Understanding Tensorflow 2 source code

D. Gueorguiev 4/27/20

# Building the tensorflow libraries using Bazel

Building with debug symbols:

bazel build --config=opt --verbose\_failures -c dbg --strip=never //tensorflow:libtensorflow\_cc.so

bazel build --config=opt --verbose\_failures -c dbg --strip=never //tensorflow:libtensorflow\_framework.so

# Build Folder structure

ROOT\_FOLDER = /opt/tensorflow

Inside ROOT\_FOLDER/bazel-tensroflow/external :



By default tensorflow master as of 4/26/2020 installs the following external package dependencies:

aws double\_conversion local\_config\_git

aws-c-common eigen\_archive local\_config\_python

aws-c-event-stream farmhash\_archive local\_config\_rocm

aws-checksums fft2d local\_config\_sycl

bazel\_tools gif local\_config\_tensorrt

boringssl highwayhash nasm

com\_google\_absl jsoncpp\_git nsync

com\_google\_protobuf libjpeg\_turbo snappy

com\_googlesource\_code\_re2 local\_config\_cc zlib

curl local\_config\_cuda

**BoringSSL:** <https://boringssl.googlesource.com/boringssl/>

BoringSSL is a fork of OpenSSL that is designed to meet Google's needs.

Although BoringSSL is an open source project, it is not intended for general use, as OpenSSL is. We don't recommend that third parties depend upon it. Doing so is likely to be frustrating because there are no guarantees of API or ABI stability.

Programs ship their own copies of BoringSSL when they use it and we update everything as needed when deciding to make API changes. This allows us to mostly avoid compromises in the name of compatibility. It works for us, but it may not work for you.

BoringSSL arose because Google used OpenSSL for many years in various ways and, over time, built up a large number of patches that were maintained while tracking upstream OpenSSL. As Google's product portfolio became more complex, more copies of OpenSSL sprung up and the effort involved in maintaining all these patches in multiple places was growing steadily.

Currently BoringSSL is the SSL library in Chrome/Chromium, Android (but it's not part of the NDK) and a number of other apps/programs.

**FarmHash:** <https://github.com/google/farmhash>

### *Introducing FarmHash*

#### Monday, March 31, 2014

*We’re pleased to announce the new [FarmHash](http://code.google.com/p/farmhash/" \t "_blank) family of hash functions for strings.  FarmHash is a successor to [CityHash](http://code.google.com/p/cityhash/" \t "_blank), and includes many of the same tricks and techniques, several of them taken from Austin Appleby’s [MurmurHash](https://code.google.com/p/smhasher/" \t "_blank).  
  
We’re heavily influenced by the types of CPUs that are common in Google’s datacenters, but FarmHash’s goals don’t end there. We want FarmHash to be fast and easy for developers to use in phones, tablets, and PCs too. So, yes, we’ve improved on CityHash64 and CityHash32 and so on.  But we’re also catering to the case where you simply want a fast, robust hash function for hash tables, and it need not be the same on every platform. To that end, we provide sample code that has one interface harboring multiple platform-specific implementations.  
  
Over time, we plan to expand FarmHash to include hash functions for integers, tuples, and other data. For now, it provides hash functions for strings, though some of the subroutines could be adapted to other uses.  
  
Overall, we believe that FarmHash provides high-performance solutions to some classic problems. Please give it a try! Contributions and bug reports are most welcome.*

**HighwayHash**: <https://github.com/google/highwayhash>

Hash functions are widely used, so it is desirable to increase their speed and security. This package provides two 'strong' (well-distributed and unpredictable) hash functions: a faster version of SipHash, and an even faster algorithm we call HighwayHash.

SipHash is a fast but 'cryptographically strong' pseudo-random function by Aumasson and Bernstein [<https://www.131002.net/siphash/siphash.pdf>].

HighwayHash is a new way of mixing inputs which may inspire new cryptographically strong hashes. Large inputs are processed at a rate of 0.24 cycles per byte, and latency remains low even for small inputs. HighwayHash is faster than SipHash for all input sizes, with 5 times higher throughput at 1 KiB. We discuss design choices and provide statistical analysis and preliminary cryptanalysis in <https://arxiv.org/abs/1612.06257>.

Inside ROOT\_FOLDER/bazel-tensorflow/external/zlib

![A close up of a screen

Description automatically generated]()

# Protobuf interfaces and formats

## Core/Protobuf/Config.proto

<https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/protobuf/config.proto>

This interface contains various options:

1. for tuning the resources occupied by the GPU (see message [GPUOptions](https://github.com/dimitarpg13/tensorflow/blob/master/tensorflow/core/protobuf/config.proto#L18-L194)).

Inside GPUOptions there are various experimental configuration options such as per virtual device memory limit, memory-specific options, kernel-specific timing and memory parameters (see message [Experimental](https://github.com/dimitarpg13/tensorflow/blob/master/tensorflow/core/protobuf/config.proto#L100-L188)).

1. Optimizer tuning parameters (see message [OptimizerOptions](https://github.com/dimitarpg13/tensorflow/blob/master/tensorflow/core/protobuf/config.proto#L197-L242)). For instance there is an option for selecting the level at which the Optimizer works where level L1 denotes common subexpression elimination and constant folding. a special option for turning on the internal Just-in-time compiler and selecting how aggressive the auto-compilation should be.
2. Various graph options (see message [GraphOptions](https://github.com/dimitarpg13/tensorflow/blob/master/tensorflow/core/protobuf/config.proto#L244-L289)). For instance when to build a cost model for the nodes in the graph in terms of memory and cpu resource consumption, parameters controlling various aspects of the graph construction and updates
3. Thread pool tuning parameters and options

# Appendix A:

UNDERSTANDING HASH FUNCTIONS

by Geoff Pike

Version 0.2 --- early draft --- comments and questions welcome!

References appear in square brackets.

1 INTRODUCTION

Hashing has proven tremendously useful in constructing various fast

data structures and algorithms. It is typically possible to simplify

the analysis of hash-based algorithms if one assumes that the relevant

hash functions are high quality. At the other extreme, if the

relevant hash functions were always to return the same value, many

hash-based algorithms become algorithms that are slower, simpler, but still well-known.

For example, a chaining hash table devolves into a linked list.

There are many possible definitions of hash function quality. For

example, one might want a list of keys and their hashes to provide no

pattern that would allow an opponent to predict anything about the

hashes of other keys. Although I cannot prove it, I think I can meet

this and many other definitions of quality with

f(s) = SHA-3(concatenation of z and s),

where z is some secret string known only to me. This well-known trick

provides, I think, more high-quality hash functions than anyone will

need, though greater computational power in the future may push us to

replace SHA-3 from time to time.

In short, discussions about choosing a hash function are almost always

discussions about speed, energy consumption, or similar. Concerns

about hash quality are easy to fix, for a price.

2 ANATOMY OF A HASH FUNCTION

Hash functions that input strings of arbitrary length are written in

terms of an internal state, S. In many cases the internal state is a

fixed number of bits and will fit in machine registers. One generic

sketch of a string hash is:

let S = some initial value

let c = the length of S in bits

while (input is not exhausted) {

let t = the next c bits of input (padded with zeroes if less than c remain)

S = M(S xor t)

}

let n = the number of bytes hashed

return F(S, n)

where M is a hash function that inputs and outputs c bits, and F is a

hash function that inputs c bits (plus, say, 64 for its second argument)

and outputs however many bits one needs to return. In some sense we have

reduced the string-hashing problem to two integer hashing problems.

2.1 INTEGER HASHING TECHNIQUES

A hash function that inputs and outputs the same number of bits, say,

32, can use reversible bit-twiddling operations, each of which is

"onto" in the mathematical sense. For example, multiplication by an

odd constant is reversible, as all odd numbers are relatively prime to

2^32. Other commonly used reversible operations include:

o Adding or xoring a constant

o Bitwise rotation or other bitwise permutations

o bit j = (bit j) xor (bit k) for unequal constants j and k

o "Shift mix": S = S xor (S >> k), where k is, say, 17

o Replacing a fixed-length bit string with its cyclic redundancy

checksum, perhaps via \_mm\_crc32\_u32(f, <some constant>) [Pike]

Each of the above is a "bad" hash function that inputs and outputs

the same number of bits. One can simply compose two or more of those

bad hash functions to construct a higher-quality hash function.

One common quality goal for integer hashing (and string hashing) is

that flipping the 19th bit, or any other small change, applied to

multiple input keys, causes a seemingly unpredictable difference each

time. Similarly, any change to an input should lead to a seemingly

unpredictable selection of the output bits to flip.

Therefore, if we want a high-quality hash function that inputs c bits

and outputs fewer than c bits, we can simply truncate the output of a

high-quality hash function that inputs and outputs c bits.

To give a concrete example, here is Bob Jenkins' mix(), published in

1996 [Jenkins]. Its input is 96 bits in three 32-bit variables, and its output

is 96 bits. However, one may use a subset of the output bits, as every

output bit is affected by every non-empty subset of the input bits.

Input: a, b, and c

Algorithm:

a -= b; a -= c; a ^= (c>>13);

b -= c; b -= a; b ^= (a<<8);

c -= a; c -= b; c ^= (b>>13);

a -= b; a -= c; a ^= (c>>12);

b -= c; b -= a; b ^= (a<<16);

c -= a; c -= b; c ^= (b>>5);

a -= b; a -= c; a ^= (c>>3);

b -= c; b -= a; b ^= (a<<10);

c -= a; c -= b; c ^= (b>>15);

Output: a, b, and c

2.2 VARIATIONS ON STRING HASHING

There are three variations on our initial sketch worth noting.

First, for speed, one can special-case short inputs, as the CityHash

and FarmHash algorithms do. The number of special cases can be

reduced by using loads that may overlap: for example, a hash of a 9-

to 16-byte string can be implemented by a hash that inputs two 8-byte

values (the first 8 and last 8 bytes of the input string) and the string

length [CityHash, FarmHash].

Second, one may choose different means of incorporating input bits

into the internal state. One example: the mixing of S and input bits

may be interleaved with the mixing of parts of S and other parts of S.

Another example: the input bits processed in a loop iteration might be

xor'ed into multiple places in S, rather than just one, or might be

hashed with each other before touching S [Murmur]. The advantages and

disadvantages of these are unclear.

Third, one may repeatedly "squeeze information" from S, by remixing it with

itself and then revealing a subset of S. This is convenient when one would

like a family of hash functions with different output lengths. A special

case of the idea, called the "sponge construction," has been well studied and

adopted by the authors of Keccak and others [SHA-3].

3 HASH FUNCTIONS FOR HASH TABLES

It isn't hard to find real-life examples where hash tables or the hash

functions for them take more than 5% of a program's CPU time.

Improvements to hash tables and their hash functions are therefore a

classic example of software performance tuning. Unfortunately, the

best choice may be platform-dependent, so to avoid writing your own

collection of #ifdefs, please consider selecting something like the

FarmHash family of hash functions, that supply decent

platform-dependent logic for you.

To tune a program, often one will replace an existing hash function with a

faster, lower-quality hash function, despite the increased chance of unlucky

or pathological performance problems. Clever algorithms can mitigate this

risk. For example, hash tables can start with one hash function and then

switch to another if things seem to be going poorly. Therefore, one should

rarely plan to spend much CPU time on a secure hash function (such as SHA-3)

or a near-universal hash function (such as VHASH) when seeking the best

possible performance from a hash table. Against that, those types of hash

functions can limit the risk of pathological performance problems when one is

designing around typical hash-based algorithms that stick with a single hash

function no matter how it behaves on the data at hand.

4 REFERENCES

[Murmur] Appleby, Austin. https://code.google.com/p/smhasher,

https://sites.google.com/site/murmurhash/

[SMHasher] Appleby, Austin. https://code.google.com/p/smhasher

[SHA-3] Bertoni, Guido, et al. http://keccak.noekeon.org/

[Jenkins] Jenkins, Bob. http://burtleburtle.net/bob/hash/doobs.html

[VHASH] Krovetz, Ted. Message authentication on 64-bit architectures. In

Selected Areas of Cryptography – SAC 2006. Springer-Verlag, 2006.

[CityHash] Pike, Geoff and Alakuijala, Jyrki. https://code.google.com/p/cityhash

[FarmHash] Pike, Geoff. https://code.google.com/p/farmhash

[Pike] Pike, Geoff. <http://www.stanford.edu/class/ee380/Abstracts/121017-slides.pdf>