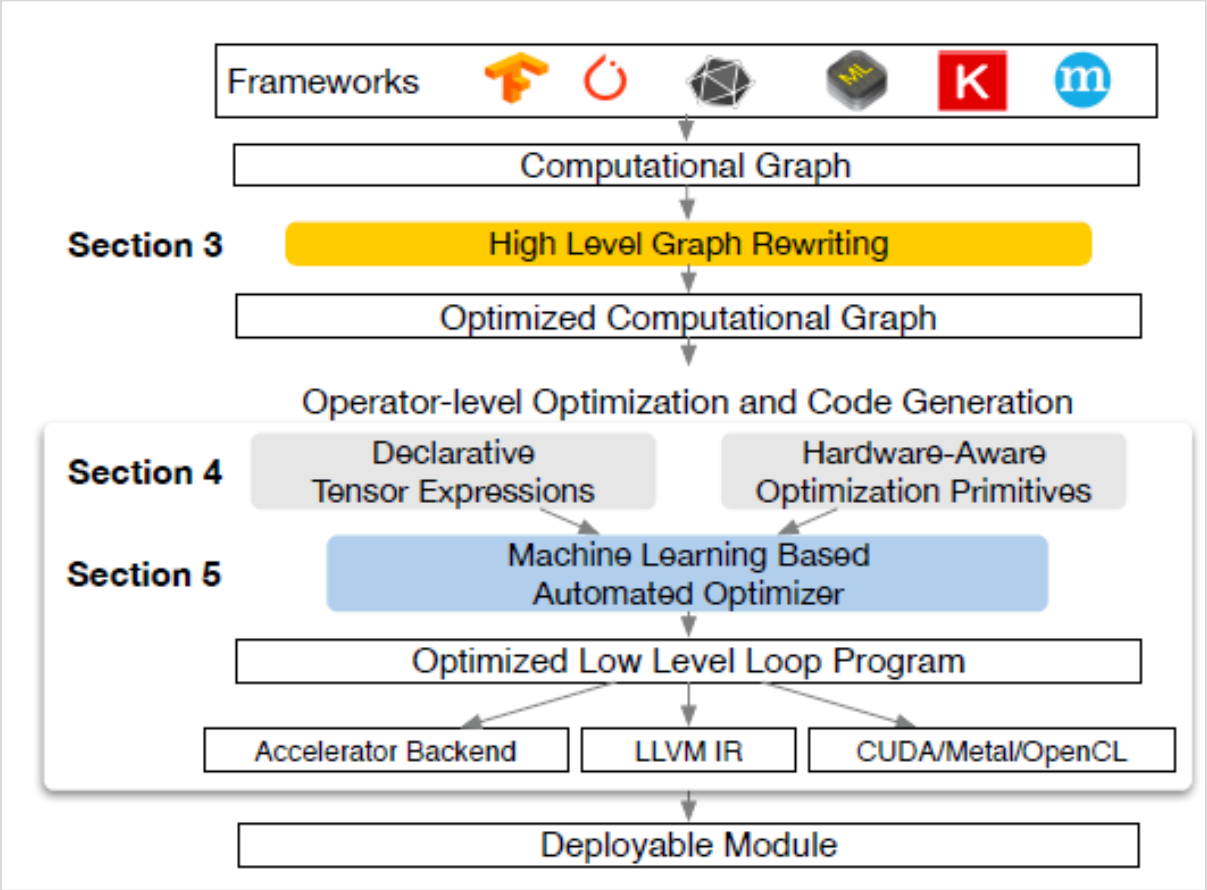
Solution Overview

TVM Overview:

The following graph summarizes the execution steps in TVM:

1. TVM will take a model from a deep learning framework and transform it into a computational graph representation.
2. After TVM gets the graph representation, it will perform high-level dataflow rewriting to generate an optimized graph.

3. The operator-level optimization module generates efficient code for fused operators in this optimized graph.
4. TVM identifies possible code optimizations for a given hardware target's operators.
5. The system packs the generated code into a deployable module.



Optimizing Computational Graphs

TVM reuses the general computational graph representation where nodes represent operations and edges represent data dependencies. It implements some graph-level optimizations:

1. Operator Fusion:

   TVM fuses multiple small operations together into a single kernel without saving the intermediate results in the memory to reduce execution time. There are 4 kinds of graph operators: injective, reduction, complex-out-fusable, and opaque. TVM use the following rule to fuse these operators:

   1. Multiple injective operators can be fused into another injective operator.
   2. A reduction operator can be fused with input injective operators
   3. Element-wise operators can be fused to complex-out-fusable operator's result

2. Data Layout Transformation:

   TVM converts the graph into one that can use better internal data layouts for execution on the target hardware. It specifies the preferred data layout to each operator and then

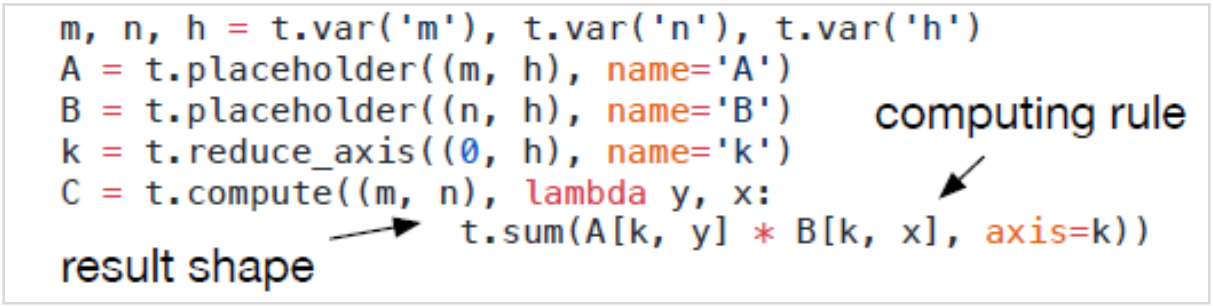
performs the proper layout transformation if the preferred data layout of the producer and consumer do not match.

Generating Tensor Operations

TVM also generates valid implementations on different hardware back-end and chooses an optimized implementation to produce efficient code for each operator. This process builds on Halide's idea of decoupling descriptions from computation rules.

Tensor Expression and Schedule Space

TVM introduces a tensor expression language to support automatic code generation. The following code shows an example tensor expression to compute transposed matrix multiplication:



TVM also builds a schedule space by incrementally applying basic transformations that preserve the program's logical equivalence.

Nested Parallelism with Coorperation

TVM proposed an idea for nested parallelism called nested parallelism with cooperation, where essentially threads can cooperate and fetch the data they need and place it into a shared memory space. The advantage of this optimization is that it can take advantage of the GPU memory hierarchy and enable data reuse.

Tensorization

In order to extend TVM to support new hardware architectures, it introduces tensorization which decouples the schedule from specific hardware primitives. It uses the same Tensor expression language to declare hardware intrinsic and lowering rule. TVM also introduces a tensorize schedule primitive to replace a unit of computation with the corresponding intrinsics.

Explicit Memory Latency Hiding

For latency hiding, TVM introduces a virtual threading scheduling primitive that lets programmers specify a high-level data-parallel program. TVM will then automatically lowers the program to a single instruction stream with low-level explicit synchronization.

Schedule Space Specification

A schedule template specification API is built inside of TVM to let a developer declare knobs in the schedule space. A generic master template for each hardware back-end is also built to automatically extract possible knobs based on the computation description.

ML-Based Cost Model

Using autotuning to find the best schedule from a large configuration space requires many

experiments. Instead, TVM takes a statistical approach to solve the cost modeling problem, where a schedule explorer proposes configurations that may improve an operator's performance. For each schedule configuration, TVM uses an ML model that takes the lowered loop program as input and predicts its running time on a given hardware back-end. For machine learning model design choice, both quality and speed should be considered and TVM employs a gradient tree boosting model.

Limitations and possible improvements

There are some limitations of TVM that we think of: It seems like TVM doesn't support deployment on different devices for distributed code generation if multiple hardware back-ends are involved at the same time. Also, TVM mainly focuses on the optimization of graphs, tensor expressions, and automated optimizations. Maybe it can go one layer deeper to specify some more advanced optimization given the devices and still achieve good performance on novel deep learning accelerators.

Summary of "An Intermediate Representation for Optimizing Machine Learning Pipelines"

Problem and Motivation

Machine learning includes not only the model training, but also includes the preprocessing steps, like data cleaning and feature transformation. The general dataflow graph will transform collections via user-defined functions (UDFs), which defer program execution by providing a type-based domain-specific language (DSL). Usually in data processing, the relational algebra and UDF are commonly used while in model training, linear algebra and iterations are involved. Current dedicated systems like SystemML and TensorFlow suffer from three issues in the context of end-to-end pipelines for model training:

1. Development, maintenance, and debugging of end-to-end pipelines is a tedious process.
2. Hard to optimize across linear and relational algebra since preprocessing and ML are often executed in different systems.
3. Neither shallowly embedded libraries nor type-based DSLs can reason about native UDFs and control flow.

To solve these issues, this paper proposes Lara, which is a quotation-based DSL for collection processing based on Emma.

Hypothesis

Lara depends highly on Emma, another quotation-based DSL for collection processing and assume people have already known about Emma and Scala. E an algebraic data type called DataBag which enables declarative program specification based on for-comprehensions. Also, Emma's intermediate representation enables operator fusion and implicit caching.

Lara extends Emma's idea with adding Matrix and Vector data type in its API and Intermediate representation. It provides two views on the IR to perform diverse optimizations:

1. The monadic view represents operations on DataBag and Matrix
2. The combinator view captures high-level semantics of relational and linear algebra operators.

Solution Overview

In the first place, this paper talks about the overview of import design decisions for Lara and introduces the two views we talked about in Hypothesis.
Lara extends the API and IR of Emma by supporting matrices and vectors. Also, Lara enables the declarative specification of relational operators with for-comprehensions.
There're some highlight key aspects of Lara's IR:

1. The UDFs for the second-order functions are defined as closure or inline.

2. Data types like DataBag, Matrix and Vector can be further optimized by add optimize operator surrounding the code.

3. Support type conversion in the API.

4. The choices of type os operator may not enforce a particular physical execution backend.

5. White-box UDFs in the IR enable reasoning about read and write accesses to the processed elements

6. High-level linear algebra operators in the API can select specialized operator implementations.

Lara also builds two views on the top of Emma's IR, which are monadic view and combinator view. For monadic view, it enables Lara to examine and optimize applications of UDFs (fusion and pushdown). For combinator view, it allows Lara to apply fusion over UDFs based on the properties of monads. These two views help Lara optimize end-to-end pipelines, which we will discuss in detail next.

There are several optimizations that Lara is able to do:

1. Operator Pushdown:
   Lara can push UDF applications from one type to another in the IR. It uses a special method called conversion methods, which allow Lara to track data provenance across type conversions so that Lara can know how and where both types are stored. Then it uses a unified representation of DataBag and Matrix as monads in the IR enables the pushdown of UDFs.

2. Operator Fusion:
   Lara can fuse the consecutive fold and map applications if the feature transformation is applied on multiple disjoint columns so that it may only require a single fold and map operation.

3. Cross Validation:
   Since Lara integrates new high-level operators into its API, it introduces an optimization for k-fold cross validation. Essentially, Lara pre-computes linear algebra operations on the individual training set splits outside of the validation loop to avoid redundant computations.

Limitations and possible improvements

Lara at the moment seems like a prototype instead of a mature DSL. Lara is not integrated in a dedicated runtime nor uses code-generation and it doesn't have a robust caching mechanism to test different models on the same feature set. So far it seems like Lara doesn't have a solid caching for its new coming Matrix and Vector data type while Emma has already supported caching for DataBag. Adding caching for Matrix and Vector may highly improve the performance of Lara. For operator fusion, Lara also doesn't support fusions of linear algebra operators as its backend doesn't support it. The improvement on this may also let Lara be more flexible in optimization.

Summary of Class Discussion

Discussion during the presentation

What type of machine learning model in automating optimization?
Supervised learning, in the figure it shows an iteration update.

How much data and how long training?
They didn't mention in detail, the focus the faster inference. It claims that it's better than auto-tuning, but they didn't compare with auto-tuning in evaluation part.

What is tensor flow XLA?
It's a similar thing with TVM, but only for TensorFlow graph. The interesting thing is that TensorFLow XLA performs worse than solely using TensorFlow.

What is the difference between embedded GPU and server-class GPU?
Usually there's lower power in embedded GPU while server-class GPU may have higher precision.

What is fold in Lara?
Fold is kind of like Reduce we have in MapReduce.

Discussion after the presentation

How tofu is related with TVM and Lara?
In tofu, it talks about if the model is too large, how it needs to be partitioned across multiple devices. But in TVM, they don't talk about partitions across multiple devices but using one device. Lara is even further away because in Lara it talks about the entire pipeline. The goal of TVM is to compile deep networks in specific devices so that they can run faster.

Is it possible to run Lara across multiple machines?
In theory, it should be possible. There's another paper called WELD where they want to co-optimize all of things like preprocessing or machine learning at the same time. Weld can be run on distributed devices. Overall it's possible but it might be expensive.

TVM focuses on hardware-specific low-level optimization and the paper mainly discusses the optimization with respect a specific computing device (e.g. CPU*GPU*TPU). But can TVM be extended to perform an overall optimization of a job given a heterogeneous system consisting of a mixture of different computing devices? (From Piazza)
It should be possible and it triggers a more interesting question: If the heterogeneous

devices can overload jobs and optimize that particular heterogeneous combination since all of these devices are interchangeable computing devices but they have different speeds. How should TVM co-optimize them as a whole instead of individually optimization.