Alternative Language Bindings for Tensorflow

Austin Guo – University of California, Los Angeles

Abstract

Applications involving machine learning algorithms can often expect to accept substantially large training sets or large inference queries. However, applications with different requirements may end up handling small queries that create or inference on extremely small models by machine learning standards. In this scenario, the overhead of setting up the graphs making up the models may be more than the cost of running the expensive computational algorithms on the graphs, especially so if the graphs are set up using a less efficiency-oriented language such as Python. In our study we investigate the performance implications of an alternative implementation of our application's TensorFlow client in Java, OCaml, and Rust, as well as the suitability of such a language in implementing an application server herd relying heavily on TensorFlow.

1. Introduction

Application server herds are especially efficient at managing frequent connections from extremely mobile clients. However, brevity is an intrinsic quality of such connections; it is not uncommon to receive extremely small queries in each connection. In an application server herd making extensive use of machine learning algorithms through the TensorFlow framework, such small queries may have negative performance implications for constructing or executing models, especially if models are prototyped in a less efficiency-oriented language such as Python.

In such a prototype, although typical TensorFlow applications bottleneck inside C++ or CUDA code, applications handling frequent small queries with models constructed from less efficient code may find their execution bottlenecked in TensorFlow client code rather than the code performing the real computational work[6]. We investigate whether using another language to develop our application may be more suitable.

2. Python

Python was originally developed as a portable scripting language with powerful developer productivity features such as dynamic strong typing, interpretative execution, automatic memory management, and support for multiple programming paradigms, including object-oriented style, imperative, functional, and procedural[3].

2.1 Advantages

Not only does Python have extensive support for standard open source libraries such as asyncio and aiohttp which make it extremely easy to prototype a server accepting asynchronous connections based on various transport protocols like TCP and HTTP, but it also makes development of machine learning models quick and painless due to dynamic typing and automatic memory management – a benefit more valuable than it may appear to be, since most data scientists will have more experience designing models and less experience prototyping such models in code.

Moreover, with Python's dynamic "duck" typing, developers can skip the overhead of typing every TensorFlow object used when setting up graphs for machine learning models. This can save lots of development time while also making the product code more compact. Further, when models from previous queries are not being used anymore, their memory does not have to be explicitly freed from the heap, saving developers the effort of collecting all relevant memory from TensorFlow worker service objects[6].

2.2 Disadvantages

Python's dynamic typing, interpretative execution scheme, and memory management model rooted in reference counts all contribute to overhead slowing down the setup of machine learning models. In general, all benefits must have tradeoffs, and Python trades off memory and computational performance for the aforementioned ease-of-development features. In terms of memory management, Python keeps track of object reference counts, and performs periodic linear time sweeps over the heap to clean up objects that aren't referenced anymore. The frequency of these linear sweeps over the entire heap makes garbage collection inefficient, and the fact that reference counts allows for reference count cycles makes Python garbage collection less robust.

Further, the fact that Python uses dynamic typing but retains strong typing principles means that Python must infer variable types at runtime, wasting runtime CPU cycles and reducing efficiency. Though this made it easier to set up programs such as a TensorFlow client setting up machine learning models, this could cause a bottleneck in our application in Python code

rather than C++ or CUDA code. Similarly, the interpretive execution eats up runtime cycles and reduces performance in our server herd.

3. Java

Java is a language made to be ultraportable and object-oriented, with strong support for concurrency. As a result, all code in Java must be mostly imperative and reside in a class, while compiling to an intermediate interpretable bytecode that can be run on the Java Virtual Machine (JVM). Java also has a few other key features that make it distinct, such as its generational garbage collection, static typing, and extensive libraries for concurrency, multithreading and synchronization.

3.1 Advantages

Given Java's strength in concurrency and object-oriented model, Java appears to be very fitted to implement a performant application server herd and TensorFlow client. In fact, the Java NIO2 API is almost just as suitable as Python's asyncio in implementing the application server herd. Using NIO2, we are able to make use of the Asynchronous Socket Channel to easily build a server that accepts TCP connections and handles them as Future objects, which are placed on a server event loop as a callback to guarantee execution in the future[1]. In this sense, Java's extensive libraries for concurrency such as asynchronous I/O make it a suitable language for our application server herd.

Further, since Java executes through the JVM, it is extremely portable. As a result, if our application server herd has several servers with different architectures, the same server application can still run on each server without modification. This portability pairs well with the performance benefits of Java compared to Python. Java's method of compiling source into bytecode before interpretation greatly decreases the runtime interpretation work Python must do on direct source to executable. In the same vein, Java's static strong typing allows Java to perform compile-time checks for typing, saving many runtime CPU cycles. It also happens that Java's generational garbage collection is much more efficient than Python's. Additionally, multithreading also plays a large role in allowing true parallelization of our application server herd: using Java, it's possible to process requests asynchronously and handle them on separate threads for increased performance.

3.2 Disadvantages

Java's static typing makes it much more difficult to quickly prototype machine learning models in TensorFlow, as it becomes the developer's responsibility to be much more attentive as to what type of data members are being exchanged by the various components of the constructed model graph, and to make sure that all type statements agree. In addition, due to Java's object-oriented nature making asynchronous HTTP requests in Java is much more involved than in Python. Rather than simply creating a session in aiohttp and issuing a GET request on a URL using the aiohttp API's format, in Java a Request object must be made implementing Callable, customized with the URL, and executed concurrently by using a Thread Pool. This manner of implementation is much more low level and perhaps more powerful than Python's, but is more complicated and does not seamlessly interface with the NIO2 API unlike asyncio and aiohttp do.

Also, though Java is extremely suited for developing server side applications such as our application server herd and takes a middle ground between optimal performance and simplicity of development, Java is not as performant as OCaml or Rust. Further investigation into the implementation of OCaml and Rust will reveal the nuances that make our statement true.

4. OCaml

OCaml is a modern implementation of ML that has strong roots in ML's functional style, but also supports imperative and object-oriented paradigms, which make it suitable for large-scale software engineering. Further, OCaml has an extensive standard library that offers support for asynchronous I/O as our application server herd requires.

4.1 Advantages

One of OCaml's strongest advantages is its performance, despite its balance of ease-of-use features like Python's. For example, OCaml uses a type-inferring system like Python, but statically checks types at compile time. This is performed at compile time after OCaml's compiler performs static program analysis to optimize value boxing and closure allocation, which allows OCaml to efficiently compile functional source into machine code[2]. In this respect, OCaml's typing system is very much like Java's, with the exception that OCaml's compilation converts source into machine code which can be executed directly after being optimized for a particular machine's architecture.

In addition, OCaml's functional programming style is very suitable for machine learning prototyping; machine learning models essentially optimize

compositions of functions over an immutable data structure such as a matrix, so we note that this functional nature of machine learning favors OCaml's preference for functional implementations, and by natural extension OCaml is a very suitable language for implementing a TensorFlow client[7].

OCaml also has an extensive standard library that includes an asynchronous networking library Lwt, which is very similar to Python's asyncio and even offers support for HTTP in addition to TCP. Memory management in OCaml also implements a performant hybrid of generational and reference counting garbage collection that is more performant than Java's and Python's.

4.2 Disadvantages

Unfortunately, OCaml's functional programming style makes it slightly harder to learn than imperative, and its static typing combination with type-inferencing makes compile time type-check problems extremely frequent and annoying. This combination of features also restricts programmers to the constraints imposed by the type system, which requires careful deliberation on the programmer's part when developing.

Further, though OCaml's compilation into native machine code greatly increases the execution speed at runtime and allows more optimizations to occur when converting from source to machine code, this compilation method makes OCaml less portable than Java (which keeps a level of interpretation on the JVM for portability). In addition, OCaml has a feature similar to Python's Global Interpreter Lock that doesn't allow multiple accesses to a single object, which effectively bars it from exhibiting true parallelism. This limits how optimal each server in our server herd can be.

5. Rust

Rust is an extremely performant concurrent systems language that prevents segfaults and guarantees thread safety. It boasts a core set of features that simultaneously emphasize reliability, performance, and ease of development.

5.1 Advantages

Before beginning any analysis of Rust's features, it may be worth noting that Rust consistently beats Java, OCaml, and Python in performance benchmarks in various algorithm implementations, often by 4-5 times. This performance boost can be attributed to its strong concurrency support, strong static typing, and compile-time optimizations for native machine code; Rust reaps similar benefits to OCaml in terms of runtime speed due to static typing and compilation into machine language, while offering the concurrency support of Java. Further, Rust does not perform any automatic memory management but rather performs memory safety checks at compile time, keeping runtime efficiency comparable to C while attaining complete memory safety.

Rust's syntax draws heavily from both C and ML, which makes it fairly easy for most programmers who know C to learn, while reaping the performance benefits of functional programming. As mentioned before, this functional nature also makes Rust particularly suitable for machine learning prototyping.

Though Rust is still a very new language, there are already numerous libraries implementing asynchronous server components. In combination with Rust's efficiency, support for highly concurrent systems, complete memory safety, and functional nature, Rust is perfectly suitable for our application server herd and machine learning prototyping in TensorFlow.

5.2 Disadvantages

Though Rust has numerous benefits that make it an extraordinary fit for our application, it is very low level and has many features. As a result, being able to use the full range of functionality of the language will be very difficult and require a steep learning curve. Further, Rust requires manual memory management, which is inconvenient for the developer.

6. Conclusion

Each language exhibits some favorable characteristic at the expense of another. In terms of performance, OCaml and Rust both have advantages over Java and Python due to their machine code compilation, static type checking and compiler optimizations – though less so with OCaml since it isn't concurrent. However, Python and Java tend to be more portable and easier to use. Thus, the most suitable language is heavily contingent on the use case of the application. In our study, our main goal is to prevent bottleneck in model setup code, and as a result language performance is the most important characteristic. Rust is our most performant language, can support event-driven servers and even allow us to make our application server herd highly concurrent, and also guarantees memory safety. Though it may be slightly harder to get company developers trained in the language, this overhead is insignificant considering developers are at the caliber of prototyping machine learning models at scale in an application server herd. We can therefore make an informed decision to use Rust to prototype our application.

References

[1] “A Guide to NIO2 Asynchronous Socket Channel.” *Baeldung*, 4 Feb. 2018, www.baeldung.com/java-nio2-async-socket-channel.

[2] “OCaml.” *Wikipedia*, Wikimedia Foundation, 12 Mar. 2018, en.wikipedia.org/wiki/OCaml.

[3] “Python (Programming Language).” *Wikipedia*, Wikimedia Foundation, 12 Mar. 2018, en.wikipedia.org/wiki/Python_(programming_language).

[4] Ronacher, Armin. “Rust for Python Programmers.” *Armin Ronacher's Thoughts and Writings*, 27 May 2015, lucumr.pocoo.org/2015/5/27/rust-for-pythonistas/.

[5] “Rust (Programming Language).” *Wikipedia*, Wikimedia Foundation, 13 Mar. 2018, en.wikipedia.org/wiki/Rust_(programming_language).

[6] “TensorFlow Architecture | TensorFlow.” *TensorFlow*, 2 Mar. 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.

Appendix

Figure 1. Language Benchmarks on Mandelbrot Test

Language	sec	mem	cpu
Rust	1.99	13,496	7.94
OCaml	55.18	2,900	55.17
Java	6.04	76,528	23.34
Python	225.24	15,736	899.25

Figure 2. Java NIO2 Server

```java
import java.nio.channels.*;

public static void main(String[] args) {
    AsyncEchoServer server = new AsyncEchoServer();
    server.runServer();
}

// inside our AsyncEchoServer
AsynchronousServerSocketChannel server
  = AsynchronousServerSocketChannel.open();
server.bind(new InetSocketAddress("127.0.0.1", 4555));
AsynchronousSocketChannel worker = acceptFuture.get(10, TimeUnit.SECONDS);

public void runServer() {
    clientChannel = acceptResult.get();
    if ((clientChannel != null) && (clientChannel.isOpen())) {
        while (true) {
            ByteBuffer buffer = ByteBuffer.allocate(32);
            Future<Integer> readResult  = clientChannel.read(buffer);

            // perform other computations
            readResult.get();

            buffer.flip();
            Future<Integer> writeResult = clientChannel.write(buffer);

            // perform other computations
            writeResult.get();
            buffer.clear();
        }
        clientChannel.close();
        serverChannel.close();
    }
}
```

We see this server setup is very similar to asyncio

Figure 3. Python vs Rust Code

```python
from threading import Lock, Thread

def fib(num):
    if num < 2:
        return 1
    return fib(num - 2) + fib(num - 1)

def thread_prog(mutex, results, i):
    rv = fib(i)
    with mutex:
        results[i] = rv

def main():
    mutex = Lock()
    results = {}

    threads = []
    for i in xrange(35):
        thread = Thread(target=thread_prog, args=(mutex, results, i))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()

    for i, rv in sorted(results.items()):
        print "fib({}) = {}".format(i, rv)
```

Python

```rust
use std::sync::{Arc, Mutex};
use std::collections::BTreeMap;
use std::thread;

fn fib(num: u64) -> u64 {
    if num < 2 { 1 } else { fib(num - 2) + fib(num - 1) }
}

fn main() {
    let locked_results = Arc::new(Mutex::new(BTreeMap::new()));
    let threads : Vec<_> = (0..35).map(|i| {
        let locked_results = locked_results.clone();
        thread::spawn(move || {
            let rv = fib(i);
            locked_results.lock().unwrap().insert(i, rv);
        })
    }).collect();
    for thread in threads { thread.join().unwrap(); }
    for (i, rv) in locked_results.lock().unwrap().iter() {
        println!("fib({}) = {}", i, rv);
    }
}
```

Rust