#ifndef TENSORFLOW\_CORE\_KERNELS\_BINARY\_SPARSE\_TENSOR\_DENSE\_MATMUL\_IMPL\_H\_

#define TENSORFLOW\_CORE\_KERNELS\_BINARY\_SPARSE\_TENSOR\_DENSE\_MATMUL\_IMPL\_H\_

#include <atomic>

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

#include "tensorflow/core/lib/core/blocking\_counter.h"

#include "tensorflow/core/lib/core/threadpool.h"

namespace tensorflow {

namespace functor {

// `ConservativeShard` is adopted rather than `Shard` in tensorflow because the

// original `Shard` may generate number of shards more than the number of

// threads, which is not ideal for this case, as it may cause too much overhead.

static void ConservativeShard(int max\_parallelism, thread::ThreadPool \*workers,

int64 total, int64 cost\_per\_unit,

std::function<void(int64, int64)> work) {

if (total == 0) {

return;

}

max\_parallelism = std::min(max\_parallelism, workers->NumThreads());

if (max\_parallelism <= 1) {

// Just inline the whole work since we only have 1 thread (core).

work(0, total);

return;

}

cost\_per\_unit = std::max(1LL, cost\_per\_unit);

// We shard [0, total) into "num\_shards" shards.

// 1 <= num\_shards <= num worker threads

//

// If total \* cost\_per\_unit is small, it is not worth shard too

// much. Let us assume each cost unit is 1ns, kMinCostPerShard=10000

// is 10us.

static const int64 kMinCostPerShard = 10000;

const int num\_shards =

std::max<int>(1, std::min(static\_cast<int64>(max\_parallelism),

total \* cost\_per\_unit / kMinCostPerShard));

// Each shard contains up to "block\_size" units. [0, total) is sharded

// into:

// [0, block\_size), [block\_size, 2\*block\_size), ...

// The 1st shard is done by the caller thread and the other shards

// are dispatched to the worker threads. The last shard may be smaller than

// block\_size.

const int64 block\_size = (total + num\_shards - 1) / num\_shards;

if (block\_size >= total) {

work(0, total);

return;

}

const int num\_shards\_used = (total + block\_size - 1) / block\_size;

BlockingCounter counter(num\_shards\_used - 1);

for (int64 start = block\_size; start < total; start += block\_size) {

auto limit = std::min(start + block\_size, total);

workers->Schedule([&work, &counter, start, limit]() {

work(start, limit); // Compute the shard.

counter.DecrementCount(); // The shard is done.

});

}

// Inline execute the 1st shard.

work(0, std::min(block\_size, total));

counter.Wait();

}

static inline void VectorSum(float \*a, const float \*b, int n) {

for (int i = 0; i < n; ++i) {

a[i] += b[i];

}

}

// This func is to vectorize the computation of segment sum.

template<typename Tindices>

static void LookupAndSegmentSum(const Tindices \*a\_indices, const float \*b,

int nnz, int outer\_right, float \*output) {

for (std::size\_t i = 0; i < nnz; ++i) {

const Tindices m = a\_indices[i \* 2];

const Tindices k = a\_indices[i \* 2 + 1];

auto output\_row\_m = output + m \* outer\_right;

auto b\_row\_k = b + k \* outer\_right;

VectorSum(output\_row\_m, b\_row\_k, outer\_right);

}

}

// This func enables sharding and multithreading, it comes with an overhead of

// duplicating output buffer to achieve lock free output. So there should not

// be too many threads.

template<typename Tindices>

static void ParallelLookupAndSegmentSum(OpKernelContext \*ctx,

const Tindices \*a\_indices,

const float \*b, int nnz, int outer\_left,

int outer\_right, float \*output) {

auto worker\_threads = \*(ctx->device()->tensorflow\_cpu\_worker\_threads());

int out\_size = outer\_left \* outer\_right;

if (worker\_threads.num\_threads <= 1) {

memset(output, 0, out\_size \* sizeof(float));

LookupAndSegmentSum<Tindices>(a\_indices, b,

nnz, outer\_right,

output);

return;

}

// this is to make buffer align with kAllocatorAlignment

int padded\_out\_size = (out\_size + (Allocator::kAllocatorAlignment - 1)) &

~(Allocator::kAllocatorAlignment - 1);

std::size\_t num\_bytes =

(worker\_threads.num\_threads - 1) \* padded\_out\_size \* sizeof(float);

auto buffer = std::unique\_ptr<float>(reinterpret\_cast<float \*>(

port::AlignedMalloc(num\_bytes, Allocator::kAllocatorAlignment)));

float \*temp\_out = buffer.get();

std::atomic<int> thread\_index(0);

auto task = [&](int64 start, int64 limit) {

int local\_thread\_index = thread\_index++;

float \*buf\_ptr = nullptr;

if (local\_thread\_index == 0) {

buf\_ptr = output;

} else {

buf\_ptr = temp\_out + (local\_thread\_index - 1) \* padded\_out\_size;

}

memset(buf\_ptr, 0, out\_size \* sizeof(float));

LookupAndSegmentSum<Tindices>(a\_indices + start \* 2, b,

limit - start, outer\_right,

buf\_ptr);

};

int cost\_per\_unit = outer\_right;

// We don't use tensorflow shard func as tf may create more shards than

// number of threads.

ConservativeShard(worker\_threads.num\_threads, worker\_threads.workers, nnz,

static\_cast<int64>(cost\_per\_unit), task);

for (int i = 1; i < thread\_index; ++i) {

VectorSum(output, temp\_out + (i - 1) \* padded\_out\_size, out\_size);

}

}

} // namespace functor

} // namespace tensorflow

#endif // TENSORFLOW\_CORE\_KERNELS\_BINARY\_SPARSE\_TENSOR\_DENSE\_MATMUL\_IMPL\_H\_