CS131 HW6: Language Bindings for TensorFlow

Chaoran Lin, University of California, Los Angeles

ABSTRACT

This executive summary examines the effects of using different programming languages to implement an application server proxy herd that uses TensorFlow. It first provides a brief overview of TensorFlow and its architecture, and considers three languages: Java, OCaml, and Haskell, by focusing on aspects such as ease of use, flexibility, generality, performance, and reliability. This summary is ideally intended for the reader who has some expertise in software and/or project management, but who would like a general discussion of the three languages mentioned and their role in the application.

1 Introduction

1.1 TensorFlow Architecture

TensorFlow is an open-source machine learning network that allows a developer or researcher deploy machine learning computations onto one or more CPUs or GPUs in a desktop, server, or mobile device with a single API. The architecture of TensorFlow consists of several important layers: the client, distributed master, worker services [1].

In the client layer, users can write TensorFlow programs that arrange TensorFlow operations into a graph called the computation graph, create a session, and send the graph definition to the distributed master layer.

The distributed master layer then prunes and partitions the graph into pieces and pass them on to each worker, as well as caching these pieces for future use.

The worker service in each task handles each request from the master and directs communication between tasks. A task can be of different things; for example, one task might be to store and update parameters for a machine learning model, while another task sends updates to these parameters as workers optimize these parameters.

When it comes to language bindings in TensorFlow, an important point is that new language support should be built on top of TensorFlow's C API in order to be able to utilize TensorFlow functionality [2]. In addition, the language should have enough qualities in itself that would allow a developer to create the client program with minimal headache. There is also the presence of Python, which we used to implement the application server herd architecture initially, so we must also consider whether this shift in programming language choice is prudent. With this in mind, we take a look at the various language possibilities in the rest of this report.

2 Java

2.1 Advantages

One of the key benefits of Java is its large support ecosystem in many aspects; a plethora of libraries and frameworks (including those for server-side networking and neural networks, for example), a large developer community, and several widely used IDEs for convenient code writing and debugging such as Eclipse. Additionally, Java has great portability and generality due to its cross-platform nature, as it runs on the non-OS-specific Java Virtual Machine (JVM) [3]. In terms of typing, Java is statically and strongly typed, so compile time type checking is quick and straightforward, thus providing good reliability. Finally, Java has an extensive library for implementing multithreaded programming, with control of the performance left to the native OS or the JVM the code is run on [4].

2.2 Disadvantages

Out of the three languages considered here, Java is probably the most verbose and complex in terms of syntax, and its class inheritance hierarchy is overly complicated [5], whereas the other two languages and Python excel in terms of ease of use.

Java is also rather ill-suited for bindings to C libraries, which would be required for our application. This makes sense due to the language's motivation to be portable across platforms, however in this case it presents a shortcoming. Within the current tools available, there is the Java Native Interface [6], but even that is too complicated to use, as it requires the developer to undergo a lengthy process of writing C code, compiling it into libraries, and then link and load those libraries

with Java. As TensorFlow is already exposed via a C API, the cumbersome methods of interacting between C and Java would add unwanted inconvenience to our application.

3 OCaml

3.1 Advantages

One major advantage of OCaml is its ease of use. Since it is a functional programming language, it comes with all the benefits attributed to that family of languages, such as avoidance of side effects and functions as first-class values, which greatly enhances the readability of the code as well as the convenience of writing it. However, OCaml is not confined to just functional programming style; it also supports imperative and object-oriented styles [7], so it provides a great amount of flexibility and versatility for the developer. In addition, like Java, OCaml is strongly and statically typed, with an emphasis on type inference, meaning that errors relating to types are guaranteed to be prevented, producing more reliable code in general. It also provides a garbage collector [9], another feature it shares with Java, so this takes off the burden of allocating and freeing memory. However, OCaml's garbage collector is better in terms of performance compared to its Java counterpart; it doesn't allocate huge amounts of memory at start-up, and it compacts the heap by moving memory areas around [10].

Unlike Java, OCaml provides a rather straightforward method of binding to C libraries using the ctypes library [11], thus being able to bind to TensorFlow. A developer can simply link an OCaml function to its binded C functions using the foreign keywords, without having to write extra C stub functions, so errors are less prone to happen [12].

3.2 Disadvantages

Ironically, OCaml's performance-enhanced garbage collector is also the reason behind one of its disadvantages: because its garbage collector does not allow threads to run in parallel, OCaml has poor parallelism support (like Python) [8]. OCaml's built-in thread library also does not provide enough support in this aspect. Therefore, using OCaml would limit a developer's chances to utilize parallel programming and all the performance benefits that come with it that would be useful for machine learning computations. Jane Street does currently maintain a monadic Async library that supports concurrent and event-driven programming and may provide a workaround for this problem, though OCaml's inherent shortcomings in this area are still worrying.

Another important thing to consider is that OCaml does not have great support currently. From its unpolished and relatively small standard library, to its dwindling community, it simply lacks enough popularity at the moment, meaning that the available developer count and future support for this language could be unstable.

4 Haskell

4.1 Advantages

Like OCaml, Haskell is another functional programming languages, so the same benefits of ease of use and readability are shared by Haskell [13]. It is also statically and strongly typed like the other two languages we had discussed. Beyond these advantages, Haskell also has some unique pros of its own. For one, Haskell has its own lazy or "non-strict" evaluation system [14], meaning that expressions are not evaluated until their results are needed by other computations. This presents a better outlook in terms of code efficiency and performance. Haskell also contains a wide array of domain-specific libraries such as repa for parallel array processing [15] and linear for linear algebra [16], which would be useful for the mathematical functions of machine learning. Additionally, we are seeing developments being made in creating machine learning libraries in Haskell itself, such as the fairly popular HLearn library [17].

Like OCaml, Haskell also provides an interface to produce bindings to a C library. Even the names are similar, in Haskell, this interface is called the Foreign Function Interface (FFI) [20], and it works similarly to its OCaml counterpart.

On paper, Haskell seems to share many traits with OCaml. However, there is an important distinguishing factor: Haskell has much better support for parallelism and multithreaded programming. It has a production-ready, green thread-based runtime [18] that makes it easy to implement highly concurrent network services for our application server herd model, which is perfect for our use case. It even allows for deterministic parallelism [19], which means that results in parallelized programs would be the same as their sequential counterparts.

4.2 Disadvantages

The special lazy evaluation system of Haskell makes it difficult to determine when actual computation happens, especially with an optimized compiler, since it tries to perform a lot of computation under the hood, and can lead to serious space leaks.

Like OCaml, Haskell is also very much a minority language, so finding enough developers for the application, as well as resources and support for the language, is similarly difficult. Haskell is also heavily research-oriented and driven [21]. Because of this, it changes somewhat quickly and unpredictably, and keeping up with these changes can be a hassle.

5 Conclusion

5.1 Final Comparison

Overall, all three of the languages we examined here, as well as Python, are effective to varying degrees, based on their advantages and disadvantages. In terms of ease of use and flexibility, all three of Python, OCaml, and Haskell are excellent due to their higher-level nature. Performance-wise, the optimiziations provided by the lazy evaluation system and multithreading capabilities of Haskell make it a clear winner here. However, the verbose language design and long-running support community for Java also makes it a considerable tool.

5.2 Recommendation

If I had to choose one language out of the ones available here, I would recommend Haskell for the purposes of our application, since it is an all-round solid solution, offering the benefits of functional programming (providing greater readability and ease of use over Java) along with highly polished multithreading and parallelism features (that OCaml lacks).

6 References

[1] *TensorFlow Architecture,* https://www.tensorflow.org/extend/architecture

[2] *TensorFlow in Other Languages,* https://www.tensorflow.org/extend/language_bindings

[3] *How is Java Platform Independent?,* https://www.geeksforgeeks.org/java-platform-independent/

[4] *Java Multithreading*, https://www.tutorialspoint.com/java/java_multithreading.htm

[5] *Java Drawbacks,* https://way2java.com/java-introduction/java-drawbacks/

[6] *Getting Started with Java Native Interface*, http://web.archive.org/web/20120419230023/http://java.sun.com/docs/books/jni/html/start.html

[7] *OCaml,* http://ocaml.org/

[8] *The Rise and Fall of OCaml*, http://flyingfrogblog.blogspot.com/2010/08/rise-and-fall-of-ocaml.html

[9] *Garbage Collection,* http://www.ffconsultancy.com/ocaml/benefits/garbage_collection.html

[10] *Garbage Collection,* https://ocaml.org/learn/tutorials/garbage_collection.html

[11] *OCaml c-types,* https://github.com/ocamllabs/ocaml-ctypes

[12] *Chapter 19. Foreign Function Interface,* https://realworldocaml.org/v1/en/html/foreign-function-interface.html

[13] *In Praise of Haskell*, http://www.drdobbs.com/architecture-and-design/in-praise-of-haskell/240163246

[14] *Why Haskell Matters,* https://wiki.haskell.org/Why_Haskell_matters#What_can_Haskell_offer_the_programmer.3F

[15] *Repa: High Performance, regular, shape polymorphic parallel arrays,* https://hackage.haskell.org/package/repa

[16] *Linear: Linear Algebra,* https://hackage.haskell.org/package/hmatrix

[17] *HLearn,* https://github.com/mikeizbicki/HLearn

[18] *Concurrent and multicore programming,* http://book.realworldhaskell.org/read/concurrent-and-multicore-programming.html

[19] *Parallelism,* https://wiki.haskell.org/Parallelism

[20] *Interfacing with C: the FFI* http://book.realworldhaskell.org/read/interfacing-with-c-the-ffi.html

[21] *Haskell in Research,* https://wiki.haskell.org/Haskell_in_research