#include "internal/linear\_search.h"

#include "internal/error.h"

#include <twml/hashing\_discretizer\_impl.h>

#include <twml/optim.h>

#include <algorithm>

namespace twml {

template<typename Tx>

static int64\_t lower\_bound\_search(const Tx \*data, const Tx val, const int64\_t buf\_size) {

auto index\_temp = std::lower\_bound(data, data + buf\_size, val);

return static\_cast<int64\_t>(index\_temp - data);

}

template<typename Tx>

static int64\_t upper\_bound\_search(const Tx \*data, const Tx val, const int64\_t buf\_size) {

auto index\_temp = std::upper\_bound(data, data + buf\_size, val);

return static\_cast<int64\_t>(index\_temp - data);

}

template<typename Tx>

using search\_method = int64\_t (\*)(const Tx \*, const Tx, const int64\_t);

typedef uint64\_t (\*hash\_signature)(uint64\_t, int64\_t, uint64\_t);

// uint64\_t integer\_multiplicative\_hashing()

//

// A function to hash discretized feature\_ids into one of 2\*\*output\_bits buckets.

// This function hashes the feature\_ids to achieve a uniform distribution of

// IDs, so the hashed IDs are with high probability far apart

// Then, bucket\_indices can simply be added, resulting in unique new IDs with high probability

// We integer hash again to again spread out the new IDs

// Finally we take the upper

// Required args:

// feature\_id:

// The feature id of the feature to be hashed.

// bucket\_index:

// The bucket index of the discretized feature value

// output\_bits:

// The number of bits of output space for the features to be hashed into.

//

// Note - feature\_ids may have arbitrary distribution within int32s

// Note - 64 bit feature\_ids can be processed with this, but the upper

// 32 bits have no effect on the output

// e.g. all feature ids 0 through 255 exist in movie-lens.

// this hashing constant is good for 32 LSBs. will use N=32. (can use N<32 also)

// this hashing constant is co-prime with 2\*\*32, therefore we have that

// a != b, a and b in [0,2\*\*32)

// implies

// f(a) != f(b) where f(x) = (hashing\_constant \* x) % (2\*\*32)

// note that we are mostly ignoring the upper 32 bits, using modulo 2\*\*32 arithmetic

uint64\_t integer\_multiplicative\_hashing(uint64\_t feature\_id,

int64\_t bucket\_index,

uint64\_t output\_bits) {

// possibly use 14695981039346656037 for 64 bit unsigned??

// = 20921 \* 465383 \* 1509404459

// alternatively, 14695981039346656039 is prime

// We would also need to use N = 64

const uint64\_t hashing\_constant = 2654435761;

const uint64\_t N = 32;

// hash once to prevent problems from anomalous input id distributions

feature\_id \*= hashing\_constant;

feature\_id += bucket\_index;

// this hash enables the following right shift operation

// without losing the bucket information (lower bits)

feature\_id \*= hashing\_constant;

// output size is a power of 2

feature\_id >>= N - output\_bits;

uint64\_t mask = (1 << output\_bits) - 1;

return mask & feature\_id;

}

uint64\_t integer64\_multiplicative\_hashing(uint64\_t feature\_id,

int64\_t bucket\_index,

uint64\_t output\_bits) {

const uint64\_t hashing\_constant = 14695981039346656039UL;

const uint64\_t N = 64;

// hash once to prevent problems from anomalous input id distributions

feature\_id \*= hashing\_constant;

feature\_id += bucket\_index;

// this hash enables the following right shift operation

// without losing the bucket information (lower bits)

feature\_id \*= hashing\_constant;

// output size is a power of 2

feature\_id >>= N - output\_bits;

uint64\_t mask = (1 << output\_bits) - 1;

return mask & feature\_id;

}

int64\_t option\_bits(int64\_t options, int64\_t high, int64\_t low) {

options >>= low;

options &= (1 << (high - low + 1)) - 1;

return options;

}

// it is assumed that start\_compute and end\_compute are valid

template<typename T>

void hashDiscretizerInfer(Tensor &output\_keys,

Tensor &output\_vals,

const Tensor &input\_ids,

const Tensor &input\_vals,

const Tensor &bin\_vals,

int output\_bits,

const Map<int64\_t, int64\_t> &ID\_to\_index,

int64\_t start\_compute,

int64\_t end\_compute,

int64\_t n\_bin,

int64\_t options) {

auto output\_keys\_data = output\_keys.getData<int64\_t>();

auto output\_vals\_data = output\_vals.getData<T>();

auto input\_ids\_data = input\_ids.getData<int64\_t>();

auto input\_vals\_data = input\_vals.getData<T>();

auto bin\_vals\_data = bin\_vals.getData<T>();

// The function pointer implementation removes the option\_bits

// function call (might be inlined) and corresponding branch from

// the hot loop, but it prevents inlining these functions, so

// there will be function call overhead. Uncertain which would

// be faster, testing needed. Also, code optimizers do weird things...

hash\_signature hash\_fn = integer\_multiplicative\_hashing;

switch (option\_bits(options, 4, 2)) {

case 0:

hash\_fn = integer\_multiplicative\_hashing;

break;

case 1:

hash\_fn = integer64\_multiplicative\_hashing;

break;

default:

hash\_fn = integer\_multiplicative\_hashing;

}

search\_method<T> search\_fn = lower\_bound\_search;

switch (option\_bits(options, 1, 0)) {

case 0:

search\_fn = lower\_bound\_search<T>;

break;

case 1:

search\_fn = linear\_search<T>;

break;

case 2:

search\_fn = upper\_bound\_search<T>;

break;

default:

search\_fn = lower\_bound\_search<T>;

}

for (uint64\_t i = start\_compute; i < end\_compute; i++) {

int64\_t id = input\_ids\_data[i];

T val = input\_vals\_data[i];

auto iter = ID\_to\_index.find(id);

if (iter != ID\_to\_index.end()) {

int64\_t feature\_idx = iter->second;

const T \*bin\_vals\_start = bin\_vals\_data + feature\_idx \* n\_bin;

int64\_t out\_bin\_idx = search\_fn(bin\_vals\_start, val, n\_bin);

output\_keys\_data[i] = hash\_fn(id, out\_bin\_idx, output\_bits);

output\_vals\_data[i] = 1;

} else {

// feature not calibrated

output\_keys\_data[i] = id & ((1 << output\_bits) - 1);

output\_vals\_data[i] = val;

}

}

}

void hashDiscretizerInfer(Tensor &output\_keys,

Tensor &output\_vals,

const Tensor &input\_ids,

const Tensor &input\_vals,

int n\_bin,

const Tensor &bin\_vals,

int output\_bits,

const Map<int64\_t, int64\_t> &ID\_to\_index,

int start\_compute,

int end\_compute,

int64\_t options) {

if (input\_ids.getType() != TWML\_TYPE\_INT64) {

throw twml::Error(TWML\_ERR\_TYPE, "input\_ids must be a Long Tensor");

}

if (output\_keys.getType() != TWML\_TYPE\_INT64) {

throw twml::Error(TWML\_ERR\_TYPE, "output\_keys must be a Long Tensor");

}

if (input\_vals.getType() != bin\_vals.getType()) {

throw twml::Error(TWML\_ERR\_TYPE,

"Data type of input\_vals does not match type of bin\_vals");

}

if (bin\_vals.getNumDims() != 1) {

throw twml::Error(TWML\_ERR\_SIZE,

"bin\_vals must be 1 Dimensional");

}

uint64\_t size = input\_ids.getDim(0);

if (end\_compute == -1) {

end\_compute = size;

}

if (start\_compute < 0 || start\_compute >= size) {

throw twml::Error(TWML\_ERR\_SIZE,

"start\_compute out of range");

}

if (end\_compute < -1 || end\_compute > size) {

throw twml::Error(TWML\_ERR\_SIZE,

"end\_compute out of range");

}

if (start\_compute > end\_compute && end\_compute != -1) {

throw twml::Error(TWML\_ERR\_SIZE,

"must have start\_compute <= end\_compute, or end\_compute==-1");

}

if (output\_keys.getStride(0) != 1 || output\_vals.getStride(0) != 1 ||

input\_ids.getStride(0) != 1 || input\_vals.getStride(0) != 1 ||

bin\_vals.getStride(0) != 1) {

throw twml::Error(TWML\_ERR\_SIZE,

"All Strides must be 1.");

}

switch (input\_vals.getType()) {

case TWML\_TYPE\_FLOAT:

twml::hashDiscretizerInfer<float>(output\_keys, output\_vals,

input\_ids, input\_vals,

bin\_vals, output\_bits, ID\_to\_index,

start\_compute, end\_compute, n\_bin, options);

break;

case TWML\_TYPE\_DOUBLE:

twml::hashDiscretizerInfer<double>(output\_keys, output\_vals,

input\_ids, input\_vals,

bin\_vals, output\_bits, ID\_to\_index,

start\_compute, end\_compute, n\_bin, options);

break;

default:

throw twml::Error(TWML\_ERR\_TYPE,

"Unsupported datatype for hashDiscretizerInfer");

}

}

} // namespace twml