syntax = "proto3";

package tensorflow;

import "tensorflow/core/framework/cost\_graph.proto";

import "tensorflow/core/framework/graph.proto";

import "tensorflow/core/framework/step\_stats.proto";

import "tensorflow/core/protobuf/cluster.proto";

import "tensorflow/core/protobuf/coordination\_config.proto";

import "tensorflow/core/protobuf/debug.proto";

import "tensorflow/core/protobuf/rewriter\_config.proto";

option cc\_enable\_arenas = true;

option java\_outer\_classname = "ConfigProtos";

option java\_multiple\_files = true;

option java\_package = "org.tensorflow.framework";

option go\_package = "github.com/tensorflow/tensorflow/tensorflow/go/core/protobuf/for\_core\_protos\_go\_proto";

message GPUOptions {

// Fraction of the available GPU memory to allocate for each process.

// 1 means to allocate all of the GPU memory, 0.5 means the process

// allocates up to ~50% of the available GPU memory.

//

// GPU memory is pre-allocated unless the allow\_growth option is enabled.

//

// If greater than 1.0, uses CUDA unified memory to potentially oversubscribe

// the amount of memory available on the GPU device by using host memory as a

// swap space. Accessing memory not available on the device will be

// significantly slower as that would require memory transfer between the host

// and the device. Options to reduce the memory requirement should be

// considered before enabling this option as this may come with a negative

// performance impact. Oversubscription using the unified memory requires

// Pascal class or newer GPUs and it is currently only supported on the Linux

// operating system. See

// https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-requirements

// for the detailed requirements.

double per\_process\_gpu\_memory\_fraction = 1;

// If true, the allocator does not pre-allocate the entire specified

// GPU memory region, instead starting small and growing as needed.

bool allow\_growth = 4;

// The type of GPU allocation strategy to use.

//

// Allowed values:

// "": The empty string (default) uses a system-chosen default

// which may change over time.

//

// "BFC": A "Best-fit with coalescing" algorithm, simplified from a

// version of dlmalloc.

string allocator\_type = 2;

// Delay deletion of up to this many bytes to reduce the number of

// interactions with gpu driver code. If 0, the system chooses

// a reasonable default (several MBs).

int64 deferred\_deletion\_bytes = 3;

// A comma-separated list of GPU ids that determines the 'visible'

// to 'virtual' mapping of GPU devices. For example, if TensorFlow

// can see 8 GPU devices in the process, and one wanted to map

// visible GPU devices 5 and 3 as "/device:GPU:0", and "/device:GPU:1",

// then one would specify this field as "5,3". This field is similar in

// spirit to the CUDA\_VISIBLE\_DEVICES environment variable, except

// it applies to the visible GPU devices in the process.

//

// NOTE:

// 1. The GPU driver provides the process with the visible GPUs

// in an order which is not guaranteed to have any correlation to

// the \*physical\* GPU id in the machine. This field is used for

// remapping "visible" to "virtual", which means this operates only

// after the process starts. Users are required to use vendor

// specific mechanisms (e.g., CUDA\_VISIBLE\_DEVICES) to control the

// physical to visible device mapping prior to invoking TensorFlow.

// 2. In the code, the ids in this list are also called "platform GPU id"s,

// and the 'virtual' ids of GPU devices (i.e. the ids in the device

// name "/device:GPU:<id>") are also called "TF GPU id"s. Please

// refer to third\_party/tensorflow/core/common\_runtime/gpu/gpu\_id.h

// for more information.

string visible\_device\_list = 5;

// In the event polling loop sleep this many microseconds between

// PollEvents calls, when the queue is not empty. If value is not

// set or set to 0, gets set to a non-zero default.

int32 polling\_active\_delay\_usecs = 6;

// This field is deprecated and ignored.

int32 polling\_inactive\_delay\_msecs = 7;

// Force all tensors to be gpu\_compatible. On a GPU-enabled TensorFlow,

// enabling this option forces all CPU tensors to be allocated with Cuda

// pinned memory. Normally, TensorFlow will infer which tensors should be

// allocated as the pinned memory. But in case where the inference is

// incomplete, this option can significantly speed up the cross-device memory

// copy performance as long as it fits the memory.

// Note that this option is not something that should be

// enabled by default for unknown or very large models, since all Cuda pinned

// memory is unpageable, having too much pinned memory might negatively impact

// the overall host system performance.

bool force\_gpu\_compatible = 8;

message Experimental {

// Configuration for breaking down a visible GPU into multiple "virtual"

// devices.

message VirtualDevices {

// Per "virtual" device memory limit, in MB. The number of elements in

// the list is the number of virtual devices to create on the

// corresponding visible GPU (see "virtual\_devices" below).

// If empty, it will create single virtual device taking all available

// memory from the device.

//

// For the concept of "visible" and "virtual" GPU, see the comments for

// "visible\_device\_list" above for more information.

repeated float memory\_limit\_mb = 1;

// Priority values to use with the virtual devices. Use the cuda function

// cudaDeviceGetStreamPriorityRange to query for valid range of values for

// priority.

//

// On a P4000 GPU with cuda 10.1, the priority range reported was 0 for

// least priority and -1 for greatest priority.

//

// If this field is not specified, then the virtual devices will be

// created with the default. If this field has values set, then the size

// of this must match with the above memory\_limit\_mb.

repeated int32 priority = 2;

}

// The multi virtual device settings. If empty (not set), it will create

// single virtual device on each visible GPU, according to the settings

// in "visible\_device\_list" above. Otherwise, the number of elements in the

// list must be the same as the number of visible GPUs (after

// "visible\_device\_list" filtering if it is set), and the string represented

// device names (e.g. /device:GPU:<id>) will refer to the virtual

// devices and have the <id> field assigned sequentially starting from 0,

// according to the order they appear in this list and the "memory\_limit"

// list inside each element. For example,

// visible\_device\_list = "1,0"

// virtual\_devices { memory\_limit: 1GB memory\_limit: 2GB }

// virtual\_devices {}

// will create three virtual devices as:

// /device:GPU:0 -> visible GPU 1 with 1GB memory

// /device:GPU:1 -> visible GPU 1 with 2GB memory

// /device:GPU:2 -> visible GPU 0 with all available memory

//

// NOTE:

// 1. It's invalid to set both this and "per\_process\_gpu\_memory\_fraction"

// at the same time.

// 2. Currently this setting is per-process, not per-session. Using

// different settings in different sessions within same process will

// result in undefined behavior.

repeated VirtualDevices virtual\_devices = 1;

// If true, uses CUDA unified memory for memory allocations. If

// per\_process\_gpu\_memory\_fraction option is greater than 1.0, then unified

// memory is used regardless of the value for this field. See comments for

// per\_process\_gpu\_memory\_fraction field for more details and requirements

// of the unified memory. This option is useful to oversubscribe memory if

// multiple processes are sharing a single GPU while individually using less

// than 1.0 per process memory fraction.

bool use\_unified\_memory = 2;

// If > 1, the number of device-to-device copy streams to create

// for each GPUDevice. Default value is 0, which is automatically

// converted to 1.

int32 num\_dev\_to\_dev\_copy\_streams = 3;

// If non-empty, defines a good GPU ring order on a single worker based on

// device interconnect. This assumes that all workers have the same GPU

// topology. Specify as a comma-separated string, e.g. "3,2,1,0,7,6,5,4".

// This ring order is used by the RingReducer implementation of

// CollectiveReduce, and serves as an override to automatic ring order

// generation in OrderTaskDeviceMap() during CollectiveParam resolution.

string collective\_ring\_order = 4;

// If true then extra work is done by GPUDevice and GPUBFCAllocator to

// keep track of when GPU memory is freed and when kernels actually

// complete so that we can know when a nominally free memory chunk

// is really not subject to pending use.

bool timestamped\_allocator = 5;

// reserved id: 6

// Parameters for GPUKernelTracker. By default no kernel tracking is done.

// Note that timestamped\_allocator is only effective if some tracking is

// specified.

//

// If kernel\_tracker\_max\_interval = n > 0, then a tracking event

// is inserted after every n kernels without an event.

int32 kernel\_tracker\_max\_interval = 7;

// If kernel\_tracker\_max\_bytes = n > 0, then a tracking event is

// inserted after every series of kernels allocating a sum of

// memory >= n. If one kernel allocates b \* n bytes, then one

// event will be inserted after it, but it will count as b against

// the pending limit.

int32 kernel\_tracker\_max\_bytes = 8;

// If kernel\_tracker\_max\_pending > 0 then no more than this many

// tracking events can be outstanding at a time. An attempt to

// launch an additional kernel will stall until an event

// completes.

int32 kernel\_tracker\_max\_pending = 9;

// BFC Allocator can return an allocated chunk of memory upto 2x the

// requested size. For virtual devices with tight memory constraints, and

// proportionately large allocation requests, this can lead to a significant

// reduction in available memory. The threshold below controls when a chunk

// should be split if the chunk size exceeds requested memory size. It is

// expressed as a fraction of total available memory for the tf device. For

// example setting it to 0.05 would imply a chunk needs to be split if its

// size exceeds the requested memory by 5% of the total virtual device/gpu

// memory size.

double internal\_fragmentation\_fraction = 10;

// When true, use CUDA cudaMallocAsync API instead of TF gpu allocator.

bool use\_cuda\_malloc\_async = 11;

// By default, BFCAllocator may sleep when it runs out of memory, in the

// hopes that another thread will free up memory in the meantime. Setting

// this to true disables the sleep; instead we'll OOM immediately.

bool disallow\_retry\_on\_allocation\_failure = 12;

}

// Everything inside experimental is subject to change and is not subject

// to API stability guarantees in

// https://www.tensorflow.org/guide/version\_compat.

Experimental experimental = 9;

}

// Options passed to the graph optimizer

message OptimizerOptions {

// If true, optimize the graph using common subexpression elimination.

// Note: the optimization Level L1 will override this setting to true. So in

// order to disable common subexpression elimination the opt\_level has to be

// set to L0.

bool do\_common\_subexpression\_elimination = 1;

// If true, perform constant folding optimization on the graph.

// Note: the optimization Level L1 will override this setting to true. So in

// order to disable constant folding the opt\_level has to be set to L0.

bool do\_constant\_folding = 2;

// Constant folding optimization replaces tensors whose values can be

// predetermined, with constant nodes. To avoid inserting too large constants,

// the size of each constant created can be limited. If this value is zero, a

// default limit of 10 MiB will be applied. If constant folding optimization

// is disabled, this value is ignored.

int64 max\_folded\_constant\_in\_bytes = 6;

// If true, perform function inlining on the graph.

bool do\_function\_inlining = 4;

// Optimization level

enum Level {

// L1 is the default level.

// Optimization performed at L1 :

// 1. Common subexpression elimination

// 2. Constant folding

L1 = 0;

// No optimizations

L0 = -1;

}

// Overall optimization level. The actual optimizations applied will be the

// logical OR of the flags that this level implies and any flags already set.

Level opt\_level = 3;

// Control the use of the compiler/jit. Experimental.

enum GlobalJitLevel {

DEFAULT = 0; // Default setting ("off" now, but later expected to be "on")

OFF = -1;

// The following settings turn on compilation, with higher values being

// more aggressive. Higher values may reduce opportunities for parallelism

// and may use more memory. (At present, there is no distinction, but this

// is expected to change.)

ON\_1 = 1;

ON\_2 = 2;

}

GlobalJitLevel global\_jit\_level = 5;

// CPU code will be autoclustered only if global\_jit\_level >= ON\_1 and either:

// - this flag is true, or

// - TF\_XLA\_FLAGS contains --tf\_xla\_cpu\_global\_jit=true.

bool cpu\_global\_jit = 7;

}

message GraphOptions {

// Removed, use optimizer\_options below.

reserved "skip\_common\_subexpression\_elimination";

reserved 1;

// If true, use control flow to schedule the activation of Recv nodes.

// (Currently ignored.)

bool enable\_recv\_scheduling = 2;

// Options controlling how graph is optimized.

OptimizerOptions optimizer\_options = 3;

// The number of steps to run before returning a cost model detailing

// the memory usage and performance of each node of the graph. 0 means

// no cost model.

int64 build\_cost\_model = 4;

// The number of steps to skip before collecting statistics for the

// cost model.

int64 build\_cost\_model\_after = 9;

// Annotate each Node with Op output shape data, to the extent it can

// be statically inferred.

bool infer\_shapes = 5;

// Only place the subgraphs that are run, rather than the entire graph.

//

// This is useful for interactive graph building, where one might

// produce graphs that cannot be placed during the debugging

// process. In particular, it allows the client to continue work in

// a session after adding a node to a graph whose placement

// constraints are unsatisfiable.

bool place\_pruned\_graph = 6;

// If true, transfer float values between processes as bfloat16.

bool enable\_bfloat16\_sendrecv = 7;

// If > 0, record a timeline every this many steps.

// EXPERIMENTAL: This currently has no effect in MasterSession.

int32 timeline\_step = 8;

// Options that control the type and amount of graph rewriting.

// Not currently configurable via the public Python API (i.e. there is no API

// stability guarantee if you import RewriterConfig explicitly).

RewriterConfig rewrite\_options = 10;

}

message ThreadPoolOptionProto {

// The number of threads in the pool.

//

// 0 means the system picks a value based on where this option proto is used

// (see the declaration of the specific field for more info).

int32 num\_threads = 1;

// The global name of the threadpool.

//

// If empty, then the threadpool is made and used according to the scope it's

// in - e.g., for a session threadpool, it is used by that session only.

//

// If non-empty, then:

// - a global threadpool associated with this name is looked

// up or created. This allows, for example, sharing one threadpool across

// many sessions (e.g., like the default behavior, if

// inter\_op\_parallelism\_threads is not configured), but still partitioning

// into a large and small pool.

// - if the threadpool for this global\_name already exists, then it is an

// error if the existing pool was created using a different num\_threads

// value as is specified on this call.

// - threadpools created this way are never garbage collected.

string global\_name = 2;

}

message RPCOptions {

// If true, always use RPC to contact the session target.

//

// If false (the default option), TensorFlow may use an optimized

// transport for client-master communication that avoids the RPC

// stack. This option is primarily for used testing the RPC stack.

bool use\_rpc\_for\_inprocess\_master = 1;

// The compression algorithm to be used. One of "deflate", "gzip".

string compression\_algorithm = 2;

// If compression\_algorithm is set, the compression level to be used.

// From 0 (no compression), up to 3.

int32 compression\_level = 3;

// Setting cache\_rpc\_response to true will enable sender side caching of

// response for RecvTensorAsync and RecvBufAsync to allow receiver to retry

// requests . This is only necessary when the network fabric is experiencing a

// significant error rate. Without it we'll fail a step on an network error,

// while with it we'll be able to complete long steps (like complex

// initializations) in the face of some network errors during RecvTensor.

bool cache\_rpc\_response = 4;

// Disables TCP connection sharing when opening a new RPC channel.

bool disable\_session\_connection\_sharing = 5;

// Setting num\_channels\_per\_target > 0 allows uses of multiple channels to

// communicate to the same target. This can be used to improve the aggregate

// throughput on high speed links (e.g 100G) where single connection is not

// sufficient to maximize link utilization. Note that a single RPC only goes

// on a single channel, this only helps in situations where there are multiple

// transfers to the same target overlapping in time.

int32 num\_channels\_per\_target = 6;

}

// Metadata about the session.

//

// This can be used by the runtime and the Ops for debugging, monitoring, etc.

//

// The (name, version) tuple is expected to be a unique identifier for

// sessions within the same process.

//

// NOTE: This is currently used and propagated only by the direct session.

message SessionMetadata {

string name = 1;

// The version is optional. If set, needs to be >= 0.

int64 version = 2;

}

// Session configuration parameters.

// The system picks appropriate values for fields that are not set.

message ConfigProto {

// Map from device type name (e.g., "CPU" or "GPU" ) to maximum

// number of devices of that type to use. If a particular device

// type is not found in the map, the system picks an appropriate

// number.

map<string, int32> device\_count = 1;

// The execution of an individual op (for some op types) can be

// parallelized on a pool of intra\_op\_parallelism\_threads.

// 0 means the system picks an appropriate number.

//

// If you create an ordinary session, e.g., from Python or C++,

// then there is exactly one intra op thread pool per process.

// The first session created determines the number of threads in this pool.

// All subsequent sessions reuse/share this one global pool.

//

// There are notable exceptions to the default behavior described above:

// 1. There is an environment variable for overriding this thread pool,

// named TF\_OVERRIDE\_GLOBAL\_THREADPOOL.

// 2. When connecting to a server, such as a remote `tf.train.Server`

// instance, then this option will be ignored altogether.

int32 intra\_op\_parallelism\_threads = 2;

// Nodes that perform blocking operations are enqueued on a pool of

// inter\_op\_parallelism\_threads available in each process.

//

// 0 means the system picks an appropriate number.

// Negative means all operations are performed in caller's thread.

//

// Note that the first Session created in the process sets the

// number of threads for all future sessions unless use\_per\_session\_threads is

// true or session\_inter\_op\_thread\_pool is configured.

int32 inter\_op\_parallelism\_threads = 5;

// If true, use a new set of threads for this session rather than the global

// pool of threads. Only supported by direct sessions.

//

// If false, use the global threads created by the first session, or the

// per-session thread pools configured by session\_inter\_op\_thread\_pool.

//

// This option is deprecated. The same effect can be achieved by setting

// session\_inter\_op\_thread\_pool to have one element, whose num\_threads equals

// inter\_op\_parallelism\_threads.

bool use\_per\_session\_threads = 9;

// This option is experimental - it may be replaced with a different mechanism

// in the future.

//

// Configures session thread pools. If this is configured, then RunOptions for

// a Run call can select the thread pool to use.

//

// The intended use is for when some session invocations need to run in a

// background pool limited to a small number of threads:

// - For example, a session may be configured to have one large pool (for

// regular compute) and one small pool (for periodic, low priority work);

// using the small pool is currently the mechanism for limiting the inter-op

// parallelism of the low priority work. Note that it does not limit the

// parallelism of work spawned by a single op kernel implementation.

// - Using this setting is normally not needed in training, but may help some

// serving use cases.

// - It is also generally recommended to set the global\_name field of this

// proto, to avoid creating multiple large pools. It is typically better to

// run the non-low-priority work, even across sessions, in a single large

// pool.

repeated ThreadPoolOptionProto session\_inter\_op\_thread\_pool = 12;

// Assignment of Nodes to Devices is recomputed every placement\_period

// steps until the system warms up (at which point the recomputation

// typically slows down automatically).

int32 placement\_period = 3;

// When any filters are present sessions will ignore all devices which do not

// match the filters. Each filter can be partially specified, e.g. "/job:ps"

// "/job:worker/replica:3", etc.

repeated string device\_filters = 4;

// Options that apply to all GPUs.

GPUOptions gpu\_options = 6;

// Whether soft placement is allowed. If allow\_soft\_placement is true,

// an op will be placed on CPU if

// 1. there's no GPU implementation for the OP

// or

// 2. no GPU devices are known or registered

// or

// 3. need to co-locate with reftype input(s) which are from CPU.

bool allow\_soft\_placement = 7;

// Whether device placements should be logged.

bool log\_device\_placement = 8;

// Options that apply to all graphs.

GraphOptions graph\_options = 10;

// Global timeout for all blocking operations in this session. If non-zero,

// and not overridden on a per-operation basis, this value will be used as the

// deadline for all blocking operations.

int64 operation\_timeout\_in\_ms = 11;

// Options that apply when this session uses the distributed runtime.

RPCOptions rpc\_options = 13;

// Optional list of all workers to use in this session.

ClusterDef cluster\_def = 14;

// If true, any resources such as Variables used in the session will not be

// shared with other sessions. However, when clusterspec propagation is

// enabled, this field is ignored and sessions are always isolated.

bool isolate\_session\_state = 15;

// When true, WorkerSessions are created with device attributes from the

// full cluster.

// This is helpful when a worker wants to partition a graph

// (for example during a PartitionedCallOp).

bool share\_cluster\_devices\_in\_session = 17;

// Everything inside Experimental is subject to change and is not subject

// to API stability guarantees in

// https://www.tensorflow.org/guide/version\_compat.

message Experimental {

// Task name for group resolution.

string collective\_group\_leader = 1;

// We removed the flag client\_handles\_error\_formatting. Marking the tag

// number as reserved.

// TODO(shikharagarwal): Should we just remove this tag so that it can be

// used in future for other purpose?

reserved 2;

// Which executor to use, the default executor will be used

// if it is an empty string or "DEFAULT"

string executor\_type = 3;

// Guidance to formatting of large RecvBuf fields for transfer.

// Any positive value sets the max chunk size. 0 defaults to 4096.

// Any negative value indicates no max, i.e. one chunk only.

int32 recv\_buf\_max\_chunk = 4;

// If true, and supported by the platform, the runtime will attempt to

// use NUMA affinity where applicable. One consequence will be the

// existence of as many CPU devices as there are available NUMA nodes.

bool use\_numa\_affinity = 5;

// If true, make collective op execution order sequential and deterministic

// for potentially concurrent collective instances.

bool collective\_deterministic\_sequential\_execution = 6;

// If true, use NCCL for CollectiveOps. This feature is highly

// experimental.

bool collective\_nccl = 7;

// In the following, session state means the value of a variable, elements

// in a hash table, or any other resource, accessible by worker sessions

// held by a TF server.

//

// When ClusterSpec propagation is enabled, the value of

// isolate\_session\_state is ignored when deciding whether to share session

// states in a TF server (for backwards compatibility reasons).

// - If share\_session\_state\_in\_clusterspec\_propagation is true, the session

// states are shared.

// - If share\_session\_state\_in\_clusterspec\_propagation is false, session

// states are isolated.

//

// When clusterspec propagation is not used, the value of

// share\_session\_state\_in\_clusterspec\_propagation is ignored when deciding

// whether to share session states in a TF server.

// - If isolate\_session\_state is true, session states are isolated.

// - If isolate\_session\_state is false, session states are shared.

//

// TODO(b/129330037): Add a single API that consistently treats

// isolate\_session\_state and ClusterSpec propagation.

bool share\_session\_state\_in\_clusterspec\_propagation = 8;

// If using a direct session, disable spinning while waiting for work in

// the thread pool. This may result in higher latency for completing ops,

// but in the case where there is a lot of spinning may result in lower

// CPU usage.

bool disable\_thread\_spinning = 9;

// This was promoted to a non-experimental API. Please use

// ConfigProto.share\_cluster\_devices\_in\_session instead.

bool share\_cluster\_devices\_in\_session = 10;

// Metadata about the session.

//

// If set, this can be used by the runtime and the Ops for debugging,

// monitoring, etc.

//

// NOTE: This is currently used and propagated only by the direct session.

SessionMetadata session\_metadata = 11;

// If true, the session may treat the graph as being static for optimization

// purposes.

//

// If this option is set to true when a session is created, the full

// GraphDef must be passed in a single call to Session::Create(), and

// Session::Extend() may not be supported.

bool optimize\_for\_static\_graph = 12;

// This field will eventually be deprecated and replaced by

// mlir\_bridge\_rollout (b/166038521).

//

// Whether to enable the MLIR-based TF->XLA bridge.

//

// This is a replacement to the existing bridge, and not ready for

// production usage yet.

// If this option is set to true when a session is created, MLIR is used to

// perform the set of graph transformations to put the graph in a form that

// can be executed with delegation of some computations to an accelerator.

// This builds on the model of XLA where a subset of the graph is

// encapsulated and attached to a "compile" operation, whose result is fed

// to an "execute" operation. The kernel for these operations is responsible

// to lower the encapsulated graph to a particular device.

bool enable\_mlir\_bridge = 13;

// An enum that describes the state of the MLIR bridge rollout.

enum MlirBridgeRollout {

// If this field is left unspecified, the MLIR bridge may be selectively

// enabled on a per graph basis.

MLIR\_BRIDGE\_ROLLOUT\_UNSPECIFIED = 0;

// Enabling the MLIR bridge enables it for all graphs in this session.

MLIR\_BRIDGE\_ROLLOUT\_ENABLED = 1;

// Disabling the MLIR bridge disables it for all graphs in this session.

MLIR\_BRIDGE\_ROLLOUT\_DISABLED = 2;

// Enable the MLIR bridge on a per graph basis based on an analysis of

// the features used in the graph. If the features used by the graph are

// supported by the MLIR bridge, the MLIR bridge will be used to run the

// graph.

MLIR\_BRIDGE\_ROLLOUT\_SAFE\_MODE\_ENABLED = 3;

// Enable the MLIR bridge in a fallback mode on a per graph basis based

// on an analysis of the features used in the graph.

// Running the MLIR bridge in the fallback mode means that it is

// executed and it commits all the changes to the TF graph in case

// of success. And it does not in case of failures and let the old bridge

// to process the TF graph.

MLIR\_BRIDGE\_ROLLOUT\_SAFE\_MODE\_FALLBACK\_ENABLED = 4;

}

// This field is underdevelopment, for now use enable\_mlir\_bridge

// (b/166038521).

//

// Whether to enable the MLIR-based TF->XLA bridge.

MlirBridgeRollout mlir\_bridge\_rollout = 17;

// Whether to enable the MLIR-based Graph optimizations.

//

// This will become a part of standard Tensorflow graph optimization

// pipeline, currently this is only used for gradual migration and testing

// new passes that are replacing existing optimizations in Grappler.

bool enable\_mlir\_graph\_optimization = 16;

// If true, the session will not store an additional copy of the graph for

// each subgraph.

//

// If this option is set to true when a session is created, the

// `RunOptions.output\_partition\_graphs` options must not be set.

bool disable\_output\_partition\_graphs = 14;

// Minimum number of batches run through the XLA graph before XLA fusion

// autotuner is enabled. Default value of zero disables the autotuner.

//

// The XLA fusion autotuner can improve performance by executing a heuristic

// search on the compiler parameters.

int64 xla\_fusion\_autotuner\_thresh = 15;

// Whether runtime execution uses TFRT.

bool use\_tfrt = 18;

// The field "coordination\_service was previously specified as a string;

// this has been replaced with a message below.

reserved 19;

// We removed the flag fetch\_remote\_devices\_in\_multi\_client. Marking the tag

// number as reserved.

reserved 20;

// Whether functional control flow op lowering should be disabled. This is

// useful when executing within a portable runtime where control flow op

// kernels may not be loaded due to selective registration.

bool disable\_functional\_ops\_lowering = 21;

// Provides a hint to XLA auto clustering to prefer forming a single large

// cluster that encompases most of the graph.

bool xla\_prefer\_single\_graph\_cluster = 22;

// Distributed coordination service configurations.

CoordinationServiceConfig coordination\_config = 23;

// Next: 24

}

Experimental experimental = 16;

// Next: 18

}

// Options for a single Run() call.

message RunOptions {

// TODO(pbar) Turn this into a TraceOptions proto which allows

// tracing to be controlled in a more orthogonal manner?

enum TraceLevel {

NO\_TRACE = 0;

SOFTWARE\_TRACE = 1;

HARDWARE\_TRACE = 2;

FULL\_TRACE = 3;

}

TraceLevel trace\_level = 1;

// Time to wait for operation to complete in milliseconds.

int64 timeout\_in\_ms = 2;

// The thread pool to use, if session\_inter\_op\_thread\_pool is configured.

// To use the caller thread set this to -1 - this uses the caller thread

// to execute Session::Run() and thus avoids a context switch. Using the

// caller thread to execute Session::Run() should be done ONLY for simple

// graphs, where the overhead of an additional context switch is

// comparable with the overhead of Session::Run().

int32 inter\_op\_thread\_pool = 3;

// Whether the partition graph(s) executed by the executor(s) should be

// outputted via RunMetadata.

bool output\_partition\_graphs = 5;

// EXPERIMENTAL. Options used to initialize DebuggerState, if enabled.

DebugOptions debug\_options = 6;

// When enabled, causes tensor allocation information to be included in

// the error message when the Run() call fails because the allocator ran

// out of memory (OOM).

//

// Enabling this option can slow down the Run() call.

bool report\_tensor\_allocations\_upon\_oom = 7;

// Everything inside Experimental is subject to change and is not subject

// to API stability guarantees in

// https://www.tensorflow.org/guide/version\_compat.

message Experimental {

// If non-zero, declares that this graph is going to use collective

// ops and must synchronize step\_ids with any other graph with this

// same group\_key value (in a distributed computation where tasks

// run disjoint graphs).

int64 collective\_graph\_key = 1;

// If true, then operations (using the inter-op pool) across all

// session::run() calls will be centrally scheduled, optimizing for (median

// and tail) latency.

// Consider using this option for CPU-bound workloads like inference.

bool use\_run\_handler\_pool = 2;

// Options for run handler thread pool.

message RunHandlerPoolOptions {

// Priority of the request. The run handler thread pool will schedule ops

// based on the priority number. The larger number means higher priority.

int64 priority = 1;

}

RunHandlerPoolOptions run\_handler\_pool\_options = 3;

}

Experimental experimental = 8;

reserved 4;

}

// Metadata output (i.e., non-Tensor) for a single Run() call.

message RunMetadata {

// Statistics traced for this step. Populated if tracing is turned on via the

// "RunOptions" proto.

// EXPERIMENTAL: The format and set of events may change in future versions.

StepStats step\_stats = 1;

// The cost graph for the computation defined by the run call.

CostGraphDef cost\_graph = 2;

// Graphs of the partitions executed by executors.

repeated GraphDef partition\_graphs = 3;

message FunctionGraphs {

// TODO(nareshmodi): Include some sort of function/cache-key identifier?

repeated GraphDef partition\_graphs = 1;

GraphDef pre\_optimization\_graph = 2;

GraphDef post\_optimization\_graph = 3;

}

// This is only populated for graphs that are run as functions in TensorFlow

// V2. There will be an entry below for each function that is traced.

// The main use cases of the post\_optimization\_graph and the partition\_graphs

// is to give the caller insight into the graphs that were actually run by the

// runtime. Additional information (such as those in step\_stats) will match

// these graphs.

// We also include the pre\_optimization\_graph since it is usually easier to

// read, and is helpful in situations where the caller wants to get a high

// level idea of what the built graph looks like (since the various graph

// optimization passes might change the structure of the graph significantly).

repeated FunctionGraphs function\_graphs = 4;

}

// Defines a connection between two tensors in a `GraphDef`.

message TensorConnection {

// A tensor name. The value of this tensor will be substituted for

// the tensor named in `to\_tensor`.

string from\_tensor = 1;

// A tensor name. The value of this tensor will be bound to the

// value of the tensor named in `from\_tensor`.

string to\_tensor = 2;

}

// Defines a subgraph in another `GraphDef` as a set of feed points and nodes

// to be fetched or executed.

//

// Compare with the arguments to `Session::Run()`.

message CallableOptions {

// Tensors to be fed in the callable. Each feed is the name of a tensor.

repeated string feed = 1;

// Fetches. A list of tensor names. The caller of the callable expects a

// tensor to be returned for each fetch[i] (see RunStepResponse.tensor). The

// order of specified fetches does not change the execution order.

repeated string fetch = 2;

// Target Nodes. A list of node names. The named nodes will be run by the

// callable but their outputs will not be returned.

repeated string target = 3;

// Options that will be applied to each run.

RunOptions run\_options = 4;

// Tensors to be connected in the callable. Each TensorConnection denotes

// a pair of tensors in the graph, between which an edge will be created

// in the callable.

repeated TensorConnection tensor\_connection = 5;

// The Tensor objects fed in the callable and fetched from the callable

// are expected to be backed by host (CPU) memory by default.

//

// The options below allow changing that - feeding tensors backed by

// device memory, or returning tensors that are backed by device memory.

//

// The maps below map the name of a feed/fetch tensor (which appears in

// 'feed' or 'fetch' fields above), to the fully qualified name of the device

// owning the memory backing the contents of the tensor.

//

// For example, creating a callable with the following options:

//

// CallableOptions {

// feed: "a:0"

// feed: "b:0"

//

// fetch: "x:0"

// fetch: "y:0"

//

// feed\_devices: {

// "a:0": "/job:localhost/replica:0/task:0/device:GPU:0"

// }

//

// fetch\_devices: {

// "y:0": "/job:localhost/replica:0/task:0/device:GPU:0"

// }

// }

//

// means that the Callable expects:

// - The first argument ("a:0") is a Tensor backed by GPU memory.

// - The second argument ("b:0") is a Tensor backed by host memory.

// and of its return values:

// - The first output ("x:0") will be backed by host memory.

// - The second output ("y:0") will be backed by GPU memory.

//

// FEEDS:

// It is the responsibility of the caller to ensure that the memory of the fed

// tensors will be correctly initialized and synchronized before it is

// accessed by operations executed during the call to Session::RunCallable().

//

// This is typically ensured by using the TensorFlow memory allocators

// (Device::GetAllocator()) to create the Tensor to be fed.

//

// Alternatively, for CUDA-enabled GPU devices, this typically means that the

// operation that produced the contents of the tensor has completed, i.e., the

// CUDA stream has been synchronized (e.g., via cuCtxSynchronize() or

// cuStreamSynchronize()).

map<string, string> feed\_devices = 6;

map<string, string> fetch\_devices = 7;

// By default, RunCallable() will synchronize the GPU stream before returning

// fetched tensors on a GPU device, to ensure that the values in those tensors

// have been produced. This simplifies interacting with the tensors, but

// potentially incurs a performance hit.

//

// If this options is set to true, the caller is responsible for ensuring

// that the values in the fetched tensors have been produced before they are

// used. The caller can do this by invoking `Device::Sync()` on the underlying

// device(s), or by feeding the tensors back to the same Session using

// `feed\_devices` with the same corresponding device name.

bool fetch\_skip\_sync = 8;

// Next: 9

}