Contents

1 Swift for TensorFlow

- Design Doc
- Why Swift

- The main part in Swift for TensorFlow is Graph Program Extraction, which is a compiler.
- The transformation algorithm can also be used to extract ***any computation that executes asynchronously with the host program*** while communicating through sends and receives.
- This is useful for ***anything that represents computation as a graph***.
- Enable the Source Code Transformation techniques to AD.
  - The graph-based approach is hard to translate control flow constructs in the host language to the computation graph, make it challenging to perform optimizations.
- Natural interoperability between accelerated tensor operations and arbitrary non-tensor host codes.

1.1 Goals

1. Provide the best possible user experience for machine learning researchers, developers and production engineers.
2. Provide usable access to high-performance computation.
3. Eliminate the compromises that have historically forced developers to choose between performance or usability.
4. Provide a simple, predictable, and reliable programming model that is easy to intuitively understandable, and the compiler can reinforce with warnings, other diagnostics and potentially optimization techniques.

New things could be achieved if we could enhance the compiler and language.

- Allows an ML programmer to write simple imperative code using normal control flow.
- Have the compiler do the job of building a TensorFlow graph.
- Performance benefits of graph abstractions.
- Allow other compiler analysis to automatically detect bugs (like shape mismatches) in user code without even running it.
- The project goal is to improve the usability of TensorFlow
- Graph execution: performance.
- Improved usability at the ***every level of the stack***.
- Improving the usability of high-performance accelerators will enable even faster breakthroughs in ML.

1.2 Swift

- The "scripting language feel" combined with high performance is very useful for machine learning.
- Rely on the principle: Progressive Disclosure principle
  - aggressively factors the cost of complexity onto the people who benefit from that complexity.
- A lot of the Swift experience is defined in the standard library, not the compiler itself.
- *that Swift is just "syntactic sugar for LLVM". This capability is very important*.
- ***Design choices about the user experience are not baked into the language or compiler***.

1.3 Graph Program Extraction

1. The compiler ***finds the tensor operations in the code***.
2. The compiler ***desugars high-level abstractions*** (like structs, tuples, generics, functions, variables, etc) that connect tensor operations through a process called "deabstraction".
   - the tensor operations are directly connected to each other through SSA data flow edges
   - the tensor operations are embedded in a control flow graph represented in the Swift Intermediate Language (SIL).
3. ***Remove the tensor operations from the host code***:
   - A transformation called "partitioning" extracts the graph operations from the program and builds a new SIL function to represent the tensor code.
   - New calls have injected that call into the new runtime library to start up TensorFlow.
   - Rendezvous to collect any results, and send/receive values between the host and the tensor program as it runs.
4. Once the tensor function is formed, it has some transformations applied to it and is ***eventually emitted to a TensorFlow graph***.

1.4 The TensorFlow module

- One most significant design constraint is that:
  - ***we don't want users of Swift for TensorFlow to write code that accidentally causes unnecessary copies back and forth between the host and the accelerator***.
    * provide two primary concepts: "arrays" and "tensors".
    * "arrays" should be thought of as data in the host program, whereas "tensors" are values that are primarily managed by TensorFlow.

1.5 Runtime Entry Points for Extraction

1. The Graph Program Extraction algorithm splits the tensor operations out to a TensorFlow graph.
2. The TensorFlow graph is serialized to a protobuf and encoded into the program's executable.
3. The Graph Program Extraction algorithm rewrites the host code to insert calls to "start tensor program", "finish tensor program", and "terminate tensor program" runtime entry points.

   The most significant unimplemented piece of our compiler and runtime model is support for sending and receiving data between the co-executing asynchronous host and TensorFlow programs.

1.6 AD

- To develop more powerful techniques to improve user experience in failure cases:
  - enable differentiating custom data structures, recursion, and higher-order differentiation.
- As such, Swift for TensorFlow builds a stand-alone AD feature for Swift
  - entirely independent of the standard TensorFlow implementation of AD.
  - entirely independent of TensorFlow support in Swift.
- Have Swift AD support ***arbitrary user-defined types***.
  - making TensorFlow's Tensor conform to the AD system.
- Automatic differentiation in Swift is ***a compiler IR transformation*** implemented with static analysis.

  - When differentiating a function in reverse mode, the compiler produces separate functions that contain the corresponding "primal code" and "adjoint code", which in turn compute the partial derivatives of the model output with respect to the input parameters.

we plan to support full-fledged control flow and discuss the need for forward-mode AD with the community

---

2 Graph Program Extraction

In proof systems, it is well known that it is difficult to make a static analysis that is both sound and complete, meaning that you get a choice of 1) an unsound model that sometimes produces incorrect results but handles all programs, 2) a sound model that is always correct but handles a limited set of programs. Since we are using static analysis for code generation, we require the analysis to be correct! Let's explore the conservatively correct but incomplete model.

2.1 related works

- Design Doc

Machine learning models contain a mix of two different kinds of code:

- tensor number crunching logic
- other general codes

All of the approaches used by machine learning frameworks are just different ways for the system to "find" the tensor logic, extract it out, and send it to an accelerator.

2.1.1 Explicit graph building APIs

- Introduce a graph abstraction and introduce APIs for building and executing that graph
- ***Advantage***
  - Many important performance optimizations become possible once the computation is expressed as a graph
    - support for different accelerator hardware
    - distribution across multiple accelerators
- ***Downside***
  - significant usability is sacrificed to achieve optimized performance.
    - dynamic control flow is difficult to express
    - difficult to step through the codes, difficult to understand the bug and the ultimate fix. Because tack trace are produced through a bunch of runtime code users didn't write.