Understanding Tensorflow 2 source code

D. Gueorguiev 4/27/20

Contents

[Preliminaries 2](#_Toc41124554)

[Tensorflow globals and macros 2](#_Toc41124555)

[TF\_PREDICT\_TRUE and TF\_PREDICT\_FALSE 2](#_Toc41124556)

[Tensorflow Logging Internals 2](#_Toc41124557)

[Tensorflow Internal structures, containers and interfaces 3](#_Toc41124558)

[Internal Data Structures and Synchronization Primitives 3](#_Toc41124559)

[Class gtl::FlatMap 3](#_Toc41124560)

[Class gtl::internal::FlatRep - the internal representation for FlatMap and FlatSet 11](#_Toc41124561)

[C++ External and Internal API 18](#_Toc41124562)

[TF\_CAPI\_EXPORT directive 19](#_Toc41124563)

[TF\_VERSION string 20](#_Toc41124564)

[TF\_Buffer struct and functionality for manipulating it 20](#_Toc41124565)

[Global functions for manipulation of TF\_SessionOptions 21](#_Toc41124566)

[New graph construction API (under construction) 23](#_Toc41124567)

[Classes Graph and GraphDef 26](#_Toc41124568)

[Protobuf interfaces and formats 35](#_Toc41124569)

[Core/Protobuf/Config.proto 35](#_Toc41124570)

[Exploring the C++ code examples 35](#_Toc41124571)

[Exploring //tensorflow/core/example 35](#_Toc41124572)

[Proto file for ***Feature*** 35](#_Toc41124573)

[Proto file for ***Example***: 37](#_Toc41124574)

[Building the tensorflow libraries and examples using Bazel 39](#_Toc41124575)

[Building tensorflow libraries 39](#_Toc41124576)

[Building tensorflow core components 39](#_Toc41124577)

[Building Example Code 39](#_Toc41124578)

[Build Folder structure 40](#_Toc41124579)

[Build first C++ Tensorflow app 41](#_Toc41124580)

[Third Party and External Packages 41](#_Toc41124581)

[**BoringSSL:** https://boringssl.googlesource.com/boringssl/ 41](#_Toc41124582)

[**FarmHash:** https://github.com/google/farmhash 42](#_Toc41124583)

[*Introducing FarmHash* 42](#_Toc41124584)

[**HighwayHash**: https://github.com/google/highwayhash 42](#_Toc41124585)

[Appendix A: Bazel tutorial for Tensorflow Builds 43](#_Toc41124586)

[Appendix B: Hashing in Tensorflow 43](#_Toc41124587)

[Bibliography 45](#_Toc41124588)

# Preliminaries

The discussion in this document is based on Tensorflow master from **April 26, 2020** which I forked in my github repo <https://github.com/dimitarpg13/tensorflow/>. For reading this document a good understanding of C++ v14 , standard template library, utilities and some compiler internals is assumed.

# Tensorflow globals and macros

The header file for the core Tensorflow macros and globals is [tensorflow/core/platform/macros.h](https://github.com/dimitarpg13/tensorflow/blob/master/tensorflow/core/platform/macros.h).

## TF\_PREDICT\_TRUE and TF\_PREDICT\_FALSE

If we are not compiling with Nvidia GPU support which defines \_\_NVCC\_\_ global then TF\_PREDICT\_FALSE|TRUE uses the compiler attribute \_\_builtin\_expect(x,y) to issue a hint to the gcc compiler which branch to optimize against.

// Compilers can be told that a certain branch is not likely to be taken

// (for instance, a CHECK failure), and use that information in static

// analysis. Giving it this information can help it optimize for the

// common case in the absence of better information (ie.

// -fprofile-arcs).

//

// We need to disable this for GPU builds, though, since nvcc8 and older

// don't recognize `\_\_builtin\_expect` as a builtin, and fail compilation.

#if (!defined(\_\_NVCC\_\_)) && \

(TF\_HAS\_BUILTIN(\_\_builtin\_expect) || (defined(\_\_GNUC\_\_) && \_\_GNUC\_\_ >= 3))

#define TF\_PREDICT\_FALSE(x) (\_\_builtin\_expect(x, 0))

#define TF\_PREDICT\_TRUE(x) (\_\_builtin\_expect(!!(x), 1))

#else

#define TF\_PREDICT\_FALSE(x) (x)

#define TF\_PREDICT\_TRUE(x) (x)

#endif

# Tensorflow Logging Internals

The main Tensorflow logging header is [tensorflow/core/platform/default/logging.h](https://github.com/dimitarpg13/tensorflow/blob/master/tensorflow/core/platform/default/logging.h). This header file contains various macros and helper classes for implementing logging internals.

Few logging global constants:

namespace tensorflow {

const int INFO = 0; // base\_logging::INFO;

const int WARNING = 1; // base\_logging::WARNING;

const int ERROR = 2; // base\_logging::ERROR;

const int FATAL = 3; // base\_logging::FATAL;

const int NUM\_SEVERITIES = 4; // base\_logging::NUM\_SEVERITIES;

}

The base class which controls the logging behavior is LogMessage

# Tensorflow Internal structures, containers and interfaces

## Internal Data Structures and Synchronization Primitives

### Class gtl::FlatMap

gtl::FlatMap is a templetized container class used throughout the tensorflow core functionality. The FlatMap is not implemented like balanced binary search tree but instead it is implemented as a hash table with O(1) insertion, deletion, and search times.

Class gtl::FlatMap is declared in [tensorflow/core/lib/gtl/flatmap.h](https://github.com/dimitarpg13/tensorflow/blob/master/tensorflow/core/lib/gtl/flatmap.h) as shown below.

The class gtl::FlatMap declares and defines the following internals in its class declaration:

struct Bucket, struct ValueType, class iterator, class const\_iterator,

// FlatMap<K,V,...> provides a map from K to V.

//

// The map is implemented using an open-addressed hash table. A

// single array holds entire map contents and collisions are resolved

// by probing at a sequence of locations in the array.

template <typename Key, typename Val, class Hash = hash<Key>,

class Eq = std::equal\_to<Key>>

class FlatMap {

private:

// Forward declare some internal types needed in public section.

struct Bucket;

// We cannot use std::pair<> since internal representation stores

// keys and values in separate arrays, so we make a custom struct

// that holds references to the internal key, value elements.

//

// We define the struct as private ValueType, and typedef it as public

// value\_type, to work around a gcc bug when compiling the iterators.

struct ValueType {

typedef Key first\_type;

typedef Val second\_type;

const Key& first;

Val& second;

ValueType(const Key& k, Val& v) : first(k), second(v) {}

};

public:

typedef Key key\_type;

typedef Val mapped\_type;

typedef Hash hasher;

typedef Eq key\_equal;

typedef size\_t size\_type;

typedef ptrdiff\_t difference\_type;

typedef ValueType value\_type;

typedef value\_type\* pointer;

typedef const value\_type\* const\_pointer;

typedef value\_type& reference;

typedef const value\_type& const\_reference;

FlatMap() : FlatMap(1) {}

explicit FlatMap(size\_t N, const Hash& hf = Hash(), const Eq& eq = Eq())

: rep\_(N, hf, eq) {}

FlatMap(const FlatMap& src) : rep\_(src.rep\_) {}

// Move constructor leaves src in a valid but unspecified state (same as

// std::unordered\_map).

FlatMap(FlatMap&& src) : rep\_(std::move(src.rep\_)) {}

template <typename InputIter>

FlatMap(InputIter first, InputIter last, size\_t N = 1,

const Hash& hf = Hash(), const Eq& eq = Eq())

: FlatMap(N, hf, eq) {

insert(first, last);

}

FlatMap(std::initializer\_list<std::pair<const Key, Val>> init, size\_t N = 1,

const Hash& hf = Hash(), const Eq& eq = Eq())

: FlatMap(init.begin(), init.end(), N, hf, eq) {}

FlatMap& operator=(const FlatMap& src) {

rep\_.CopyFrom(src.rep\_);

return \*this;

}

// Move-assignment operator leaves src in a valid but unspecified state (same

// as std::unordered\_map).

FlatMap& operator=(FlatMap&& src) {

rep\_.MoveFrom(std::move(src.rep\_));

return \*this;

}

~FlatMap() {}

void swap(FlatMap& x) { rep\_.swap(x.rep\_); }

void clear\_no\_resize() { rep\_.clear\_no\_resize(); }

void clear() { rep\_.clear(); }

void reserve(size\_t N) { rep\_.Resize(std::max(N, size())); }

void rehash(size\_t N) { rep\_.Resize(std::max(N, size())); }

void resize(size\_t N) { rep\_.Resize(std::max(N, size())); }

size\_t size() const { return rep\_.size(); }

bool empty() const { return size() == 0; }

size\_t bucket\_count() const { return rep\_.bucket\_count(); }

hasher hash\_function() const { return rep\_.hash\_function(); }

key\_equal key\_eq() const { return rep\_.key\_eq(); }

class iterator {

public:

typedef typename FlatMap::difference\_type difference\_type;

typedef typename FlatMap::value\_type value\_type;

typedef typename FlatMap::pointer pointer;

typedef typename FlatMap::reference reference;

typedef ::std::forward\_iterator\_tag iterator\_category;

iterator() : b\_(nullptr), end\_(nullptr), i\_(0) {}

// Make iterator pointing at first element at or after b.

iterator(Bucket\* b, Bucket\* end) : b\_(b), end\_(end), i\_(0) { SkipUnused(); }

// Make iterator pointing exactly at ith element in b, which must exist.

iterator(Bucket\* b, Bucket\* end, uint32 i) : b\_(b), end\_(end), i\_(i) {

FillValue();

}

reference operator\*() { return \*val(); }

pointer operator->() { return val(); }

bool operator==(const iterator& x) const {

return b\_ == x.b\_ && i\_ == x.i\_;

}

bool operator!=(const iterator& x) const { return !(\*this == x); }

iterator& operator++() {

DCHECK(b\_ != end\_);

i\_++;

SkipUnused();

return \*this;

}

iterator operator++(int /\*indicates postfix\*/) {

iterator tmp(\*this);

++\*this;

return tmp;

}

private:

friend class FlatMap;

Bucket\* b\_;

Bucket\* end\_;

char space\_ alignas(value\_type)[sizeof(value\_type)];

uint32 i\_;

pointer val() { return reinterpret\_cast<pointer>(space\_); }

void FillValue() { new (space\_) value\_type(b\_->key(i\_), b\_->val(i\_)); }

void SkipUnused() {

while (b\_ < end\_) {

if (i\_ >= Rep::kWidth) {

i\_ = 0;

b\_++;

} else if (b\_->marker[i\_] < 2) {

i\_++;

} else {

FillValue();

break;

}

}

}

};

class const\_iterator {

private:

mutable iterator rep\_; // Share state and logic with non-const iterator.

public:

typedef typename FlatMap::difference\_type difference\_type;

typedef typename FlatMap::value\_type value\_type;

typedef typename FlatMap::const\_pointer pointer;

typedef typename FlatMap::const\_reference reference;

typedef ::std::forward\_iterator\_tag iterator\_category;

const\_iterator() : rep\_() {}

const\_iterator(Bucket\* start, Bucket\* end) : rep\_(start, end) {}

const\_iterator(Bucket\* b, Bucket\* end, uint32 i) : rep\_(b, end, i) {}

reference operator\*() const { return \*rep\_.val(); }

pointer operator->() const { return rep\_.val(); }

bool operator==(const const\_iterator& x) const { return rep\_ == x.rep\_; }

bool operator!=(const const\_iterator& x) const { return rep\_ != x.rep\_; }

const\_iterator& operator++() {

++rep\_;

return \*this;

}

const\_iterator operator++(int /\*indicates postfix\*/) {

const\_iterator tmp(\*this);

++\*this;

return tmp;

}

};

iterator begin() { return iterator(rep\_.start(), rep\_.limit()); }

iterator end() { return iterator(rep\_.limit(), rep\_.limit()); }

const\_iterator begin() const {

return const\_iterator(rep\_.start(), rep\_.limit());

}

const\_iterator end() const {

return const\_iterator(rep\_.limit(), rep\_.limit());

}

size\_t count(const Key& k) const { return rep\_.Find(k).found ? 1 : 0; }

iterator find(const Key& k) {

auto r = rep\_.Find(k);

return r.found ? iterator(r.b, rep\_.limit(), r.index) : end();

}

const\_iterator find(const Key& k) const {

auto r = rep\_.Find(k);

return r.found ? const\_iterator(r.b, rep\_.limit(), r.index) : end();

}

Val& at(const Key& k) {

auto r = rep\_.Find(k);

DCHECK(r.found);

return r.b->val(r.index);

}

const Val& at(const Key& k) const {

auto r = rep\_.Find(k);

DCHECK(r.found);

return r.b->val(r.index);

}

template <typename P>

std::pair<iterator, bool> insert(const P& p) {

return Insert(p.first, p.second);

}

std::pair<iterator, bool> insert(const std::pair<const Key, Val>& p) {

return Insert(p.first, p.second);

}

template <typename InputIter>

void insert(InputIter first, InputIter last) {

for (; first != last; ++first) {

insert(\*first);

}

}

Val& operator[](const Key& k) { return IndexOp(k); }

Val& operator[](Key&& k) { return IndexOp(std::forward<Key>(k)); }

template <typename... Args>

std::pair<iterator, bool> emplace(Args&&... args) {

return InsertPair(std::make\_pair(std::forward<Args>(args)...));

}

size\_t erase(const Key& k) {

auto r = rep\_.Find(k);

if (!r.found) return 0;

rep\_.Erase(r.b, r.index);

return 1;

}

iterator erase(iterator pos) {

rep\_.Erase(pos.b\_, pos.i\_);

++pos;

return pos;

}

iterator erase(iterator pos, iterator last) {

for (; pos != last; ++pos) {

rep\_.Erase(pos.b\_, pos.i\_);

}

return pos;

}

std::pair<iterator, iterator> equal\_range(const Key& k) {

auto pos = find(k);

if (pos == end()) {

return std::make\_pair(pos, pos);

} else {

auto next = pos;

++next;

return std::make\_pair(pos, next);

}

}

std::pair<const\_iterator, const\_iterator> equal\_range(const Key& k) const {

auto pos = find(k);

if (pos == end()) {

return std::make\_pair(pos, pos);

} else {

auto next = pos;

++next;

return std::make\_pair(pos, next);

}

}

bool operator==(const FlatMap& x) const {

if (size() != x.size()) return false;

for (auto& p : x) {

auto i = find(p.first);

if (i == end()) return false;

if (i->second != p.second) return false;

}

return true;

}

bool operator!=(const FlatMap& x) const { return !(\*this == x); }

// If key exists in the table, prefetch the associated value. This

// is a hint, and may have no effect.

void prefetch\_value(const Key& key) const { rep\_.Prefetch(key); }

private:

using Rep = internal::FlatRep<Key, Bucket, Hash, Eq>;

// Bucket stores kWidth <marker, key, value> triples.

// The data is organized as three parallel arrays to reduce padding.

struct Bucket {

uint8 marker[Rep::kWidth];

// Wrap keys and values in union to control construction and destruction.

union Storage {

struct {

Key key[Rep::kWidth];

Val val[Rep::kWidth];

};

Storage() {}

~Storage() {}

} storage;

Key& key(uint32 i) {

DCHECK\_GE(marker[i], 2);

return storage.key[i];

}

Val& val(uint32 i) {

DCHECK\_GE(marker[i], 2);

return storage.val[i];

}

template <typename V>

void InitVal(uint32 i, V&& v) {

new (&storage.val[i]) Val(std::forward<V>(v));

}

void Destroy(uint32 i) {

storage.key[i].Key::~Key();

storage.val[i].Val::~Val();

}

void MoveFrom(uint32 i, Bucket\* src, uint32 src\_index) {

new (&storage.key[i]) Key(std::move(src->storage.key[src\_index]));

new (&storage.val[i]) Val(std::move(src->storage.val[src\_index]));

}

void CopyFrom(uint32 i, Bucket\* src, uint32 src\_index) {

new (&storage.key[i]) Key(src->storage.key[src\_index]);

new (&storage.val[i]) Val(src->storage.val[src\_index]);

}

};

template <typename Pair>

std::pair<iterator, bool> InsertPair(Pair&& p) {

return Insert(std::forward<decltype(p.first)>(p.first),

std::forward<decltype(p.second)>(p.second));

}

template <typename K, typename V>

std::pair<iterator, bool> Insert(K&& k, V&& v) {

rep\_.MaybeResize();

auto r = rep\_.FindOrInsert(std::forward<K>(k));

const bool inserted = !r.found;

if (inserted) {

r.b->InitVal(r.index, std::forward<V>(v));

}

return {iterator(r.b, rep\_.limit(), r.index), inserted};

}

template <typename K>

Val& IndexOp(K&& k) {

rep\_.MaybeResize();

auto r = rep\_.FindOrInsert(std::forward<K>(k));

Val\* vptr = &r.b->val(r.index);

if (!r.found) {

new (vptr) Val(); // Initialize value in new slot.

}

return \*vptr;

}

Rep rep\_;

};

### Class gtl::internal::FlatRep - the internal representation for FlatMap and FlatSet

The internal representation used in FlatMap and FlatSet by the using declaration

using Rep = internal::FlatRep<Key, Bucket, Hash, Eq>;

is declared in [tensorflow/core/lib/gtl/flatrep.h](https://github.com/dimitarpg13/tensorflow/blob/master/tensorflow/core/lib/gtl/flatrep.h).

// Internal representation for FlatMap and FlatSet.

//

// The representation is an open-addressed hash table. Conceptually,

// the representation is a flat array of entries. However we

// structure it as an array of buckets where each bucket holds

// kWidth entries along with metadata for the kWidth entries. The

// metadata marker is

//

// (a) kEmpty: the entry is empty

// (b) kDeleted: the entry has been deleted

// (c) other: the entry is occupied and has low-8 bits of its hash.

// These hash bits can be used to avoid potentially expensive

// key comparisons.

//

// FlatMap passes in a bucket that contains keys and values, FlatSet

// passes in a bucket that does not contain values.

template <typename Key, typename Bucket, class Hash, class Eq>

class FlatRep {

public:

// kWidth is the number of entries stored in a bucket.

static constexpr uint32 kBase = 3;

static constexpr uint32 kWidth = (1 << kBase);

FlatRep(size\_t N, const Hash& hf, const Eq& eq) : hash\_(hf), equal\_(eq) {

Init(N);

}

FlatRep(const FlatRep& src) : hash\_(src.hash\_), equal\_(src.equal\_) {

Init(src.size());

CopyEntries(src.array\_, src.end\_, CopyEntry());

}

FlatRep(FlatRep&& src)

// Copy rather than move src.hash\_ and src.equal\_. This is necessary to

// leave src in a valid state -- otherwise e.g. if hash\_ is an

// std::function, moving it would null it out.

: hash\_(src.hash\_), equal\_(src.equal\_) {

// TODO(jlebar): Init(1) still allocates some memory, so this isn't as cheap

// as it could be. The fundamental problem is that we need to leave src in

// a valid state, and FlatRep \*always\* owns a nonzero amount of memory.

Init(1);

swap(src);

}

~FlatRep() {

clear\_no\_resize();

delete[] array\_;

}

// Simple accessors.

size\_t size() const { return not\_empty\_ - deleted\_; }

size\_t bucket\_count() const { return mask\_ + 1; }

Bucket\* start() const { return array\_; }

Bucket\* limit() const { return end\_; }

const Hash& hash\_function() const { return hash\_; }

const Eq& key\_eq() const { return equal\_; }

// Overwrite contents of \*this with contents of src.

void CopyFrom(const FlatRep& src) {

if (this != &src) {

clear\_no\_resize();

delete[] array\_;

Init(src.size());

CopyEntries(src.array\_, src.end\_, CopyEntry());

}

}

void MoveFrom(FlatRep&& src) {

if (this != &src) {

swap(src);

}

}

void clear\_no\_resize() {

for (Bucket\* b = array\_; b != end\_; b++) {

for (uint32 i = 0; i < kWidth; i++) {

if (b->marker[i] >= 2) {

b->Destroy(i);

b->marker[i] = kEmpty;

}

}

}

not\_empty\_ = 0;

deleted\_ = 0;

}

void clear() {

clear\_no\_resize();

grow\_ = 0; // Consider shrinking in MaybeResize()

MaybeResize();

}

void swap(FlatRep& x) {

using std::swap;

swap(array\_, x.array\_);

swap(end\_, x.end\_);

swap(lglen\_, x.lglen\_);

swap(mask\_, x.mask\_);

swap(not\_empty\_, x.not\_empty\_);

swap(deleted\_, x.deleted\_);

swap(grow\_, x.grow\_);

swap(shrink\_, x.shrink\_);

}

struct SearchResult {

bool found;

Bucket\* b;

uint32 index;

};

// Hash value is partitioned as follows:

// 1. Bottom 8 bits are stored in bucket to help speed up comparisons.

// 2. Next 3 bits give index inside bucket.

// 3. Remaining bits give bucket number.

// Find bucket/index for key k.

SearchResult Find(const Key& k) const {

size\_t h = hash\_(k);

const uint32 marker = Marker(h & 0xff);

size\_t index = (h >> 8) & mask\_; // Holds bucket num and index-in-bucket

uint32 num\_probes = 1; // Needed for quadratic probing

while (true) {

uint32 bi = index & (kWidth - 1);

Bucket\* b = &array\_[index >> kBase];

const uint32 x = b->marker[bi];

if (x == marker && equal\_(b->key(bi), k)) {

return {true, b, bi};

} else if (x == kEmpty) {

return {false, nullptr, 0};

}

index = NextIndex(index, num\_probes);

num\_probes++;

}

}

// Find bucket/index for key k, creating a new one if necessary.

//

// KeyType is a template parameter so that k's type is deduced and it

// becomes a universal reference which allows the key initialization

// below to use an rvalue constructor if available.

template <typename KeyType>

SearchResult FindOrInsert(KeyType&& k) {

size\_t h = hash\_(k);

const uint32 marker = Marker(h & 0xff);

size\_t index = (h >> 8) & mask\_; // Holds bucket num and index-in-bucket

uint32 num\_probes = 1; // Needed for quadratic probing

Bucket\* del = nullptr; // First encountered deletion for kInsert

uint32 di = 0;

while (true) {

uint32 bi = index & (kWidth - 1);

Bucket\* b = &array\_[index >> kBase];

const uint32 x = b->marker[bi];

if (x == marker && equal\_(b->key(bi), k)) {

return {true, b, bi};

} else if (!del && x == kDeleted) {

// Remember deleted index to use for insertion.

del = b;

di = bi;

} else if (x == kEmpty) {

if (del) {

// Store in the first deleted slot we encountered

b = del;

bi = di;

deleted\_--; // not\_empty\_ does not change

} else {

not\_empty\_++;

}

b->marker[bi] = marker;

new (&b->key(bi)) Key(std::forward<KeyType>(k));

return {false, b, bi};

}

index = NextIndex(index, num\_probes);

num\_probes++;

}

}

void Erase(Bucket\* b, uint32 i) {

b->Destroy(i);

b->marker[i] = kDeleted;

deleted\_++;

grow\_ = 0; // Consider shrinking on next insert

}

void Prefetch(const Key& k) const {

size\_t h = hash\_(k);

size\_t index = (h >> 8) & mask\_; // Holds bucket num and index-in-bucket

uint32 bi = index & (kWidth - 1);

Bucket\* b = &array\_[index >> kBase];

port::prefetch<port::PREFETCH\_HINT\_T0>(&b->marker[bi]);

port::prefetch<port::PREFETCH\_HINT\_T0>(&b->storage.key[bi]);

}

inline void MaybeResize() {

if (not\_empty\_ < grow\_) {

return; // Nothing to do

}

if (grow\_ == 0) {

// Special value set by erase to cause shrink on next insert.

if (size() >= shrink\_) {

// Not small enough to shrink.

grow\_ = static\_cast<size\_t>(bucket\_count() \* 0.8);

if (not\_empty\_ < grow\_) return;

}

}

Resize(size() + 1);

}

void Resize(size\_t N) {

Bucket\* old = array\_;

Bucket\* old\_end = end\_;

Init(N);

CopyEntries(old, old\_end, MoveEntry());

delete[] old;

}

private:

enum { kEmpty = 0, kDeleted = 1 }; // Special markers for an entry.

Hash hash\_; // User-supplied hasher

Eq equal\_; // User-supplied comparator

uint8 lglen\_; // lg(#buckets)

Bucket\* array\_; // array of length (1 << lglen\_)

Bucket\* end\_; // Points just past last bucket in array\_

size\_t mask\_; // (# of entries in table) - 1

size\_t not\_empty\_; // Count of entries with marker != kEmpty

size\_t deleted\_; // Count of entries with marker == kDeleted

size\_t grow\_; // Grow array when not\_empty\_ >= grow\_

size\_t shrink\_; // Shrink array when size() < shrink\_

// Avoid kEmpty and kDeleted markers when computing hash values to

// store in Bucket::marker[].

static uint32 Marker(uint32 hb) { return hb + (hb < 2 ? 2 : 0); }

void Init(size\_t N) {

// Make enough room for N elements.

size\_t lg = 0; // Smallest table is just one bucket.

while (N >= 0.8 \* ((1 << lg) \* kWidth)) {

lg++;

}

const size\_t n = (1 << lg);

Bucket\* array = new Bucket[n];

for (size\_t i = 0; i < n; i++) {

Bucket\* b = &array[i];

memset(b->marker, kEmpty, kWidth);

}

const size\_t capacity = (1 << lg) \* kWidth;

lglen\_ = lg;

mask\_ = capacity - 1;

array\_ = array;

end\_ = array + n;

not\_empty\_ = 0;

deleted\_ = 0;

grow\_ = static\_cast<size\_t>(capacity \* 0.8);

if (lg == 0) {

// Already down to one bucket; no more shrinking.

shrink\_ = 0;

} else {

shrink\_ = static\_cast<size\_t>(grow\_ \* 0.4); // Must be less than 0.5

}

}

// Used by FreshInsert when we should copy from source.

struct CopyEntry {

inline void operator()(Bucket\* dst, uint32 dsti, Bucket\* src, uint32 srci) {

dst->CopyFrom(dsti, src, srci);

}

};

// Used by FreshInsert when we should move from source.

struct MoveEntry {

inline void operator()(Bucket\* dst, uint32 dsti, Bucket\* src, uint32 srci) {

dst->MoveFrom(dsti, src, srci);

src->Destroy(srci);

src->marker[srci] = kDeleted;

}

};

template <typename Copier>

void CopyEntries(Bucket\* start, Bucket\* end, Copier copier) {

for (Bucket\* b = start; b != end; b++) {

for (uint32 i = 0; i < kWidth; i++) {

if (b->marker[i] >= 2) {

FreshInsert(b, i, copier);

}

}

}

}

// Create an entry for the key numbered src\_index in \*src and return

// its bucket/index. Used for insertion into a fresh table. We

// assume that there are no deletions, and k does not already exist

// in the table.

template <typename Copier>

void FreshInsert(Bucket\* src, uint32 src\_index, Copier copier) {

size\_t h = hash\_(src->key(src\_index));

const uint32 marker = Marker(h & 0xff);

size\_t index = (h >> 8) & mask\_; // Holds bucket num and index-in-bucket

uint32 num\_probes = 1; // Needed for quadratic probing

while (true) {

uint32 bi = index & (kWidth - 1);

Bucket\* b = &array\_[index >> kBase];

const uint32 x = b->marker[bi];

if (x == 0) {

b->marker[bi] = marker;

not\_empty\_++;

copier(b, bi, src, src\_index);

return;

}

index = NextIndex(index, num\_probes);

num\_probes++;

}

}

inline size\_t NextIndex(size\_t i, uint32 num\_probes) const {

// Quadratic probing.

return (i + num\_probes) & mask\_;

}

};

## C++ External and Internal API

The main header file which declares the C++ API interface is:

[bazel-tensorflow/tensorflow/c/c\_api.h](https://github.com/dimitarpg13/tensorflow/blob/master/tensorflow/c/c_api.h)

// --------------------------------------------------------------------------

// C API for TensorFlow.

//

// The API leans towards simplicity and uniformity instead of convenience

// since most usage will be by language specific wrappers.

//

// Conventions:

// \* We use the prefix TF\_ for everything in the API.

// \* Objects are always passed around as pointers to opaque structs

// and these structs are allocated/deallocated via the API.

// \* TF\_Status holds error information. It is an object type

// and therefore is passed around as a pointer to an opaque

// struct as mentioned above.

// \* Every call that has a TF\_Status\* argument clears it on success

// and fills it with error info on failure.

// \* unsigned char is used for booleans (instead of the 'bool' type).

// In C++ bool is a keyword while in C99 bool is a macro defined

// in stdbool.h. It is possible for the two to be inconsistent.

// For example, neither the C99 nor the C++11 standard force a byte

// size on the bool type, so the macro defined in stdbool.h could

// be inconsistent with the bool keyword in C++. Thus, the use

// of stdbool.h is avoided and unsigned char is used instead.

// \* size\_t is used to represent byte sizes of objects that are

// materialized in the address space of the calling process.

// \* int is used as an index into arrays.

// \* Deletion functions are safe to call on nullptr.

//

// Questions left to address:

// \* Might at some point need a way for callers to provide their own Env.

// \* Maybe add TF\_TensorShape that encapsulates dimension info.

//

// Design decisions made:

// \* Backing store for tensor memory has an associated deallocation

// function. This deallocation function will point to client code

// for tensors populated by the client. So the client can do things

// like shadowing a numpy array.

// \* We do not provide TF\_OK since it is not strictly necessary and we

// are not optimizing for convenience.

// \* We make assumption that one session has one graph. This should be

// fine since we have the ability to run sub-graphs.

// \* We could allow NULL for some arguments (e.g., NULL options arg).

// However since convenience is not a primary goal, we don't do this.

// \* Devices are not in this API. Instead, they are created/used internally

// and the API just provides high level controls over the number of

// devices of each type.

For Linux and MacOS the symbol TF\_CAPI\_EXPORT is defined as:

### TF\_CAPI\_EXPORT directive

#define TF\_CAPI\_EXPORT \_\_attribute\_\_((visibility("default")))

GCC-specific details on the visibility attribute can be found [here](https://gcc.gnu.org/wiki/Visibility). Basically, by judicious use of the visibility attribute we can decrease dramatically the load times of dynamically shared objects i.e. tensorflow library libtensorflow\_cc.so. Great [article](#UlrichDrepperSharedLibrary) about writing shared libraries by Ulrich Drepper can be found [here](https://akkadia.org/drepper/dsohowto.pdf).

### TF\_VERSION string

The first member of the API is the TF\_VERSION string :

// --------------------------------------------------------------------------

// TF\_Version returns a string describing version information of the

// TensorFlow library. TensorFlow using semantic versioning.

TF\_CAPI\_EXPORT extern const char\* TF\_Version(void);

TensorFlow follows Semantic Versioning 2.0 ([semver](https://semver.org/spec/v2.0.0.html)) for its public API. Each release version of TensorFlow has the form MAJOR.MINOR.PATCH. For example, TensorFlow version 1.2.3 has MAJOR version 1, MINOR version 2, and PATCH version 3. Changes to each number have the following meaning:

MAJOR: Potentially backwards incompatible changes. Code and data that worked with a previous major release will not necessarily work with the new release. However, in some cases existing TensorFlow graphs and checkpoints may be migratable to the newer release; see Compatibility of graphs and checkpoints for details on data compatibility.

MINOR: Backwards compatible features, speed improvements, etc. Code and data that worked with a previous minor release and which depends only on the non-experimental public API will continue to work unchanged. For details on what is and is not the public API, see What is covered.

PATCH: Backwards compatible bug fixes.

For example, release 1.0.0 introduced backwards incompatible changes from release 0.12.1. However, release 1.1.1 was backwards compatible with release 1.0.0.

### TF\_Buffer struct and functionality for manipulating it

The struct **TF\_Buffer** is defined next. Its purpose and usage are described in the comment lines:

// --------------------------------------------------------------------------

// TF\_Buffer holds a pointer to a block of data and its associated length.

// Typically, the data consists of a serialized protocol buffer, but other data

// may also be held in a buffer.

//

// By default, TF\_Buffer itself does not do any memory management of the

// pointed-to block. If need be, users of this struct should specify how to

// deallocate the block by setting the `data\_deallocator` function pointer.

typedef struct TF\_Buffer {

const void\* data;

size\_t length;

void (\*data\_deallocator)(void\* data, size\_t length);

} TF\_Buffer;

The next member is the global function **TF\_NewBufferFromString** which instantiates a new **TF\_Buffer** from read-only protobuf instances

// Makes a copy of the input and sets an appropriate deallocator. Useful for

// passing in read-only, input protobufs.

TF\_CAPI\_EXPORT extern TF\_Buffer\* TF\_NewBufferFromString(const void\* proto, size\_t proto\_len);

Here is an implementation for the function:

TF\_Buffer\* TF\_NewBufferFromString(const void\* proto, size\_t proto\_len) {

void\* copy = tensorflow::port::Malloc(proto\_len);

memcpy(copy, proto, proto\_len);

TF\_Buffer\* buf = new TF\_Buffer;

buf->data = copy;

buf->length = proto\_len;

buf->data\_deallocator = [](void\* data, size\_t length) {

tensorflow::port::Free(data);

};

return buf;

}

Follow three more global functions for manipulation of **TF\_Buffer**:

// Useful for passing \*out\* a protobuf.

TF\_CAPI\_EXPORT extern TF\_Buffer\* TF\_NewBuffer(void);

TF\_CAPI\_EXPORT extern void TF\_DeleteBuffer(TF\_Buffer\*);

TF\_CAPI\_EXPORT extern TF\_Buffer TF\_GetBuffer(TF\_Buffer\* buffer);

TF\_Buffer\* TF\_NewBuffer() { return new TF\_Buffer{nullptr, 0, nullptr}; }

void TF\_DeleteBuffer(TF\_Buffer\* buffer) {

if (buffer == nullptr) return;

if (buffer->data\_deallocator != nullptr) {

(\*buffer->data\_deallocator)(const\_cast<void\*>(buffer->data),

buffer->length);

}

delete buffer;

}

TF\_Buffer TF\_GetBuffer(TF\_Buffer\* buffer) { return \*buffer; }

### Global functions for manipulation of TF\_SessionOptions

// --------------------------------------------------------------------------

// TF\_SessionOptions holds options that can be passed during session creation.

typedef struct TF\_SessionOptions TF\_SessionOptions;

// Return a new options object.

TF\_CAPI\_EXPORT extern TF\_SessionOptions\* TF\_NewSessionOptions(void);

// Set the target in TF\_SessionOptions.options.

// target can be empty, a single entry, or a comma separated list of entries.

// Each entry is in one of the following formats :

// "local"

// ip:port

// host:port

TF\_CAPI\_EXPORT extern void TF\_SetTarget(TF\_SessionOptions\* options,

const char\* target);

// Set the config in TF\_SessionOptions.options.

// config should be a serialized tensorflow.ConfigProto proto.

// If config was not parsed successfully as a ConfigProto, record the

// error information in \*status.

TF\_CAPI\_EXPORT extern void TF\_SetConfig(TF\_SessionOptions\* options,

const void\* proto, size\_t proto\_len,

TF\_Status\* status);

// Destroy an options object.

TF\_CAPI\_EXPORT extern void TF\_DeleteSessionOptions(TF\_SessionOptions\*);

TF\_SessionOptions\* TF\_NewSessionOptions() { return new TF\_SessionOptions; }

void TF\_DeleteSessionOptions(TF\_SessionOptions\* opt) { delete opt; }

void TF\_SetTarget(TF\_SessionOptions\* options, const char\* target) {

options->options.target = target;

}

void TF\_SetConfig(TF\_SessionOptions\* options, const void\* proto,

size\_t proto\_len, TF\_Status\* status) {

if (!options->options.config.ParseFromArray(proto, proto\_len)) {

status->status = InvalidArgument("Unparseable ConfigProto");

}

}

/// Configuration information for a Session.

struct TF\_SessionOptions {

tensorflow::SessionOptions options;

};

[tensorflow/core/public/session\_options.h](https://github.com/dimitarpg13/tensorflow/blob/master/tensorflow/core/public/session_options.h#L28-L61)

struct SessionOptions {

/// The environment to use.

Env\* env;

/// \brief The TensorFlow runtime to connect to.

///

/// If 'target' is empty or unspecified, the local TensorFlow runtime

/// implementation will be used. Otherwise, the TensorFlow engine

/// defined by 'target' will be used to perform all computations.

///

/// "target" can be either a single entry or a comma separated list

/// of entries. Each entry is a resolvable address of the

/// following format:

/// local

/// ip:port

/// host:port

/// ... other system-specific formats to identify tasks and jobs ...

///

/// NOTE: at the moment 'local' maps to an in-process service-based

/// runtime.

///

/// Upon creation, a single session affines itself to one of the

/// remote processes, with possible load balancing choices when the

/// "target" resolves to a list of possible processes.

///

/// If the session disconnects from the remote process during its

/// lifetime, session calls may fail immediately.

std::string target;

/// Configuration options.

ConfigProto config;

SessionOptions();

};

### New graph construction API (under construction)

// Represents a computation graph. Graphs may be shared between sessions.

// Graphs are thread-safe when used as directed below.

typedef struct TF\_Graph TF\_Graph;

// Return a new graph object.

TF\_CAPI\_EXPORT extern TF\_Graph\* TF\_NewGraph(void);

// Destroy an options object. Graph will be deleted once no more

// TFSession's are referencing it.

TF\_CAPI\_EXPORT extern void TF\_DeleteGraph(TF\_Graph\*);

// Operation being built. The underlying graph must outlive this.

typedef struct TF\_OperationDescription TF\_OperationDescription;

// Operation that has been added to the graph. Valid until the graph is

// deleted -- in particular adding a new operation to the graph does not

// invalidate old TF\_Operation\* pointers.

typedef struct TF\_Operation TF\_Operation;

// Represents a specific input of an operation.

typedef struct TF\_Input {

TF\_Operation\* oper;

int index; // The index of the input within oper.

} TF\_Input;

// Represents a specific output of an operation.

typedef struct TF\_Output {

TF\_Operation\* oper;

int index; // The index of the output within oper.

} TF\_Output;

// TF\_Function is a grouping of operations with defined inputs and outputs.

// Once created and added to graphs, functions can be invoked by creating an

// operation whose operation type matches the function name.

typedef struct TF\_Function TF\_Function;

// Function definition options. TODO(iga): Define and implement

typedef struct TF\_FunctionOptions TF\_FunctionOptions;

// Sets the shape of the Tensor referenced by `output` in `graph` to

// the shape described by `dims` and `num\_dims`.

//

// If the number of dimensions is unknown, `num\_dims` must be set to

// -1 and `dims` can be null. If a dimension is unknown, the

// corresponding entry in the `dims` array must be -1.

//

// This does not overwrite the existing shape associated with `output`,

// but merges the input shape with the existing shape. For example,

// setting a shape of [-1, 2] with an existing shape [2, -1] would set

// a final shape of [2, 2] based on shape merging semantics.

//

// Returns an error into `status` if:

// \* `output` is not in `graph`.

// \* An invalid shape is being set (e.g., the shape being set

// is incompatible with the existing shape).

TF\_CAPI\_EXPORT extern void TF\_GraphSetTensorShape(TF\_Graph\* graph,

TF\_Output output,

const int64\_t\* dims,

const int num\_dims,

TF\_Status\* status);

// Returns the number of dimensions of the Tensor referenced by `output`

// in `graph`.

//

// If the number of dimensions in the shape is unknown, returns -1.

//

// Returns an error into `status` if:

// \* `output` is not in `graph`.

TF\_CAPI\_EXPORT extern int TF\_GraphGetTensorNumDims(TF\_Graph\* graph,

TF\_Output output,

TF\_Status\* status);

// Returns the shape of the Tensor referenced by `output` in `graph`

// into `dims`. `dims` must be an array large enough to hold `num\_dims`

// entries (e.g., the return value of TF\_GraphGetTensorNumDims).

//

// If the number of dimensions in the shape is unknown or the shape is

// a scalar, `dims` will remain untouched. Otherwise, each element of

// `dims` will be set corresponding to the size of the dimension. An

// unknown dimension is represented by `-1`.

//

// Returns an error into `status` if:

// \* `output` is not in `graph`.

// \* `num\_dims` does not match the actual number of dimensions.

TF\_CAPI\_EXPORT extern void TF\_GraphGetTensorShape(TF\_Graph\* graph,

TF\_Output output,

int64\_t\* dims, int num\_dims,

TF\_Status\* status);

// Operation will only be added to \*graph when TF\_FinishOperation() is

// called (assuming TF\_FinishOperation() does not return an error).

// \*graph must not be deleted until after TF\_FinishOperation() is

// called.

TF\_CAPI\_EXPORT extern TF\_OperationDescription\* TF\_NewOperation(

TF\_Graph\* graph, const char\* op\_type, const char\* oper\_name);

// Specify the device for `desc`. Defaults to empty, meaning unconstrained.

TF\_CAPI\_EXPORT extern void TF\_SetDevice(TF\_OperationDescription\* desc,

const char\* device);

bazel-tensorflow/tensorflow/c/c\_api.cc

bazel-tensorflow/tensorflow/c/c\_api\_internal.h

bazel-tensorflow/tensorflow/c/c\_api\_function.cc

bazel-tensorflow/tensorflow/c/eager/c\_api\_unified\_experimental\_graph.cc // defines struct

// GraphTensor, struct GraphFunction

bazel-tensorflow/tensorflow/c/c\_api\_experimental.cc

bazel-tensorflow/tensorflow/c/c\_api\_test.cc

bazel-tensorflow/tensorflow/c/c\_api\_function\_test.cc

bazel-tensorflow/tensorflow/c/while\_loop\_test.cc

bazel-tensorflow/tensorflow/c/c\_test\_util.cc

bazel-tensorflow/tensorflow/c/eager/c\_api\_experimental\_test.cc

## Classes Graph and GraphDef

The classes Graph (or Computation Graph) is a core concept of tensorflow to present computation. When first using TF, we first will create Computation Graph and pass the Graph to TF.

The Computation Graph is given by class TF\_Graph defined in

struct TF\_Graph {

TF\_Graph();

tensorflow::mutex mu;

tensorflow::Graph graph TF\_GUARDED\_BY(mu);

// Runs shape inference.

tensorflow::ShapeRefiner refiner TF\_GUARDED\_BY(mu);

// Maps from name of an operation to the Node\* in 'graph'.

std::unordered\_map<tensorflow::string, tensorflow::Node\*> name\_map

TF\_GUARDED\_BY(mu);

// The keys of this map are all the active sessions using this graph. Each

// value records whether the graph has been mutated since the corresponding

// session has been run (this is detected in RecordMutation function). If the

// string is empty, no mutation has occurred. Otherwise the string is a

// description of the mutation suitable for returning to the user.

//

// Sessions are added to this map in TF\_NewSession, and removed in

// TF\_DeleteSession.

// TF\_Graph may only / must be deleted when

// sessions.size() == 0 && delete\_requested == true

//

// TODO(b/74949947): mutations currently trigger a warning instead of a bad

// status, this should be reverted when possible.

tensorflow::gtl::FlatMap<TF\_Session\*, tensorflow::string> sessions

TF\_GUARDED\_BY(mu);

bool delete\_requested TF\_GUARDED\_BY(mu); // set true by TF\_DeleteGraph

// Used to link graphs contained in TF\_WhileParams to the parent graph that

// will eventually contain the full while loop.

TF\_Graph\* parent;

TF\_Output\* parent\_inputs;

};

The internal container class tensorflow::Graph is defined in [tensorflow/core/graph/graph.h](https://github.com/dimitarpg13/tensorflow/blob/master/tensorflow/core/graph/graph.h#L473-L761) as:

// Thread compatible but not thread safe.

class Graph {

public:

// Constructs a graph with a single SOURCE (always id kSourceId) and a

// single SINK (always id kSinkId) node, and an edge from SOURCE->SINK.

//

// The graph can hold ops found in the registry. `ops`s lifetime must be at

// least that of the constructed graph's.

explicit Graph(const OpRegistryInterface\* ops);

// Constructs a graph with a single SOURCE (always id kSourceId) and a

// single SINK (always id kSinkId) node, and an edge from SOURCE->SINK.

//

// The graph can hold ops found in `flib\_def`. Unlike the constructor taking

// an OpRegistryInterface, this constructor copies the function definitions in

// `flib\_def` so its lifetime may be shorter than that of the graph's. The

// OpRegistryInterface backing `flib\_def` must still have the lifetime of the

// graph though.

explicit Graph(const FunctionLibraryDefinition& flib\_def);

~Graph();

static const int kControlSlot;

// The GraphDef version range of this graph (see graph.proto).

const VersionDef& versions() const;

void set\_versions(const VersionDef& versions);

// Adds a new node to this graph, and returns it. Infers the Op and

// input/output types for the node. \*this owns the returned instance.

// Returns nullptr and sets \*status on error.

Node\* AddNode(NodeDef node\_def, Status\* status);

// Copies \*node, which may belong to another graph, to a new node,

// which is returned. Does not copy any edges. \*this owns the

// returned instance.

Node\* CopyNode(const Node\* node);

// Removes a node from this graph, including all edges from or to it.

// \*node should not be accessed after calling this function.

// REQUIRES: node->IsOp()

void RemoveNode(Node\* node);

// Adds an edge that connects the xth output of `source` to the yth input of

// `dest` and returns it. Does not update dest's NodeDef.

const Edge\* AddEdge(Node\* source, int x, Node\* dest, int y);

// Adds a control edge (no data flows along this edge) that connects `source`

// to `dest`. If `dest`s NodeDef is missing the corresponding control input,

// adds the control input.

//

// If such a control edge already exists and `allow\_duplicates` is false, no

// edge is added and the function returns nullptr. Otherwise the edge is

// unconditionally created and returned. The NodeDef is not updated if

// `allow\_duplicates` is true.

// TODO(skyewm): // TODO(skyewm): allow\_duplicates is needed only by

// graph\_partition.cc. Figure out if we can do away with it.

const Edge\* AddControlEdge(Node\* source, Node\* dest,

bool allow\_duplicates = false);

// Removes edge from the graph. Does not update the destination node's

// NodeDef.

// REQUIRES: The edge must exist.

void RemoveEdge(const Edge\* edge);

// Removes control edge `edge` from the graph. Note that this also updates

// the corresponding NodeDef to reflect the change.

// REQUIRES: The control edge must exist.

void RemoveControlEdge(const Edge\* e);

// Updates the input to a node. The existing edge to `dst` is removed and an

// edge from `new\_src` to `dst` is created. The NodeDef associated with `dst`

// is also updated.

Status UpdateEdge(Node\* new\_src, int new\_src\_index, Node\* dst, int dst\_index);

// Like AddEdge but updates dst's NodeDef. Used to add an input edge to a

// "While" op during gradient construction, see AddInputWhileHack in

// python\_api.h for more details.

Status AddWhileInputHack(Node\* new\_src, int new\_src\_index, Node\* dst);

// Adds the function and gradient definitions in `fdef\_lib` to this graph's op

// registry. Ignores duplicate functions, and returns a bad status if an

// imported function differs from an existing function or op with the same

// name.

Status AddFunctionLibrary(const FunctionDefLibrary& fdef\_lib);

// The number of live nodes in the graph.

//

// Because nodes can be removed from the graph, num\_nodes() is often

// smaller than num\_node\_ids(). If one needs to create an array of

// nodes indexed by node ids, num\_node\_ids() should be used as the

// array's size.

int num\_nodes() const { return num\_nodes\_; }

// The number of live nodes in the graph, excluding the Source and Sink nodes.

int num\_op\_nodes() const {

DCHECK\_GE(num\_nodes\_, 2);

return num\_nodes\_ - 2;

}

// The number of live edges in the graph.

//

// Because edges can be removed from the graph, num\_edges() is often

// smaller than num\_edge\_ids(). If one needs to create an array of

// edges indexed by edge ids, num\_edge\_ids() should be used as the

// array's size.

int num\_edges() const { return num\_edges\_; }

// Serialize the nodes starting at `from\_node\_id` to a GraphDef.

void ToGraphDefSubRange(GraphDef\* graph\_def, int from\_node\_id) const;

// Serialize to a GraphDef.

void ToGraphDef(GraphDef\* graph\_def) const;

// This version can be called from debugger to inspect the graph content.

// Use the previous version outside debug context for efficiency reasons.

//

// Note: We do not expose a DebugString() API, since GraphDef.DebugString() is

// not defined in some TensorFlow builds.

GraphDef ToGraphDefDebug() const;

// Generate new node name with the specified prefix that is unique

// across this graph.

string NewName(StringPiece prefix);

// Access to the list of all nodes. Example usage:

// for (Node\* node : graph.nodes()) { ... }

gtl::iterator\_range<NodeIter> nodes() const;

// Access to the list of all nodes, excluding the Source and Sink nodes.

gtl::iterator\_range<NodeIter> op\_nodes() const;

// Returns one more than the maximum id assigned to any node.

int num\_node\_ids() const { return nodes\_.size(); }

// Returns the node associated with an id, or nullptr if no node

// with that id (the node with that id was removed and the id has

// not yet been re-used). \*this owns the returned instance.

// REQUIRES: 0 <= id < num\_node\_ids().

Node\* FindNodeId(int id) const { return nodes\_[id]; }

// Returns one more than the maximum id assigned to any edge.

int num\_edge\_ids() const { return edges\_.size(); }

// Returns the Edge associated with an id, or nullptr if no edge

// with that id (the node with that id was removed and the id has

// not yet been re-used). \*this owns the returned instance.

// REQUIRES: 0 <= id < num\_node\_ids().

const Edge\* FindEdgeId(int id) const { return edges\_[id]; }

// Access to the set of all edges. Example usage:

// for (const Edge\* e : graph.edges()) { ... }

GraphEdgesIterable edges() const { return GraphEdgesIterable(edges\_); }

// The pre-defined nodes.

enum { kSourceId = 0, kSinkId = 1 };

Node\* source\_node() const { return FindNodeId(kSourceId); }

Node\* sink\_node() const { return FindNodeId(kSinkId); }

const OpRegistryInterface\* op\_registry() const { return &ops\_; }

const FunctionLibraryDefinition& flib\_def() const { return ops\_; }

void CheckDeviceNameIndex(int index) {

DCHECK\_GE(index, 0);

DCHECK\_LT(index, static\_cast<int>(device\_names\_.size()));

}

int InternDeviceName(const string& device\_name);

const string& get\_assigned\_device\_name(const Node& node) const {

return device\_names\_[node.assigned\_device\_name\_index()];

}

void set\_assigned\_device\_name\_index(Node\* node, int device\_name\_index) {

CheckDeviceNameIndex(device\_name\_index);

node->assigned\_device\_name\_index\_ = device\_name\_index;

}

void set\_assigned\_device\_name(Node\* node, const string& device\_name) {

node->assigned\_device\_name\_index\_ = InternDeviceName(device\_name);

}

// Returns OK if `node` is non-null and belongs to this graph

Status IsValidNode(const Node\* node) const;

// Returns OK if IsValidNode(`node`) and `idx` is a valid output. Does not

// accept control outputs.

Status IsValidOutputTensor(const Node\* node, int idx) const;

// Returns OK if IsValidNode(`node`) and `idx` a valid input. Does not accept

// control inputs.

Status IsValidInputTensor(const Node\* node, int idx) const;

// Create and return a new WhileContext owned by this graph. This is called

// when a new while loop is created. `frame\_name` must be unique among

// WhileContexts in this graph.

Status AddWhileContext(StringPiece frame\_name, std::vector<Node\*> enter\_nodes,

std::vector<Node\*> exit\_nodes,

OutputTensor cond\_output,

std::vector<OutputTensor> body\_inputs,

std::vector<OutputTensor> body\_outputs,

WhileContext\*\* result);

// Builds a node name to node pointer index for all nodes in the graph.

std::unordered\_map<string, Node\*> BuildNodeNameIndex() const;

absl::optional<std::vector<bool>>& GetConstArgIndicesCache() const {

return const\_arg\_indices\_cache\_;

}

// TODO(josh11b): uint64 hash() const;

private:

// If cost\_node is non-null, then cost accounting (in CostModel)

// will be associated with that node rather than the new one being

// created.

//

// Ownership of the returned Node is not transferred to caller.

Node\* AllocateNode(std::shared\_ptr<NodeProperties> props,

const Node\* cost\_node, Node::NodeClass node\_class);

void ReleaseNode(Node\* node);

// Insert edge in free\_edges\_ for possible reuse.

void RecycleEdge(const Edge\* edge);

// Registry of all known ops, including functions.

FunctionLibraryDefinition ops\_;

// GraphDef versions

const std::unique\_ptr<VersionDef> versions\_;

// Allocator which will give us good locality.

core::Arena arena\_;

// Map from node ids to allocated nodes. nodes\_[id] may be nullptr if

// the node with that id was removed from the graph.

std::vector<Node\*> nodes\_;

// Number of nodes alive.

int64 num\_nodes\_ = 0;

// Map from edge ids to allocated edges. edges\_[id] may be nullptr if

// the edge with that id was removed from the graph.

std::vector<Edge\*> edges\_;

// The number of entries in edges\_ that are not nullptr.

int num\_edges\_ = 0;

// Allocated but free nodes and edges.

std::vector<Node\*> free\_nodes\_;

std::vector<Edge\*> free\_edges\_;

// For generating unique names.

int name\_counter\_ = 0;

// In most graphs, the number of unique values used for the

// Node::assigned\_device\_name() property is quite small. If the graph is

// large, then this duplication of values can consume a significant amount of

// memory. Instead, we represent the same information using an interning

// table, which consists of a vector of unique strings (device\_names\_), as

// well a map (device\_names\_map\_) from unique strings to indices within the

// unique string table.

//

// The InternDeviceName() method handles adding a new entry into the table,

// or locating the index of an existing entry.

//

// The fact that Node::assigned\_device\_name() is implemented using an

// interning table is intentionally public. This allows algorithms that

// frequently access this field to do so efficiently, especially for the case

// where the assigned\_device\_name of one Node is copied directly from that

// of another Node.

// A table of the unique assigned device names. Indices do NOT correspond

// to node IDs. Index 0 is always the empty string.

std::vector<string> device\_names\_;

// Maps unique device names to indices within device\_names\_[i].

std::unordered\_map<string, int> device\_names\_map\_;

// All the while contexts owned by this graph, keyed by frame name,

// corresponding to all the while loops contained in this graph (including

// nested loops). The stored contexts are usually accessed via

// AddWhileContext() or Node::while\_ctx(), but this manages the lifetime.

std::map<string, WhileContext> while\_ctxs\_;

// Cache of the indices of the arguments which need to be constant for the XLA

// compilation.

mutable absl::optional<std::vector<bool>> const\_arg\_indices\_cache\_;

TF\_DISALLOW\_COPY\_AND\_ASSIGN(Graph);

};

GraphDef is a serialization utility class which binds to Graph. The corresponding Protobuf interface is defined as:

// Represents the graph of operations

message GraphDef {

repeated NodeDef node = 1;

// Compatibility versions of the graph. See core/public/version.h for version

// history. The GraphDef version is distinct from the TensorFlow version, and

// each release of TensorFlow will support a range of GraphDef versions.

VersionDef versions = 4;

// Deprecated single version field; use versions above instead. Since all

// GraphDef changes before "versions" was introduced were forward

// compatible, this field is entirely ignored.

int32 version = 3 [deprecated = true];

// EXPERIMENTAL. DO NOT USE OR DEPEND ON THIS YET.

//

// "library" provides user-defined functions.

//

// Naming:

// \* library.function.name are in a flat namespace.

// NOTE: We may need to change it to be hierarchical to support

// different orgs. E.g.,

// { "/google/nn", { ... }},

// { "/google/vision", { ... }}

// { "/org\_foo/module\_bar", { ... }}

// map<string, FunctionDefLib> named\_lib;

// \* If node[i].op is the name of one function in "library",

// node[i] is deemed as a function call. Otherwise, node[i].op

// must be a primitive operation supported by the runtime.

//

//

// Function call semantics:

//

// \* The callee may start execution as soon as some of its inputs

// are ready. The caller may want to use Tuple() mechanism to

// ensure all inputs are ready in the same time.

//

// \* The consumer of return values may start executing as soon as

// the return values the consumer depends on are ready. The

// consumer may want to use Tuple() mechanism to ensure the

// consumer does not start until all return values of the callee

// function are ready.

FunctionDefLibrary library = 2;

}

# 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

# Exploring the C++ code examples

## Exploring //tensorflow/core/example

### Proto file for ***Feature***

This proto file contains protocol messages for describing features for machine learning model training or inference. There are three base ***Feature*** types:

* bytes
* float
* int64

A ***Feature*** contains Lists which may hold zero or more values. These lists are the base values ***BytesList***, ***FloatList***, ***Int64List***.

***Features*** are organized into categories by name. The ***Features*** message contains the mapping from the name to Feature. Here are example Features for a movie recommendation application:

*Feature {*

*key: “age”*

*value: { float\_list {*

*value: 29.0*

*}}*

*}*

*Feature {*

*key: “movie”*

*value: { bytes\_list {*

*value: “The Shawshank Redemption”*

*value: “Fight Clubs”*

*}}*

*}*

*Feature {*

*key: “movie\_ratings”*

*value: { float\_list {*

*value: 9.0,*

*value: 9.7*

*}}*

*}*

*Feature {*

*key: “suggestion”*

*value: { byte\_list {*

*value: “Inception”*

*}}*

*}*

*Feature {*

*key: “suggestion\_purchased”*

*value: { int64\_list {*

*value: 1*

*}}*

*}*

*Feature {*

*key: “purchase\_price”*

*value: { float\_list {*

*value: 9.99*

*}}*

*}*

|  |
| --- |
|  |

// containers to hold repeated fundamental values

message ByteList {

repeated bytes value = 1;

}

message FloatList {

repeated float value = 1 [packed = true];

}

message Int64List {

repeated int64 value = 1 [packed = true];

}

// Containers for non-sequential data.

message Feature {

// Each feature can be exactly one kind.

oneof kind {

BytesList bytes\_list = 1;

FloatList float\_list = 2;

Int64List int64\_list = 3;

}

}

message Features {

// Map from feature name to feature.

map<string, Feature> feature = 1;

}

// Containers for sequential data.

//

// A FeatureList contains lists of Features. These may hold zero or more

// Feature values.

//

// FeatureLists are organized into categories by name. The FeatureLists message

// contains the mapping from name to FeatureList.

//

message FeatureList {

repeated Feature feature = 1;

}

message FeatureLists {

// Map from feature name to feature list.

map<string, FeatureList> feature\_list = 1;

}

### Proto file for ***Example***:

An ***Example*** is a mostly-normalized data format for storing data for training and inference. It contains key-value store (features); where each key (string) maps to a Feature message which is one of packed BytesList, FloatList, or Int64List. This flexible and compact format allows the storage of large amounts of typed data, but it requires that the data shape and use be determined by the configuration files and parsers that are used to read and write that format. That is the ***Example*** is mostly not self-describing format. In TF, Examples are read in row-major format so any configuration that describes data with rank-2 or above should keep that in mind. For example, to store an M x N Matrix of Bytes, the BytesList must contain M\*N bytes with M rows of N contiguous values each. That is, the ByteList value must store the matrix as:

// .... row 0 .... .... row 1 .... // ........... // ... row M-1 ....

An Example for a movie recommendation application:

Features {

Feature {

key: "age"

value { float\_list {

value: 29.0

}}

}

Feature {

key: "movie"

value { bytes\_list {

value: "The Shawshank Redemption"

value: "Fight Club"

}}

}

Feature {

key: "movie\_ratings"

value { float\_list {

value: 9.0

value: 9.7

}}

}

Feature {

key: "suggestion"

value { bytes\_list {

value: "Inception"

}}

}

# Note that this feature exists to be used as a label in training.

# E.g., if training a logistic regression model to predict purchase

# probability in our learning tool we would set the label feature to

# "suggestion\_purchased".

Feature {

key: "suggestion\_purchased"

value { float\_list {

value: 1.0

}}

}

# Similar to "suggestion\_purchased" above this feature exists to be used

# as a label in training.

# E.g., if training a linear regression model to predict purchase

# price in our learning tool we would set the label feature to

# "purchase\_price".

// feature {

// key: "purchase\_price"

// value { float\_list {

// value: 9.99

// }}

// }

// }

//

# Building the tensorflow libraries and examples using Bazel

## Building tensorflow libraries

Building tensorflow libraries 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

Building tensorflow C++ api library using monolithic config

bazel build -c opt --config=monolithic //tensorflow:libtensorflow\_cc.so

## Building tensorflow core components

bazel build --config=opt //tensorflow/core:lib

exports the public non-test headers for:

//third\_party/tensorflow/core/platform: platform-specific code and external dependencies

lib/: Low-level libraries that are not TensorFlow-specific

bazel build --config=opt //tensorflow/core:framework

exports the public non-test headers for:

util/: General low-level TensorFlow-specific libraries

framework/: Support for adding new ops & kernels

example/: Wrappers to simplify access to Example proto

## Building Example Code

Building example code: parser configuration test

bazel build --config=opt //tensorflow/core/example:example\_parser\_configuration\_test

Building label\_image example code with debug symbols:

bazel build --config=opt --verbose\_failures -c dbg --strip=never tensorflow/examples/label\_image/...

Building label\_image example code with debug symbols and limited RAM resource (<2GB):

bazel build --config=opt --verbose\_failures -c dbg --strip=never --jobs 1 --local\_ram\_resources 2048 tensorflow/examples/label\_image/...

# 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* *aws-c-common aws-c-event-stream*

*aws-checksums bazel\_tools boringssl*

*com\_google\_absl com\_google\_protobuf com\_googlesource\_code\_re2*

*com\_google\_grpc\_grpc curl double\_conversion*

*eigen\_archive farmhash\_archive fft2d*

*gif highwayhash jsoncpp\_git*

*libjpeg\_turbo local\_config\_cc local\_config\_cuda*

*local\_config\_git local\_config\_python local\_config\_rocm*

*local\_config\_sycl local\_config\_tensorrt nasm*

*nsync snappy zlib*

Some of the important libraries to build a C++ tensorflow app using the C++ tensorflow shared library libtensorflow\_cc.so:

The location of protobuf source: ROOT\_FOLDER/bazel-tensorflow/external/com\_google\_protobuf

![A screenshot of a cell phone

Description automatically generated]()

The location of eigen source: ROOT\_FOLDER/bazel-tensorflow/external/eigen\_archive

![A picture containing monitor

Description automatically generated]()

The location of grpc source: ROOT\_FOLDER/bazel-tensorflow/external/com\_github\_grpc\_grpc

![A screen shot of a monitor

Description automatically generated]()

The location of absl source: ROOT\_FOLDER/bazel-tensorflow/external/com\_google\_absl



The location of zlib source: ROOT\_FOLDER/bazel-tensorflow/external/zlib

![A close up of a screen

Description automatically generated]()

# Build first C++ Tensorflow app

1. Build the tensorflow C++ API library with

bazel build -c opt --config=monolithic //tensorflow:libtensorflow\_cc.so

or

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

if we want to have DEBUG symbols available.

1. Build

# Third Party and External Packages

## **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>.

# Appendix A: Bazel tutorial for Tensorflow Builds

# Appendix B: Hashing in Tensorflow

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>

# Bibliography

1. *How to write shared libraries*, Ulrich Drepper, [drepper@gmail.com](mailto:drepper@gmail.com), Dec 10, 2011
2. *Semantic Versioning v2.0.0* <https://semver.org/spec/v2.0.0.html>