NNVM, A Survey

Author: Wang, Shaodong

Table of Contents:

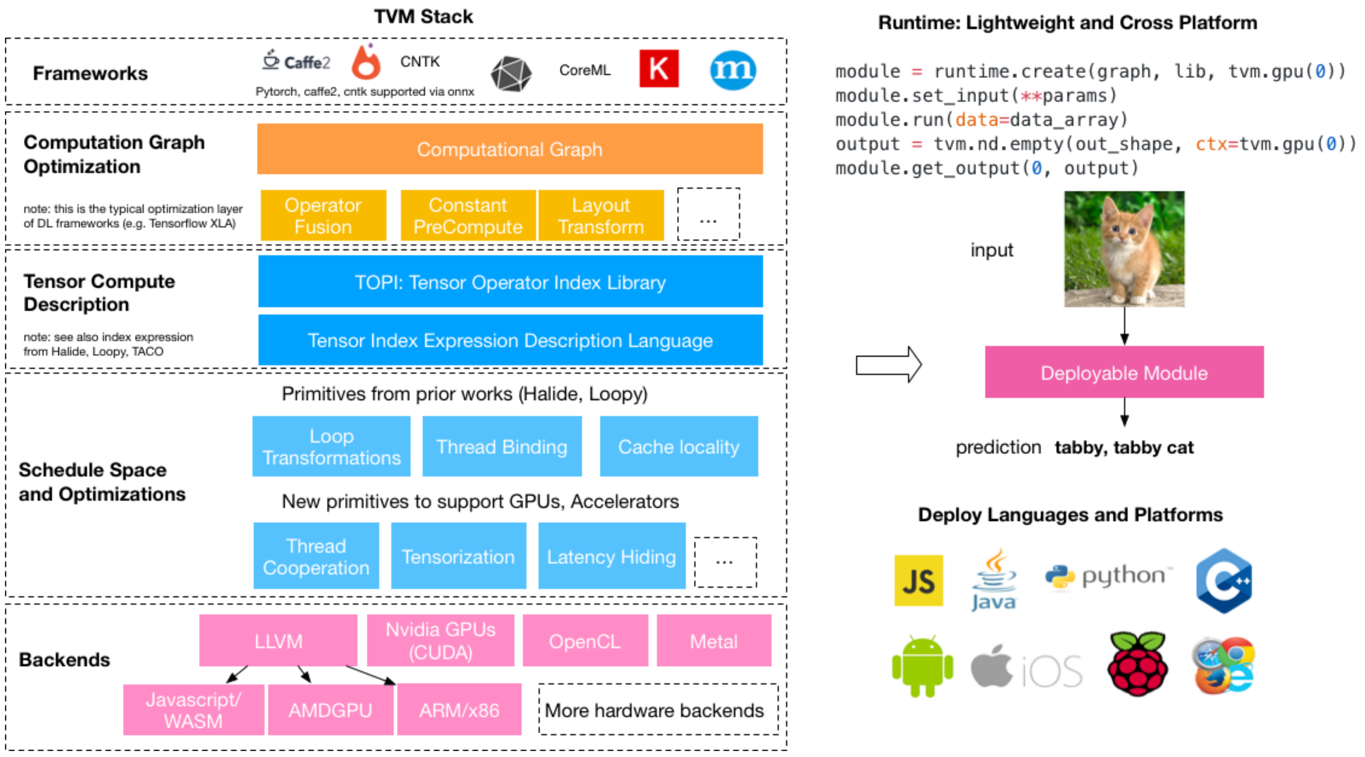
Introduction

Nowadays, deep learning developers have a wide range of AI frameworks to choose from, i.e. Tensorflow, Theano, Keras, PyTorch, MxNet, Caffe, to name a few. And as a convention, AI frameworks always need to support various hardware backends ranging from GPU, CPU to embedded chips. This is both a bless and a curse, since the diversity of AI solutions provide a lot of functionalities for the users to choose from, meanwhile it has brought a lot of difficulties not only for developers and researchers, but also for the framework maintainers and chip vendors. More specifically, the AI community now faces the following challenges:

1. Users need to learn different frameworks and constantly switch between them. There are many reasons for this. First of all, different open-source research projects are often implemented using various AI frameworks. Second, some frameworks are more user-friendly which is suitable for experimenting ideas, while others are less accessible but heavily optimized which is good for deployment.
2. Framework developers need to support and optimize for a lot of hardware systems, meanwhile maintaining a consistent API, which is tedious work to do.
3. The hardware vendors also have a hard time optimizing theirs chips for various AI tools since they all execute in a unique way and even a simple convolution operation might need to be treated differently.

For these reasons, researchers and developers from University of Washington and and the DMLC community have created the TVM stack, a unified optimization tool stack aiming to solve this mess and close the gap between AI frameworks and hardwares. This tool stack is composed of

- TVM: the tensor level optimization stack.
- NNVM: the graph level optimization stack.

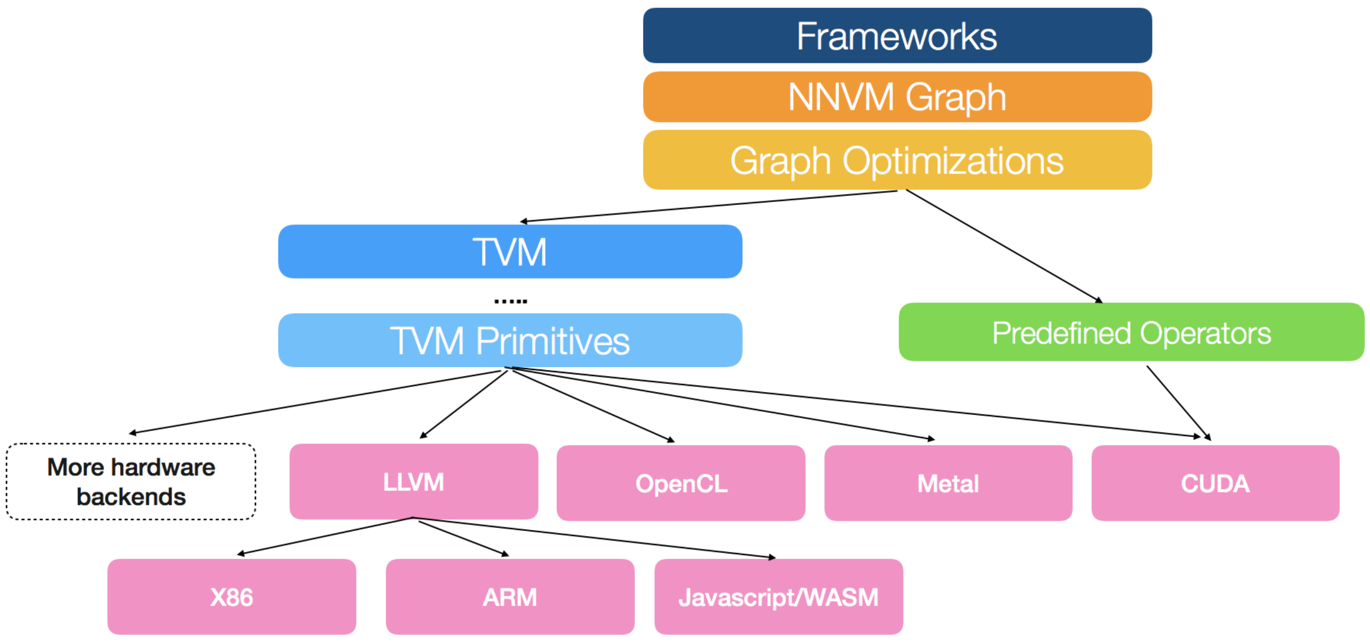

TVM Stack
Frameworks
Caffe2
CNTK
CoreML
Pytorch, caffe2, cntk supported via onnx
Computation Graph Optimization
note: this is the typical optimization layer of DL frameworks (e.g. Tensorflow XLA)
Computational Graph
Operator Fusion
Constant PreCompute
Layout Transform
...
Tensor Compute Description
note: see also index expression from Halide, Loopy, TACO
TOPI: Tensor Operator Index Library
Tensor Index Expression Description Language
Schedule Space and Optimizations
Primitives from prior works (Halide, Loopy)
Loop Transformations
Thread Binding
Cache locality
New primitives to support GPUs, Accelerators
Thread Cooperation
Tensorization
Latency Hiding
...
Backends
LLVM
Nvidia GPUs (CUDA)
OpenCL
Metal
Javascript/ WASM
AMDGPU
ARM/x86
More hardware backends
Runtime: Lightweight and Cross Platform
input
Deployable Module
prediction tabby, tabby cat
Deploy Languages and Platforms
JS
Java
python
iOS

```
module = runtime.create(graph, lib, tvm.gpu(0))
module.set_input(**params)
module.run(data=data_array)
output = tvm.nd.empty(out_shape, ctx=tvm.gpu(0))
module.get_output(0, output)
```

The above picture shows the overall architecture of the TVM software stack and its common usage. It can be seen that the TVM stack support different hardwares and AI frameworks and enables cross platform deployment using different programming languages. In the remainder of this survey, I'll first explain more details on these two major components as well as the architecture of the TVM stack. After that, I'll focus on NNVM and explain its internal components and techniques.

The TVM Stack

Overall Architecture



As seen in the above figure and the figure in the previous section, the TVM stack is composed of several layers, where higher-level components are built on top of more fundamental layers.

The bare metal of this software stack is a bunch of hardware apis, including LLVM, Cuda, OpenCL, Metal, OpenGL and more. On top of that is the low-level IR layer and tensor level optimization layer, where the hardware-targeted optimization took place. Then there's the graph level optimization layer where computational graphs from different AI frameworks are optimized to reduce computation and memory usage. The top of this software stack is the api to integrate and compile models from various AI frameworks.

TVM and Tensor Level Optimization

TVM borrows the wisdoms from the traditional compiler community and builds a bunch of optimization primitives to optimize deep learning computation kernels, such as convolution. This component is heavily hardware oriented and aims to optimize computations as efficiently as possible on different hardware platforms. This survey will focus on NNVM and readers may refer to http://tvmlang.org/ for more technical details.

NNVM and Graph Level Optimization

On the other hand, NNVM is targeted at optimizing computational graphs. It first contains a graph compiler to compile models defined in different frameworks into a common graph IR. Then TVM will operate on this intermediate representation for low-level optimizations. Beyond that, NNVM will also optimize the graph structure for better efficiency.

Using NNVM

Getting Started

Here's an example on how to use NNVM to compile a trained network into NNVM's intermediate representation and deploy it on other platforms as you like. Please refer to the official NNVM documentation for the full tutorial. Here I'll explain the key workflow.

First of all, you need to declare the computational graph, either converted from external frameworks using `nnvm.frontend`, or using NNVM symbolic infrastructures `nnvm.symbol`.

Next up, you can compile the graph into NNVM graph IR using `nnvm.graph.build` function. The returned values are particularly interesting. The first value is an optimized graph. Notice how the graph IR is simplified and operations are fused into one. The second value is the a `tvm.module` which contains compiled codes for the graph. The third value is a dict of trainable parameters.

After that, we can save the network, including the graph definition, compiled codes and parameters onto hard disk and deploy it later on the target hardware platform.

Modules

The NNVM toolset contains the following components:

`nnvm.compiler`

The graph compiler which compile symbolic graph definitions into an intermediate representation.

`nnvm.frontend`

Convert graphs defined in other deep learning frameworks, currently supporting MxNet, ONNX, CoreML and Keras.

`nnvm.symbol`

Symbolic tensor operations supported by NNVM, including common mathematical operations used in other AI frameworks.

`nnvm.graph`

The NNVM graph IR api.

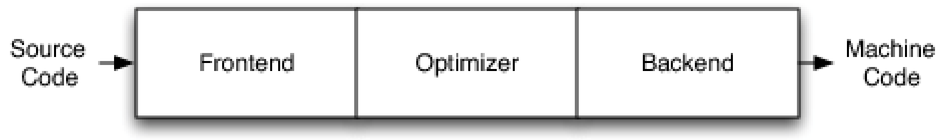
`nnvm.top`

Provide low level access to tensor operations.

Inside NNVM

Compiler 101

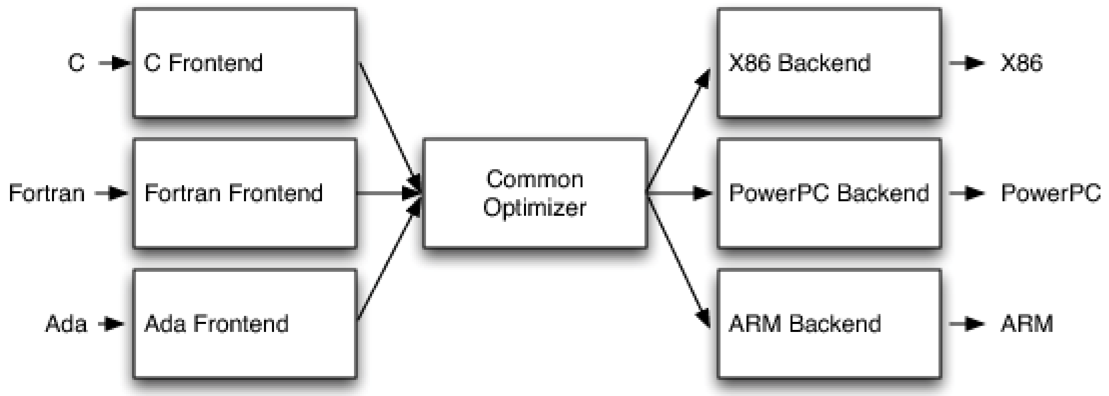
A classical compiler is often separated into three major components, namely the frontend, the optimizer and the backend, shown in the figure below.



The frontend is responsible of parsing the source code into an intermediate representation, and reporting any grammar error in the source code. This stage is often broken into three phases, which are lexical analysis (aka lexing or tokenization), syntax analysis (aka parsing) and semantic analysis. These three phases gradually parse the source code into tokens, and build up a parse tree and a symbol table.

The optimizer is responsible of optimizing the IR and producing an IR that runs faster, like a filter in the pipeline. There're many commonly adopted optimization techniques, such as inline expansion, dead code elimination, loop transformation, etc.. These techniques are organized into optimization passes and a user often has the choice to enable or disable each one on demand.

After the code is optimized, the backend is responsible of translating the IR into target machine languages. In this stage, a compiler often need to conduct instruction selection, register allocation, and instruction scheduling. This stage is also known as code generation.



This architecture is particularly powerful when you try to extend your compiler to support more source languages and hardware systems. Since your compiler is designed around a powerful IR, adding support for a new programming language simply comes down to writing a new frontend and the rest of the compiler could be reused without any effort. The same is true when you need to support another hardware platform.

Core Components

This three-stage design described in the last section is adopted by LLVM and achieves great success. And that's the reason why NNVM also resembles this architecture. Since NNVM is focused on graph level compilation, the key components of NNVM are as follows,

- A compiler frontend to compile graph definitions from different deep learning frameworks into NNVM's graph IR.
- A computational graph IR.
- A bunch of common tensor operators.
- A bunch of graph optimization passes.

The end product of NNVM will be an optimized graph IR which will further be handed down to TVM for tensor-level compilation.

Other than that, NNVM is also designed in a fashion that favors evolution. Since deep learning is a fast developing area, it is not enough to provide a fixed number of operators and passes. Users need be able to add any custom operator and optimization technique without affecting the core NNVM library. This requirement can be achieved thanks to

- An extensible operator registration system to add operators to the framework.
- An operator attribute registration system to further add arbitrary attributes to the operators.
- An optimization pass registration system to add passes.

Let's take a quick look at the source code structure of the repository,

- nnvm
  - src
    - c_api // c interface
    - compiler // nnvm graph compiler
    - core // core data structures
      - graph.cc // graph IR
      - node.cc // graph node
      - op.cc // tensor operator management
      - pass.cc // optimization pass management
      - symbolic.cc // (optional) symbolic api
    - pass // built-in passes
    - top // built-in operator

I'll now elaborate more on some key components.

The Graph IR

The IR is quite straight forward, since all deep learning networks could be modeled as a directed acyclic graph. Thus, the major facilities of the IR is located in `core/graph.cc` and `core/node.cc`.

The Node data structure represents operations in an NNVM graph.

The Graph data structure is like a manager of Nodes and contains other attributes of the computational graph and other acceleration data structures.

The Operator System and Built-in Tensor Operators

As stated in the previous section, an Operator in attached to a Node to do some computation. A set of operators are already supported by NNVM, including common neural network operators like convolution and pooling, as well as common tensor operators like matrix operation, tensor transformation and broadcasting.

However, like mentioned above, the full power of NNVM is built on top of the extensible operator registration system which enables users to add new operators in a decentralized fashion. The follow code block shows how NNVM uses its operator system to register the dense operator to its core.

```
NNVM_REGISTER_OP(dense)
.describe(R"code(Applies a linear transformation: :math:`Y = XW^T + b`.

- **data**: `(x1, x2, ..., xn, input_dim)`
- **weight**: `(units, input_dim)`
- **bias**: `(units,)`
- **out**: `(x1, x2, ..., xn, units)`

The learnable parameters include both ``weight`` and ``bias``.

If ``use_bias`` is set to be false, then the ``bias`` term is ignored.

)code" NNVM_ADD_FILELINE)
.add_argument("data", "nD Tensor", "Input data.")
.add_argument("weight", "2D Tensor", "Weight matrix.")
.add_argument("bias", "1D Tensor", "Bias parameter.")
.add_arguments(DenseParam::__FIELDS__())
.set_attr_parser(ParamParser<DenseParam>)
.set_attr<FGetAttrDict>("FGetAttrDict", ParamGetAttrDict<DenseParam>)
.set_num_outputs(1)
.set_num_inputs(UseBiasNumInputs<DenseParam>)
.set_attr<FListInputNames>("FListInputNames",
UseBiasListInputNames<DenseParam>)
.set_attr<FInferShape>("FInferShape", DenseInferShape)
.set_attr<FInferType>("FInferType", ElemwiseType<-1, 1>)
// leave weight & bias layout undefined
.set_attr<FCorrectLayout>("FCorrectLayout", ElemwiseFixedLayoutCopyToOut<1,
1>)
.set_attr<FGradient>(
  "FGradient", [](const NodePtr& n,
                  const std::vector<NodeEntry>& ograds) {
    const DenseParam& param = nnvm::get<DenseParam>(n->attrs.parsed);

    NodeEntry data_grad = MakeNode("matmul",
                                   n->attrs.name + "_data_grad",
                                   {ograds[0], n-
>inputs[DenseParam::kWeight]});
    NodeEntry w_grad_sub = MakeNode("matmul",
                                     n->attrs.name + "_weight_grad_sub0",
                                     {ograds[0], n-
>inputs[DenseParam::kData]},
                                     {{"transpose_a", "true"}});
    TShape w_reduce_axis = {0, -1};
    std::ostringstream w_oss; w_oss << w_reduce_axis;
    NodeEntry w_grad = MakeNode("sum", n->attrs.name + "_weight_grad",
                                {w_grad_sub},
                                {{"axis", w_oss.str()}, {"exclude",
"true"}});
    std::vector<NodeEntry> grads = {data_grad, w_grad};

    if (param.use_bias) {
      TShape axis = {-1};
      std::ostringstream b_oss; b_oss << axis;
      grads.push_back(MakeNode("sum", n->attrs.name + "_bias_grad",
                      {ograds[0]},
                      {{"axis", b_oss.str()}, {"exclude", "true"}}));
    }
    return grads;
```

```
})
.set_support_level(1);
```

First of all, `dense` is an operator defined in the TVM layer using basic tensors. The `NNVM_REGISTER_OP(op)` function registers this operator to the system, by providing essential attributes like arguments, number of inputs and outputs. Meanwhile, other generic attributes could also be attached to this operator using `.set_attr<YourAttr>(args)`. Notice how the gradient computation is attached to this operator by providing a lambda expression.

The Pass System and Built-in Optimization Passes

An optimization pass acts like a function on the graph, which consumes a graph and produces another graph, by transforming its structure or adding some attributes.

The built-in passes are capable of planning memory usage, pruning pre-computed parts off the graph, fusing multiple nodes into one, printing useful information about the graph, and so many else. I'll not going into algorithmic details of each pass here. Notice that in the source tree, passes are located in different folders. The `src/pass` folder contains low-level passes, well the `src/compiler` folder contains some high-level passes.

The optimization passes are registered to the system in a similar fashion like the operators.

```
NNVM_REGISTER_PASS(GraphFuseCompile)
.set_body(GraphFuseCompile);
```

Summary

In this survey, we have reviewed the basic architecture of TVM and NNVM, the basic usage of NNVM and its core internal design that powers its engine.

The TVM stack is a powerful tool to compile and deploy deep learning models from different frameworks. It's designed into layers, so that cohesion is minimum. This system is aimed at extension, so that more support for other deep learning frameworks will be added in the future.

Although as a researcher and deep learning user, I would really love a tool that could convert a model from one framework into another. Since a lot of research projects are implemented with different tools, and if we'd like to experiment with other people's research, we have to first understand their codes! That means we have to learn about all the deep learning frameworks out there, probably written with different programming languages. If NNVM and TVM could use their internal graph and tensor data structures to compile the model not only to deployable codes, but also to the same model written in another framework, then this tool will definitely appeal even more users.

References

1. https://github.com/dmlc/nnvm
2. http://tvmlang.org/2017/08/17/tvm-release-announcement.html

3. http://tvmlang.org/2017/10/06/nnvm-compiler-announcement.html
4. https://aws.amazon.com/cn/blogs/machine-learning/introducing-nnvm-compiler-a-new-open-end-to-end-compiler-for-ai-frameworks/
5. http://nnvm.tvmlang.org/index.html
6. http://docs.tvmlang.org/index.html
7. http://www.aosabook.org/en/llvm.html