Enhancing TVM with MLIR

Tony Wang
Technical University Munich

1 Motivation

Consider following problem: you are reviewing your Python code and encounter this matrix multiplication in your program; you notice that it is bottlenecking your program severely and therefore has to be heavily optimized.

```
def mat_mul(X, Y):
Z = [[0 for x in range(len(X))]
      for y in range(len(Y[0]))]
for i in range(0, len(Y[0])):
    for j in range(0, len(X)):
        for k in range(0, len(X[0])):
            Z[j][i] += X[j][k] * Y[k][i]
return Z
```

Most people will most likely revert to using the *NumPy* library, because why not. It is easy to use, reliable, safe and fast. However, what if you could not use *NumPy*, what if *NumPy* can only be used on one specific Python program and has no compatability with other Python codes at all. Then, even though there are already many implemented solutions for your problem, you have to write everything on your own again. Starting from simple tiling, to complex parallelizations and vectorizations, in the end you simply reinvented the wheel. Eventually this non-compatability could cost you thousands of engineering hours and maybe - in the end - even a worse result.

A similar issue is emerging in todays compiler industry. The list of deep learning compilers is getting longer and longer, you have for instance Facebook's Tensor Comprehensions and Glow, Google's XLA, Intel's nGraph or Apache's TVM. But TVM is no silver bullet, every compiler has their own advantages in certain aspects, but currently there is no way to share the state-of-the-art optimizations between different compilers. The problem thus is not the number of compilers, but their incompatability. They are all very inconsistent because most of them are proprietary and hence also do not share any code. For there is no common infrastructure among deep learning compilers, many reinvent their own high- and low-level IRs. For instance nGraph has nGraph IR, Tensor Comprehensions uses TC IR and Polyhedral, XLA uses HLO and TVM uses Relay and Halide. Since all these compilers have similar goals they share analogical optimizations, which, however, need to be reimplemented. MLIR (Multi-Level-Intermediate Representation) is an open-source general compiler infrastructure, which is part of the LLVM family and therefore also share the same ideology; it provides an environment for co-designing different compiler abstractions. MLIR is already used in Google's TensorFlow, ONNX, PlaidML, but also beyond machine learning, such as CIRCT. I propose to create a Relay Dialect in MLIR and connects TVM with the MLIR infrastructure.

2 Unification

MLIR does not define IRs, it can be seen as a toolkit to represent and transform compiler abstractions. Thus it is not necessary to replace an existing IR in TVM, since MLIR allows you to build a compiler abstractions on top of your IRs.

This enables TVM to choose from a bigger set of optimizations, which is imperative when targeting a wider area of hardware back-ends.

3 Optimization

Optimization Passes The optimization benefits that come along with MLIR are from being compatible and flexible across all dialects. When designing a new dialect, it is requred to add further information, such as, whether the operation has any side effects, or is a control flow terminator, etc. This allows MLIR to reuse common optimization passes that are based on properties, such as constant-folding, common-subexpression or dead-code elimination across different dialects, without having to reimplement them again. However, it is also valid to write compiler passes for specific dialects only, which can be useful if they are too complex and do not fit the broader spectrum

Dialect specific optimizations MLIR allows mixing different dialects together into one program, while still being able to preserve dialect specific optimizations.

4 Summary and Outlook

Even though we have many different deep learning compilers, we still lack the compatability between these compilers.

Here's a typical reference to a floating figure: Figure 1. Floats should usually be placed where latex wants then. Figure1 is centered, and has a caption that instructs you to make sure that the size of the text within the figures that you use is as big as (or bigger than) the size of the text in the caption of the figures. Please do. Really.

In our case, we've explicitly drawn the figure inlined in latex, to allow this tex file to cleanly compile. But usually, your figures will reside in some file.pdf, and you'd include them in your document with, say, \includegraphics.

Lists are sometimes quite handy. If you want to itemize things, feel free:

fread a function that reads from a `stream` into the array `ptr` at most `nobj` objects of size `size`, returning returns the number of objects read.

Fred a person's name, e.g., there once was a dude named Fred who separated usenix.sty from this file to allow for easy inclusion.

The noindent at the start of this paragraph in its tex version makes it clear that it's a continuation of the preceding paragraph, as opposed to a new paragraph in its own right.

4.1 LaTeX-ing Your TeX File

People often use `pdflatex` these days for creating pdf-s from tex files via the shell. And `bibtex`, of course. Works for us.

Acknowledgments

The USENIX latex style is old and very tired, which is why there's no \acks command for you to use when acknowledging. Sorry.

Availability

USENIX program committees give extra points to submissions that are backed by artifacts that are publicly available. If you made your code or data available, it's worth mentioning this fact in a dedicated section.

References

[1] Remzi H. Arpaci-Dusseau and Arpaci-Dusseau Andrea C. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, LLC, 1.00 edition, 2015. `http://pages.cs.wisc.edu/~remzi/OSTEP/`.

[2] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 181–194, 2002. `https://www.usenix.org/legacy/event/osdi02/tech/waldspurger/waldspurger.pdf`.