#include "block\_format\_reader.h"

#include "tensorflow/core/framework/dataset.h"

#include "tensorflow/core/framework/partial\_tensor\_shape.h"

#include "tensorflow/core/framework/tensor.h"

#include "tensorflow/core/lib/io/random\_inputstream.h"

#if !defined(DISABLE\_ZLIB)

#include "tensorflow/core/lib/io/zlib\_inputstream.h"

#endif

#include <twml.h>

#include <cstdio>

#include <algorithm>

#include <iterator>

using namespace tensorflow;

inline std::string stripPath(std::string const &file\_name) {

const auto pos = file\_name.find\_last\_of("/");

if (pos == std::string::npos) return file\_name;

return file\_name.substr(pos + 1);

}

inline std::string getExtension(std::string const &file\_name) {

const auto stripped\_file\_name = stripPath(file\_name);

const auto pos = stripPath(stripped\_file\_name).find\_last\_of(".");

if (pos == std::string::npos) return "";

return stripped\_file\_name.substr(pos + 1);

}

REGISTER\_OP("BlockFormatDatasetV2")

.Input("filenames: string")

.Input("compression\_type: string")

.Input("buffer\_size: int64")

.Output("handle: variant")

.SetIsStateful()

.SetShapeFn(shape\_inference::ScalarShape)

.Doc(R"doc(

Creates a dataset for streaming BlockFormat data in compressed (e.g. gzip), uncompressed formats.

This op also has the ability stream a dataset containing files from multiple formats mentioned above.

filenames: A scalar or vector containing the name(s) of the file(s) to be read.

compression\_type: A scalar string denoting the compression type. Can be 'none', 'zlib', 'auto'.

buffer\_size: A scalar denoting the buffer size to use during decompression.

Outputs

handle: A handle to the dataset. This handle is later used to create an iterator to stream the data from the dataset.

)doc");

class BlockFormatDatasetV2 : public DatasetOpKernel {

public:

using DatasetOpKernel::DatasetOpKernel;

void MakeDataset(OpKernelContext\* ctx, DatasetBase \*\*output) override {

const Tensor\* filenames\_tensor;

OP\_REQUIRES\_OK(ctx, ctx->input("filenames", &filenames\_tensor));

OP\_REQUIRES(

ctx, filenames\_tensor->dims() <= 1,

errors::InvalidArgument("`filenames` must be a scalar or a vector."));

const auto filenames\_flat = filenames\_tensor->flat<string>();

const int64 num\_files = filenames\_tensor->NumElements();

std::vector<string> filenames;

filenames.reserve(num\_files);

std::copy(filenames\_flat.data(),

filenames\_flat.data() + num\_files,

std::back\_inserter(filenames));

string compression\_type;

OP\_REQUIRES\_OK(

ctx, tensorflow::data::ParseScalarArgument<string>(

ctx, "compression\_type", &compression\_type));

int64 buffer\_size = -1;

OP\_REQUIRES\_OK(

ctx, tensorflow::data::ParseScalarArgument<int64>(

ctx, "buffer\_size", &buffer\_size));

OP\_REQUIRES(ctx, buffer\_size >= 0,

errors::InvalidArgument(

"`buffer\_size` must be >= 0 (0 == no buffering)"));

OP\_REQUIRES(ctx,

compression\_type == "auto" ||

compression\_type == "gz" ||

compression\_type == "",

errors::InvalidArgument("Unknown extension: ", compression\_type));

\*output = new Dataset(ctx, std::move(filenames), compression\_type, buffer\_size);

}

private:

class Dataset : public DatasetBase {

public:

Dataset(OpKernelContext\* ctx,

std::vector<string> filenames,

std::string compression\_type,

int64 buffer\_size)

: DatasetBase(DatasetContext(ctx)),

compression\_type\_(compression\_type),

buffer\_size\_(buffer\_size),

filenames\_(std::move(filenames))

{}

const DataTypeVector& output\_dtypes() const override {

static DataTypeVector\* dtypes = new DataTypeVector({DT\_STRING});

return \*dtypes;

}

const std::vector<PartialTensorShape>& output\_shapes() const override {

static std::vector<PartialTensorShape>\* shapes =

new std::vector<PartialTensorShape>({{}});

return \*shapes;

}

string DebugString() const override { return "BlockFormatDatasetV2::Dataset"; }

protected:

Status AsGraphDefInternal(SerializationContext\* ctx,

DatasetGraphDefBuilder\* b,

Node\*\* output) const override {

Node\* filenames = nullptr;

Node\* compression\_type = nullptr;

Node\* buffer\_size = nullptr;

TF\_RETURN\_IF\_ERROR(b->AddVector(filenames\_, &filenames));

TF\_RETURN\_IF\_ERROR(b->AddScalar(compression\_type\_, &compression\_type));

TF\_RETURN\_IF\_ERROR(

b->AddScalar(buffer\_size\_, &buffer\_size));

TF\_RETURN\_IF\_ERROR(b->AddDataset(

this, {filenames, compression\_type, buffer\_size}, output));

return Status::OK();

}

private:

std::unique\_ptr<IteratorBase> MakeIteratorInternal(

const string& prefix) const override {

return std::unique\_ptr<IteratorBase>(

new Iterator({this, strings::StrCat(prefix, "::BlockFormat")}));

}

class Iterator : public DatasetIterator<Dataset> {

public:

explicit Iterator(const Params &params)

: DatasetIterator<Dataset>(params) {}

Status GetNextInternal(IteratorContext\* ctx,

std::vector<Tensor>\* out\_tensors,

bool\* end\_of\_sequence) override {

mutex\_lock l(mu\_);

do {

// We are currently processing a file, so try to read the next record.

if (reader\_) {

Tensor result\_tensor(cpu\_allocator(), DT\_STRING, {});

Status s = reader\_->ReadNext(&result\_tensor.scalar<string>()());

if (s.ok()) {

out\_tensors->emplace\_back(std::move(result\_tensor));

\*end\_of\_sequence = false;

return Status::OK();

} else if (!errors::IsOutOfRange(s)) {

return s;

}

// We have reached the end of the current file, so maybe

// move on to next file.

reader\_.reset();

++current\_file\_index\_;

}

// Iteration ends when there are no more files to process.

if (current\_file\_index\_ == dataset()->filenames\_.size()) {

\*end\_of\_sequence = true;

return Status::OK();

}

// Actually move on to next file.

const string& next\_filename =

dataset()->filenames\_[current\_file\_index\_];

auto compression\_type = dataset()->compression\_type\_;

int64 buffer\_size = dataset()->buffer\_size\_;

if (compression\_type == "auto") {

compression\_type = getExtension(next\_filename);

}

if (compression\_type != "gz" && compression\_type != "") {

return errors::InvalidArgument("Unknown extension: ", compression\_type);

}

tensorflow::Env\* env = tensorflow::Env::Default();

TF\_CHECK\_OK(env->NewRandomAccessFile(next\_filename, &file\_));

// RandomAccessInputstream defaults the second param to "false".

// The second parameter "false" is the key issue.

// "false" assumes the ownership of the file is elsewhere.

// But making that "true" causes segfaults down the line.

// So keep the ownership of "file\_" in this class and clean up properly.

file\_stream\_.reset(new tensorflow::io::RandomAccessInputStream(file\_.get(), false));

if (compression\_type == "gz") {

// unpack\_stream does not take ownership of file\_stream\_

#if !defined(DISABLE\_ZLIB)

unpack\_stream\_.reset(new tensorflow::io::ZlibInputStream(

file\_stream\_.get(),

buffer\_size,

buffer\_size,

tensorflow::io::ZlibCompressionOptions::GZIP()));

reader\_.reset(new BlockFormatReader(unpack\_stream\_.get()));

#else

return errors::InvalidArgument("libtwml compiled without zlib support");

#endif

} else {

unpack\_stream\_.reset(nullptr);

reader\_.reset(new BlockFormatReader(file\_stream\_.get()));

}

} while (true);

}

private:

mutex mu\_;

uint64\_t current\_file\_index\_ GUARDED\_BY(mu\_) = 0;

std::unique\_ptr<tensorflow::RandomAccessFile> file\_;

std::unique\_ptr<tensorflow::io::InputStreamInterface> file\_stream\_;

std::unique\_ptr<tensorflow::io::InputStreamInterface> unpack\_stream\_;

std::unique\_ptr<BlockFormatReader> reader\_ GUARDED\_BY(mu\_);

};

const std::string compression\_type\_;

const int64 buffer\_size\_;

const std::vector<string> filenames\_;

};

};

REGISTER\_KERNEL\_BUILDER(

Name("BlockFormatDatasetV2")

.Device(DEVICE\_CPU),

BlockFormatDatasetV2);