Alternative Language Implementations of Tensorflow
Java, OCaml, and Swift

Robathan Harries - University of California, Los Angeles

Abstract

Machine learning algorithms usually operate on large training sets and large inference queries. Some applications, however, use machine learning algorithms to handle many small queries requiring work with relatively small models. If the graphs making up the models are small enough, the main bottleneck in system performance may be the overhead of setting up the graphs making up these models, rather than the usually intensive computations on those models. This problem is compounded when the machine learning framework being used is implemented in a language like Python, which is less focused on efficiency than other programming languages. In this study, I will investigate the performance and productivity implications of implementing a machine learning server-herd application, exhibiting the previously-described problem, with Tensorflow bindings to various languages: Python, Java, OCaml, and Swift.

1. Introduction

Server Herds are very good at managing frequent connections from mobile clients, who can connect to whichever server is closest. In such applications, these frequent connections are often used to transmit relatively small queries. When these small queries are applied to a Tensorflow machine learning framework, performance may be negatively affected, as the overhead of setting up machine learning models is greater than the actually computing the result of a small query.

Most Tensorflow applications have bottlenecks inside the C++ or CUDA code which handles the real computational load of a machine learning model. In this situation, where a server-herd application receives many small queries which it must pass through a machine learning model, the execution bottleneck is often in the Tensorflow client code, which sets up an environment in which to run the core Tensorflow C++ or CUDA code[6].

In this study, I will investigate whether this problem can be avoided by implementing this application with Tensorflow client code in a different language.

2. Python

Python began as a portable scripting language with many conveniences for developers, such as dynamic strong typing, interpretative execution, automatic memory management, and support for programming paradigms from object-oriented to functional[5].

2.1 Python - Advantages

Most of Python's advantages are directed towards speeding up the prototyping and development process. Python uses duck typing, in which variable containers are extremely flexible and general, implicitly determining object types at runtime. This allows developers to save time, because they can skip the overhead of explicitly declaring the type of every Tensorflow object used to set up graphs for machine learning models.

In addition, Python has automatic memory management, which saves developers time and effort, as they can largely ignore issues of allocating and freeing memory. For example, when models from old queries are no longer used, their memory will automatically be deallocated from the heap, saving developers the significant effort of collecting all relevant memory from Tensorflow worker service objects[6].

Python also has extensive support for numerous open-source libraries, like asyncio and aiohttp, which, along with the previous advantages, make prototyping a machine-learning server-herd which accepts asynchronous connections on TCP and HTTP fast and relatively painless.

2.2 Python - Disadvantages

By focusing on easing the development process, Python has to make some sacrifices in terms of performance. Interpretative execution, dynamic typing, and automatic memory management all create more runtime overhead, slowing down the setup of machine learning models.

Python's automatic memory management maintains reference counts for every object, and performs periodic sweeps over the heap to deallocate unreferenced objects. These sweeps cannot operate on another thread, due to Python's Global Interpreter Lock, which prevents different threads from concurrently accessing shared data. In addition, garbage collection through reference counts cannot catch and deallocate reference cycles (unreferenced objects which reference each other), which is a significant weakness. As such, this method of garbage collection is inefficient and less robust.

In addition, since the Python interpreter uses dynamic typing while retaining strong typing principles, the interpreter must infer variable types at runtime, wasting valuable CPU cycles and reducing throughput. Since Python uses interpretative execution, it must do more work at runtime than languages like C++ and CUDA, which do much of this work at compile time.

All of these factors, which make developing in Python incredibly efficient, may cause a bottleneck in the Python client code which sets up Tensorflow core functionality.

3. Java

Java is an object-oriented programming language, developed to be very portable, with extensive support for multithreading and concurrency. Java code compiles to an intermediate interpretable bytecode which runs on the Java Virtual Machine. In addition to extensive multithreading libraries, Java's distinct features include generational garbage collection and static typing.

3.1 Java - Advantages

One of Java's greatest strengths is its portability. As it executes through the JVM (Java Virtual Machine), the same application program can run on servers with different architectures, which is a great advantage for implementing large server-herds.

Executing on the JVM also gives Java many performance benefits over Python. Compiling to bytecode before execution eliminates the need to do as much runtime work as Python must do. A large part of this is Java's static typing, which performs type-checking at compile time, allowing the JVM to avoid the overhead of inferring variable types at runtime.

Java's generational garbage collection method is both more robust and more efficient than Python's. It doesn't rely on reference counts, instead sweeping through all accessible parts of the heap and deallocating all unreachable memory objects. Because of this, it can't be fooled by reference cycles. In addition, the garbage collection process occurs concurrently on a separate thread, making it much more efficient than Python's sequential garbage collection.

Java's extensive support for multithreading also makes it a perfect fit for developing a performant server herd and Tensorflow client. In fact, Java's NIO2 API os almost as suitable as Python's asyncio library for implementing the server-herd. With it, one can easily build a server which accepts asynchronous TCP connections, handling them on an event loop[2]. With multithreading, it's possible to process requests asynchronously on separate threads, increasing performance.

3.2 Java - Disadvantages

Naturally, Java's performance benefits over Python come at the expense of development speed. With static-typing, developers must know the exact types of the data being exchanged by the various components of the constructed model graph. Though being explicit about the data types being sent between parts of the application can be useful when developing large projects, it drastically slows down the prototyping process.

In general, Java takes a middle ground between performance and programming ease, with partial compilation and partial interpretative execution.

Though it is extremely well-suited for implementing an asynchronous server-herd, with support for parallelization to increase performance, it is not clear that Java is the best option for our situation.

4. OCaml

OCaml is a modern version of ML which, while strongly reminiscent of ML's functional programming style, supports imperative and object-oriented programming. In addition, OCaml has an extensive standard library which includes asynchronous I/O, making it a possible candidate for programming a server-herd application.

4.1 OCaml - Advantages

OCaml's strongest advantage lies in its performance, though it still manages to maintain some ease-of-use features just like Python. Like Python, OCaml infers variable types, but does so statically at compile-time. In addition, OCaml's static program analysis optimizes value boxing and closure allocation, allowing it to efficiently compile functional programs into machine code[4].

OCaml's memory management system strikes a balance between Python's and Java's methods. It mixes generations and reference counts in its garbage collection, and the result displays better performance than either Java or Python.

Also, OCaml's extensive standard library includes an asynchronous networking library Lwt, which is strikingly similar to Python's asyncio, with extra support for HTTP as well as TCP. This makes it a good fit for implementing at least the server-herd parts of our application

Finally, OCaml's functional programming paradigm is very suitable for prototyping machine learning models, which essentially optimize the compositions of functions on immutable data structures. OCaml performs best with functional implementations, which makes it a great language for implementing a Tensorflow client[7]. In concert with its asynchronous networking library, this makes OCaml uniquely suited to implement our server-herd Tensorflow client code.

4.2 Ocaml - Disadvantages

Naturally, the performance benefits of working with OCaml come with significant drawbacks. Its functional programming paradigm is harder to learn than imperative programming, and the combination of static-typing and type-inferring makes compile-time errors extremely frequent, forcing the developer to be very aware of variable types without explicitly declaring them in the program.

Furthermore, OCaml is less portable than Java and Python, since it compiles into native machine code. As such, a server-herd with different server architectures will have to run different versions of the application on different machines.

Finally, just like Python with its Global Interpreter Lock, OCaml doesn't allow multiple concurrent accesses to a single object. As such, it cannot exhibit true parallelism, limiting the performance of servers in our server-herd.

5. Swift

Swift is a relatively new programming language released by Apple for iOS development. As its name implies, Swift focuses on achieving good performance, which is very important for applications on phones and other small devices. The language is open-source, and with support from a huge corporation such as Apple, it has gained a thriving developer community.

5.1 Swift - Advantages

Swift's primary advantage, like OCaml, is its performance. Designed to be used on phones and other iOS devices with strictly limited memory, Swift has many features which optimize its performance. For example, Swift uses strong static-typing, decreasing the amount of work it must do at runtime. Generally, Swift performs 8.4 times as fast as Python[3].

Even with its performance focus, Swift also displays many features increasing the ease of development. Like Python, it has clear and concise syntax, and performs automatic memory management using reference counts. Like in Python, these features increase developer

productivity and allow developers to avoid accidental performance bugs from mismanaging allocated memory.

With a large open-source developer base, Swift has libraries for asynchronous networking, making it a worthwhile candidate for implementing this server-herd Tensorflow client.

5.2 Swift - Disadvantages

Like Python, Swift's ease-of-use considerations require some performance sacrifices. Swift's reference counting is generally slower than OCaml's hybrid memory management system, and can still fall prey to reference cycles. Swift does include methods to help developers avoid reference cycles, such as 'weak' object references which don't count towards an object's reference count, but in the end it is still up to the developer to avoid costly reference cycle bugs[1].

In addition, because Swift is still a new programming language, it is constantly undergoing changes, sometimes even breaking backwards compatibility. This also means that information on Swift and experienced Swift developers are not as widespread as with other languages.

6. Conclusion

All of these languages come with certain advantages and disadvantages. The biggest difference between these languages is where they lean in the tradeoff between performance and ease-of-use. Python focuses mainly on being easy to use, with dynamic type-inferring and interpretative execution, whereas OCaml does extensive compile-time optimization. Java and Swift fall somewhere in the middle of this distinction.

As such, the best language choice depends entirely on our use-case. Sometimes quick prototyping and development is paramount, and sometimes we need to squeeze all the performance we can out of a machine. Since in this case we wish to reduce our code's overhead, we fall into the latter category, and must go with the language with the best throughput for both asynchronous networking and setting up Tensorflow models.

As such, OCaml is the best language to use for implementing Tensorflow client code in a server-herd application. Its functional programming style fits well with setting up machine learning models, and it has a great asynchronous networking library. Ocaml's memory management system is faster than those of the other languages considered, and its compile-time optimization allows it to achieve the best performance out of all the languages considered.

Therefore, I can give the informed recommendation to use OCaml to prototype this specific application with reduced client-code overhead.

References

[1] Apple Inc. "Automatic Reference Counting" The Basics - The Swift Programming Language (Swift 4.2), docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html.

[2] Baeldung. "A Guide to NIO2 Asynchronous Socket Channel." Baeldung, 26 Oct. 2018, www.baeldung.com/java-nio2-async-socket-channel.

[3] "The Good and the Bad of Swift Programming Language." AltexSoft, 28 Mar. 2018, www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-swift-programming-language/.

[4] "OCaml." Wikipedia, Wikimedia Foundation, 21 Nov. 2018, en.wikipedia.org/wiki/OCaml.

[5] "Python (Programming Language)." Wikipedia, Wikimedia Foundation, 4 Dec. 2018, en.wikipedia.org/wiki/Python_%28programming_language%29.

[6] "TensorFlow Architecture | TensorFlow." TensorFlow, 30 Nov. 2018, www.tensorflow.org/extend/architecture.

[7] Xu, Joyce. "Functional Programming for Deep Learning – Towards Data Science." Towards Data Science, Towards Data Science, 29 June 2017, towardsdatascience.com/functional-programming-for-deep-learning-bc7b80e347e9.