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

#include "tensorflow/core/framework/shape\_inference.h"

#include "tensorflow/core/framework/op\_kernel.h"

#include <twml.h>

#include "tensorflow\_utils.h"

using namespace tensorflow;

REGISTER\_OP("PartitionSparseTensorMod")

.Attr("T: {float, double}")

.Input("indices: int64")

.Input("values: T")

.Output("result: output\_types")

.Attr("num\_partitions: int")

.Attr("output\_types: list({int64, float, double})")

.SetShapeFn([](::tensorflow::shape\_inference::InferenceContext\* c) {

return Status::OK();

}).Doc(R"doc(

A tensorflow OP that partitions an input batch represented as a sparse tensor

(indices are [ids, keys]) into separate sparse tensors to more optimally place

sparse computations in distributed training.

Inputs

indices: Indices from sparse tensor ([ids, keys] from the batch).

values: Batch values from the original features dict.

Attr

num\_partitions: Number of partitions to generate.

output\_types: A list of types for the output tensors like

[tf.int64, tf.float32, tf.int64, tf.float32, ...]

The length must be 2 \* num\_partitions (see Outputs below)

Outputs

List of dense tensors containing for each partition:

- partitioned indices tensor ([ids, keys] from partitioned batch)

- partitioned values tensor

The list lenth is 2 \* num\_partitions. Example:

[ [ids\_1, keys\_1], values\_1, [ids\_2, keys\_2], values\_2, ... ]

)doc");

template<typename T>

class PartitionSparseTensorMod : public OpKernel {

private:

int64 num\_partitions;

public:

explicit PartitionSparseTensorMod(OpKernelConstruction\* context) : OpKernel(context) {

OP\_REQUIRES\_OK(context, context->GetAttr("num\_partitions", &num\_partitions));

OP\_REQUIRES(context, num\_partitions > 0,

errors::InvalidArgument("Number of partitions must be positive"));

}

void Compute(OpKernelContext\* context) override {

// grab input tensors

const Tensor& indices\_tensor = context->input(0); // (ids, keys)

const Tensor& values\_tensor = context->input(1);

// check sizes

int64 num\_keys = indices\_tensor.shape().dim\_size(0);

OP\_REQUIRES(context, indices\_tensor.dims() == 2,

errors::InvalidArgument("Indices tensor must be 2D [ids, keys]"));

OP\_REQUIRES(context, indices\_tensor.shape().dim\_size(1) == 2,

errors::InvalidArgument("Indices tensor must have 2 cols [ids, keys]"));

OP\_REQUIRES(context, values\_tensor.shape().dim\_size(0) == num\_keys,

errors::InvalidArgument("Number of values must match number of keys"));

// grab input vectors

auto indices = indices\_tensor.flat<int64>();

auto values = values\_tensor.flat<T>();

// count the number of features that fall in each partition

std::vector<int64> partition\_counts(num\_partitions);

for (int i = 0; i < num\_keys; i++) {

int64 key = indices(2 \* i + 1);

int64 partition\_id = key % num\_partitions;

partition\_counts[partition\_id]++;

}

// allocate outputs for each partition and keep references

std::vector<int64\*> output\_indices\_partitions;

std::vector<T\*> output\_values\_partitions;

output\_indices\_partitions.reserve(num\_partitions);

output\_values\_partitions.reserve(num\_partitions);

for (int i = 0; i < num\_partitions; i++) {

Tensor \*output\_indices = nullptr, \*output\_values = nullptr;

TensorShape shape\_indices = TensorShape({partition\_counts[i], 2});

TensorShape shape\_values = TensorShape({partition\_counts[i]});

OP\_REQUIRES\_OK(context, context->allocate\_output(2 \* i, shape\_indices, &output\_indices));

OP\_REQUIRES\_OK(context, context->allocate\_output(2 \* i + 1, shape\_values, &output\_values));

output\_indices\_partitions.push\_back(output\_indices->flat<int64>().data());

output\_values\_partitions.push\_back(output\_values->flat<T>().data());

}

// assign a partition id to each feature

// populate tensors for each partition

std::vector<int64> partition\_indices(num\_partitions);

for (int i = 0; i < num\_keys; i++) {

int64 key = indices(2 \* i + 1);

int64 pid = key % num\_partitions; // partition id

int64 idx = partition\_indices[pid]++;

output\_indices\_partitions[pid][2 \* idx] = indices(2 \* i);

output\_indices\_partitions[pid][2 \* idx + 1] = key / num\_partitions;

output\_values\_partitions[pid][idx] = values(i);

}

}

};

#define REGISTER(Type) \

\

REGISTER\_KERNEL\_BUILDER( \

Name("PartitionSparseTensorMod") \

.Device(DEVICE\_CPU) \

.TypeConstraint<Type>("T"), \

PartitionSparseTensorMod<Type>); \

REGISTER(float);

REGISTER(double);