#include "internal/thrift.h"

#include "internal/error.h"

#include <string>

#include <twml/TensorRecordReader.h>

#include <twml/RawTensor.h>

namespace twml {

template<typename T> struct TensorTraits;

#define INSTANTIATE(TYPE, THRIFT\_TYPE, TWML\_TYPE) \

template<> struct TensorTraits<TYPE> { \

static const TTYPES ThriftType = THRIFT\_TYPE; \

static const twml\_type TwmlType = TWML\_TYPE; \

}; \

INSTANTIATE(int64\_t, TTYPE\_I64, TWML\_TYPE\_INT64)

INSTANTIATE(int32\_t, TTYPE\_I32, TWML\_TYPE\_INT32)

INSTANTIATE(double, TTYPE\_DOUBLE, TWML\_TYPE\_DOUBLE)

INSTANTIATE(bool, TTYPE\_BOOL, TWML\_TYPE\_BOOL)

static

std::vector<uint64\_t> calcStrides(const std::vector<uint64\_t> &shape) {

int ndims = static\_cast<int>(shape.size());

std::vector<uint64\_t> strides(ndims);

uint64\_t stride = 1;

for (int i = ndims-1; i >= 0; i--) {

strides[i] = stride;

stride \*= shape[i];

}

return strides;

}

static twml\_type getTwmlType(int dtype) {

// Convert tensor.thrift enum to twml enum

switch (dtype) {

case DATA\_TYPE\_FLOAT:

return TWML\_TYPE\_FLOAT;

case DATA\_TYPE\_DOUBLE:

return TWML\_TYPE\_DOUBLE;

case DATA\_TYPE\_INT64:

return TWML\_TYPE\_INT64;

case DATA\_TYPE\_INT32:

return TWML\_TYPE\_INT32;

case DATA\_TYPE\_UINT8:

return TWML\_TYPE\_UINT8;

case DATA\_TYPE\_STRING:

return TWML\_TYPE\_STRING;

case DATA\_TYPE\_BOOL:

return TWML\_TYPE\_BOOL;

}

return TWML\_TYPE\_UNKNOWN;

}

std::vector<uint64\_t> TensorRecordReader::readShape() {

int32\_t length = readInt32();

std::vector<uint64\_t> shape;

shape.reserve(length);

for (int32\_t i = 0; i < length; i++) {

shape.push\_back(static\_cast<uint64\_t>(readInt64()));

}

return shape;

}

template<typename T>

RawTensor TensorRecordReader::readTypedTensor() {

std::vector<uint64\_t> shape;

int32\_t length = 0;

const uint8\_t \*data = nullptr;

uint64\_t raw\_length = 0;

uint8\_t field\_type = TTYPE\_STOP;

while ((field\_type = readByte()) != TTYPE\_STOP) {

int16\_t field\_id = readInt16();

switch (field\_id) {

case 1:

CHECK\_THRIFT\_TYPE(field\_type, TTYPE\_LIST, "data");

CHECK\_THRIFT\_TYPE(readByte(), TensorTraits<T>::ThriftType, "data\_type");

length = getRawBuffer<T>(&data);

raw\_length = length \* sizeof(T);

break;

case 2:

CHECK\_THRIFT\_TYPE(field\_type, TTYPE\_LIST, "shape");

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_I64, "shape\_type");

shape = readShape();

break;

default:

throw ThriftInvalidField(field\_id, "TensorRecordReader::readTypedTensor");

}

}

// data is required

if (data == nullptr) {

throw twml::Error(TWML\_ERR\_THRIFT, "data field not found for TypedTensor");

}

// shape is optional

if (shape.size() == 0) {

shape.push\_back((uint64\_t)length);

}

// TODO: Try avoiding stride calculation

std::vector<uint64\_t> strides = calcStrides(shape);

// FIXME: Try to use const void \* in Tensors.

return RawTensor(const\_cast<void \*>(static\_cast<const void \*>(data)),

shape, strides, (twml\_type)TensorTraits<T>::TwmlType, true, raw\_length);

}

RawTensor TensorRecordReader::readRawTypedTensor() {

std::vector<uint64\_t> shape;

const uint8\_t \*data = nullptr;

twml\_type type = TWML\_TYPE\_UNKNOWN;

uint64\_t raw\_length = 0;

uint8\_t field\_type = TTYPE\_STOP;

while ((field\_type = readByte()) != TTYPE\_STOP) {

int16\_t field\_id = readInt16();

switch (field\_id) {

case 1:

CHECK\_THRIFT\_TYPE(field\_type, TTYPE\_I32, "DataType");

type = getTwmlType(readInt32());

break;

case 2:

CHECK\_THRIFT\_TYPE(field\_type, TTYPE\_STRING, "content");

raw\_length = getRawBuffer<uint8\_t>(&data);

break;

case 3:

CHECK\_THRIFT\_TYPE(field\_type, TTYPE\_LIST, "shape");

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_I64, "shape\_type");

shape = readShape();

break;

default:

throw ThriftInvalidField(field\_id, "TensorRecordReader::readRawTypedTensor");

}

}

// data type is required

if (type == TWML\_TYPE\_UNKNOWN) {

throw twml::Error(TWML\_ERR\_THRIFT, "DataType is a required field for RawTypedTensor");

}

// data is required

if (data == nullptr) {

throw twml::Error(TWML\_ERR\_THRIFT, "content is a required field for RawTypedTensor");

}

// shape is optional in the thrift file, but it is really required for string types.

if (shape.size() == 0) {

if (type == TWML\_TYPE\_STRING) {

throw twml::Error(TWML\_ERR\_THRIFT, "shape required for string types in RawTypedTensor");

}

shape.push\_back((uint64\_t)(raw\_length / getSizeOf(type)));

}

// TODO: Try avoiding stride calculation

std::vector<uint64\_t> strides = calcStrides(shape);

// FIXME: Try to use const void \* data inside Tensors.

return RawTensor(const\_cast<void \*>(static\_cast<const void \*>(data)),

shape, strides, type, false, raw\_length);

}

RawTensor TensorRecordReader::readStringTensor() {

std::vector<uint64\_t> shape;

int32\_t length = 0;

const uint8\_t \*data = nullptr;

uint64\_t raw\_length = 0;

uint8\_t field\_type = TTYPE\_STOP;

const uint8\_t \*dummy = nullptr;

while ((field\_type = readByte()) != TTYPE\_STOP) {

int16\_t field\_id = readInt16();

switch (field\_id) {

case 1:

CHECK\_THRIFT\_TYPE(field\_type, TTYPE\_LIST, "data");

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_STRING, "data\_type");

length = readInt32();

// Store the current location of the byte stream.

// Use this at to "deocde strings" at a later point.

data = getBuffer();

for (int32\_t i = 0; i < length; i++) {

// Skip reading the strings

getRawBuffer<uint8\_t>(&dummy);

}

raw\_length = length;

break;

case 2:

CHECK\_THRIFT\_TYPE(field\_type, TTYPE\_LIST, "shape");

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_I64, "shape\_type");

shape = readShape();

break;

default:

throw ThriftInvalidField(field\_id, "TensorRecordReader::readTypedTensor");

}

}

// data is required

if (data == nullptr) {

throw twml::Error(TWML\_ERR\_THRIFT, "data field not found for TypedTensor");

}

// shape is optional

if (shape.size() == 0) {

shape.push\_back((uint64\_t)length);

}

// TODO: Try avoiding stride calculation

std::vector<uint64\_t> strides = calcStrides(shape);

// FIXME: Try to use const void \* in Tensors.

return RawTensor(const\_cast<void \*>(static\_cast<const void \*>(data)),

shape, strides, TWML\_TYPE\_UINT8, false, raw\_length);

}

RawTensor TensorRecordReader::readGeneralTensor() {

// No loop is required because GeneralTensor is union. It is going to contain one field only.

// All the fields are structs

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_STRUCT, "type");

int16\_t field\_id = readInt16();

RawTensor output;

switch (field\_id) {

case GT\_RAW:

output = readRawTypedTensor();

break;

case GT\_STRING:

output = readStringTensor();

break;

case GT\_INT32:

output = readTypedTensor<int32\_t>();

break;

case GT\_INT64:

output = readTypedTensor<int64\_t>();

break;

case GT\_FLOAT:

case GT\_DOUBLE:

// Store both FloatTensor and DoubleTensor as double tensor as both are list of doubles.

output = readTypedTensor<double>();

break;

case GT\_BOOL:

output = readTypedTensor<bool>();

break;

default:

throw ThriftInvalidField(field\_id, "TensorRecordReader::readGeneralTensor()");

}

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_STOP, "stop");

return output;

}

RawSparseTensor TensorRecordReader::readCOOSparseTensor() {

std::vector<uint64\_t> shape;

uint8\_t field\_type = TTYPE\_STOP;

RawTensor indices, values;

while ((field\_type = readByte()) != TTYPE\_STOP) {

int16\_t field\_id = readInt16();

switch (field\_id) {

case 1:

CHECK\_THRIFT\_TYPE(field\_type, TTYPE\_LIST, "shape");

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_I64, "shape\_type");

shape = readShape();

break;

case 2:

indices = readTypedTensor<int64\_t>();

break;

case 3:

values = readGeneralTensor();

break;

default:

throw twml::Error(TWML\_ERR\_THRIFT, "Invalid field when deocidng COOSparseTensor");

}

}

return RawSparseTensor(indices, values, shape);

}

void TensorRecordReader::readTensor(const int feature\_type, TensorRecord \*record) {

CHECK\_THRIFT\_TYPE(feature\_type, TTYPE\_MAP, "type");

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_I64, "key\_type");

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_STRUCT, "value\_type");

int32\_t length = readInt32();

for (int32\_t i = 0; i < length; i++) {

int64\_t id = readInt64();

record->m\_tensors.emplace(id, readGeneralTensor());

}

}

void TensorRecordReader::readSparseTensor(const int feature\_type, TensorRecord \*record) {

CHECK\_THRIFT\_TYPE(feature\_type, TTYPE\_MAP, "type");

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_I64, "key\_type");

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_STRUCT, "value\_type");

int32\_t length = readInt32();

for (int32\_t i = 0; i < length; i++) {

int64\_t id = readInt64();

// No loop is required because SparseTensor is union. It is going to contain one field only.

// All the fields are structs

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_STRUCT, "field");

int16\_t field\_id = readInt16();

RawSparseTensor output;

// Only COOSparsetensor is supported.

switch (field\_id) {

case SP\_COO:

output = readCOOSparseTensor();

break;

default:

throw ThriftInvalidField(field\_id, "TensorRecordReader::readSparseTensor()");

}

// Read the last byte of the struct.

CHECK\_THRIFT\_TYPE(readByte(), TTYPE\_STOP, "stop");

// Add to the map.

record->m\_sparse\_tensors.emplace(id, output);

}

}

} // namespace twml