#include "internal/khash.h"

#include "internal/error.h"

#include <twml/defines.h>

#include <twml/Hashmap.h>

#include <cstdint>

namespace twml {

HashMap::HashMap() :

m\_hashmap(nullptr) {

TWML\_CHECK(twml\_hashmap\_create(&m\_hashmap), "Failed to create HashMap");

}

HashMap::~HashMap() {

// Do not throw exceptions from the destructor

twml\_hashmap\_delete(m\_hashmap);

}

void HashMap::clear() {

TWML\_CHECK(twml\_hashmap\_clear(m\_hashmap), "Failed to clear HashMap");

}

uint64\_t HashMap::size() const {

uint64\_t size;

TWML\_CHECK(twml\_hashmap\_get\_size(&size, m\_hashmap), "Failed to get HashMap size");

return size;

}

int8\_t HashMap::insert(const HashKey\_t key) {

int8\_t result;

TWML\_CHECK(twml\_hashmap\_insert\_key(&result, m\_hashmap, key),

"Failed to insert key");

return result;

}

int8\_t HashMap::insert(const HashKey\_t key, const HashKey\_t val) {

int8\_t result;

TWML\_CHECK(twml\_hashmap\_insert\_key\_and\_value(&result, m\_hashmap, key, val),

"Failed to insert key");

return result;

}

int8\_t HashMap::get(HashVal\_t &val, const HashKey\_t key) const {

int8\_t result;

TWML\_CHECK(twml\_hashmap\_get\_value(&result, &val, m\_hashmap, key),

"Failed to insert key,value pair");

return result;

}

void HashMap::insert(Tensor &mask, const Tensor keys) {

TWML\_CHECK(twml\_hashmap\_insert\_keys(mask.getHandle(), m\_hashmap, keys.getHandle()),

"Failed to insert keys tensor");

}

void HashMap::insert(Tensor &mask, const Tensor keys, const Tensor vals) {

TWML\_CHECK(twml\_hashmap\_insert\_keys\_and\_values(mask.getHandle(), m\_hashmap,

keys.getHandle(), vals.getHandle()),

"Failed to insert keys,values tensor pair");

}

void HashMap::remove(const Tensor keys) {

TWML\_CHECK(twml\_hashmap\_remove\_keys(m\_hashmap, keys.getHandle()),

"Failed to remove keys tensor");

}

void HashMap::get(Tensor &mask, Tensor &vals, const Tensor keys) const {

TWML\_CHECK(twml\_hashmap\_get\_values(mask.getHandle(), vals.getHandle(),

m\_hashmap, keys.getHandle()),

"Failed to get values tensor");

}

void HashMap::getInplace(Tensor &mask, Tensor &keys\_vals) const {

TWML\_CHECK(twml\_hashmap\_get\_values\_inplace(mask.getHandle(),

keys\_vals.getHandle(),

m\_hashmap),

"Failed to get values tensor");

}

void HashMap::toTensors(Tensor &keys, Tensor &vals) const {

TWML\_CHECK(twml\_hashmap\_to\_tensors(keys.getHandle(),

vals.getHandle(),

m\_hashmap),

"Failed to get keys,values tensors from HashMap");

}

} // namespace twml

using twml::HashKey\_t;

using twml::HashVal\_t;

KHASH\_MAP\_INIT\_INT64(HashKey\_t, HashVal\_t);

typedef khash\_t(HashKey\_t)\* hash\_map\_t;

twml\_err twml\_hashmap\_create(twml\_hashmap \*hashmap) {

hash\_map\_t \*h = reinterpret\_cast<hash\_map\_t \*>(hashmap);

\*h = kh\_init(HashKey\_t);

return TWML\_ERR\_NONE;

}

twml\_err twml\_hashmap\_clear(const twml\_hashmap hashmap) {

hash\_map\_t h = (hash\_map\_t)hashmap;

kh\_clear(HashKey\_t, h);

return TWML\_ERR\_NONE;

}

twml\_err twml\_hashmap\_get\_size(uint64\_t \*size, const twml\_hashmap hashmap) {

hash\_map\_t h = (hash\_map\_t)hashmap;

\*size = kh\_size(h);

return TWML\_ERR\_NONE;

}

twml\_err twml\_hashmap\_delete(const twml\_hashmap hashmap) {

hash\_map\_t h = (hash\_map\_t)hashmap;

kh\_destroy(HashKey\_t, h);

return TWML\_ERR\_NONE;

}

// insert, remove, get single key / value

twml\_err twml\_hashmap\_insert\_key(int8\_t \*mask,

const twml\_hashmap hashmap,

const HashKey\_t key) {

hash\_map\_t h = (hash\_map\_t)hashmap;

int ret = 0;

khiter\_t k = kh\_put(HashKey\_t, h, key, &ret);

\*mask = ret >= 0;

if (\*mask) {

HashVal\_t v = kh\_size(h);

kh\_value(h, k) = v;

}

return TWML\_ERR\_NONE;

}

twml\_err twml\_hashmap\_insert\_key\_and\_value(int8\_t \*mask, twml\_hashmap hashmap,

const HashKey\_t key, const HashVal\_t val) {

hash\_map\_t h = (hash\_map\_t)hashmap;

int ret = 0;

khiter\_t k = kh\_put(HashKey\_t, h, key, &ret);

\*mask = ret >= 0;

if (\*mask) {

kh\_value(h, k) = val;

}

return TWML\_ERR\_NONE;

}

twml\_err twml\_hashmap\_remove\_key(const twml\_hashmap hashmap,

const HashKey\_t key) {

hash\_map\_t h = (hash\_map\_t)hashmap;

khiter\_t k = kh\_get(HashKey\_t, h, key);

if (k != kh\_end(h)) {

kh\_del(HashKey\_t, h, k);

}

return TWML\_ERR\_NONE;

}

twml\_err twml\_hashmap\_get\_value(int8\_t \*mask, HashVal\_t \*val,

const twml\_hashmap hashmap, const HashKey\_t key) {

hash\_map\_t h = (hash\_map\_t)hashmap;

khiter\_t k = kh\_get(HashKey\_t, h, key);

if (k == kh\_end(h)) {

\*mask = false;

} else {

\*val = kh\_value(h, k);

\*mask = true;

}

return TWML\_ERR\_NONE;

}

// insert, get, remove tensors of keys / values

twml\_err twml\_hashmap\_insert\_keys(twml\_tensor masks,

const twml\_hashmap hashmap,

const twml\_tensor keys) {

auto masks\_tensor = twml::getTensor(masks);

auto keys\_tensor = twml::getConstTensor(keys);

if (masks\_tensor->getType() != TWML\_TYPE\_INT8) {

return TWML\_ERR\_TYPE;

}

if (keys\_tensor->getType() != TWML\_TYPE\_INT64) {

return TWML\_ERR\_TYPE;

}

if (keys\_tensor->getNumElements() != masks\_tensor->getNumElements()) {

return TWML\_ERR\_SIZE;

}

int8\_t \*mptr = masks\_tensor->getData<int8\_t>();

const HashKey\_t \*kptr = keys\_tensor->getData<HashKey\_t>();

uint64\_t num\_elements = keys\_tensor->getNumElements();

hash\_map\_t h = (hash\_map\_t)hashmap;

for (uint64\_t i = 0; i < num\_elements; i++) {

int ret = 0;

khiter\_t k = kh\_put(HashKey\_t, h, kptr[i], &ret);

mptr[i] = ret >= 0;

if (mptr[i]) {

HashVal\_t v = kh\_size(h);

kh\_value(h, k) = v;

}

}

return TWML\_ERR\_NONE;

}

twml\_err twml\_hashmap\_insert\_keys\_and\_values(twml\_tensor masks,

twml\_hashmap hashmap,

const twml\_tensor keys,

const twml\_tensor vals) {

auto masks\_tensor = twml::getTensor(masks);

auto keys\_tensor = twml::getConstTensor(keys);

auto vals\_tensor = twml::getConstTensor(vals);

if (masks\_tensor->getType() != TWML\_TYPE\_INT8) {

return TWML\_ERR\_TYPE;

}

if (keys\_tensor->getType() != TWML\_TYPE\_INT64) {

return TWML\_ERR\_TYPE;

}

if (vals\_tensor->getType() != TWML\_TYPE\_INT64) {

return TWML\_ERR\_TYPE;

}

if (keys\_tensor->getNumElements() != vals\_tensor->getNumElements() ||

keys\_tensor->getNumElements() != masks\_tensor->getNumElements()) {

return TWML\_ERR\_SIZE;

}

int8\_t \*mptr = masks\_tensor->getData<int8\_t>();

const HashKey\_t \*kptr = keys\_tensor->getData<HashKey\_t>();

const HashVal\_t \*vptr = twml::getConstTensor(vals)->getData<HashVal\_t>();

uint64\_t num\_elements = keys\_tensor->getNumElements();

hash\_map\_t h = (hash\_map\_t)hashmap;

for (uint64\_t i = 0; i < num\_elements; i++) {

int ret = 0;

khiter\_t k = kh\_put(HashKey\_t, h, kptr[i], &ret);

mptr[i] = ret >= 0;

if (mptr[i]) {

kh\_value(h, k) = vptr[i];

}

}

return TWML\_ERR\_NONE;

}

twml\_err twml\_hashmap\_remove\_keys(const twml\_hashmap hashmap,

const twml\_tensor keys) {

auto keys\_tensor = twml::getConstTensor(keys);

if (keys\_tensor->getType() != TWML\_TYPE\_INT64) {

return TWML\_ERR\_TYPE;

}

const HashKey\_t \*kptr = keys\_tensor->getData<HashKey\_t>();

uint64\_t num\_elements = keys\_tensor->getNumElements();

hash\_map\_t h = (hash\_map\_t)hashmap;

for (uint64\_t i = 0; i < num\_elements; i++) {

khiter\_t k = kh\_get(HashKey\_t, h, kptr[i]);

if (k != kh\_end(h)) {

kh\_del(HashKey\_t, h, kptr[i]);

}

}

return TWML\_ERR\_NONE;

}

twml\_err twml\_hashmap\_get\_values(twml\_tensor masks, twml\_tensor vals,

const twml\_hashmap hashmap, const twml\_tensor keys) {

auto masks\_tensor = twml::getTensor(masks);

auto vals\_tensor = twml::getTensor(vals);

auto keys\_tensor = twml::getConstTensor(keys);

if (masks\_tensor->getType() != TWML\_TYPE\_INT8) {

return TWML\_ERR\_TYPE;

}

if (keys\_tensor->getType() != TWML\_TYPE\_INT64) {

return TWML\_ERR\_TYPE;

}

if (vals\_tensor->getType() != TWML\_TYPE\_INT64) {

return TWML\_ERR\_TYPE;

}

if (keys\_tensor->getNumElements() != vals\_tensor->getNumElements() ||

keys\_tensor->getNumElements() != masks\_tensor->getNumElements()) {

return TWML\_ERR\_SIZE;

}

int8\_t \*mptr = masks\_tensor->getData<int8\_t>();

HashVal\_t \*vptr = vals\_tensor->getData<HashVal\_t>();

const HashKey\_t \*kptr = keys\_tensor->getData<HashKey\_t>();

uint64\_t num\_elements = keys\_tensor->getNumElements();

hash\_map\_t h = (hash\_map\_t)hashmap;

for (uint64\_t i = 0; i < num\_elements; i++) {

khiter\_t k = kh\_get(HashKey\_t, h, kptr[i]);

if (k == kh\_end(h)) {

mptr[i] = false;

} else {

mptr[i] = true;

vptr[i] = kh\_value(h, k);

}

}

return TWML\_ERR\_NONE;

}

twml\_err twml\_hashmap\_get\_values\_inplace(twml\_tensor masks, twml\_tensor keys\_vals,

const twml\_hashmap hashmap) {

auto masks\_tensor = twml::getTensor(masks);

auto keys\_tensor = twml::getTensor(keys\_vals);

if (masks\_tensor->getType() != TWML\_TYPE\_INT8) {

return TWML\_ERR\_TYPE;

}

if (keys\_tensor->getType() != TWML\_TYPE\_INT64) {

return TWML\_ERR\_TYPE;

}

if (keys\_tensor->getNumElements() != masks\_tensor->getNumElements()) {

return TWML\_ERR\_SIZE;

}

int8\_t \*mptr = masks\_tensor->getData<int8\_t>();

HashKey\_t \*kptr = keys\_tensor->getData<HashKey\_t>();

uint64\_t num\_elements = keys\_tensor->getNumElements();

hash\_map\_t h = (hash\_map\_t)hashmap;

for (uint64\_t i = 0; i < num\_elements; i++) {

khiter\_t k = kh\_get(HashKey\_t, h, kptr[i]);

if (k == kh\_end(h)) {

mptr[i] = false;

} else {

mptr[i] = true;

kptr[i] = kh\_value(h, k);

}

}

return TWML\_ERR\_NONE;

}

twml\_err twml\_hashmap\_to\_tensors(twml\_tensor keys, twml\_tensor vals,

const twml\_hashmap hashmap) {

hash\_map\_t h = (hash\_map\_t)hashmap;

const uint64\_t size = kh\_size(h);

auto keys\_tensor = twml::getTensor(keys);

auto vals\_tensor = twml::getTensor(vals);

if (keys\_tensor->getType() != TWML\_TYPE\_INT64) {

return TWML\_ERR\_TYPE;

}

if (vals\_tensor->getType() != TWML\_TYPE\_INT64) {

return TWML\_ERR\_TYPE;

}

if (size != keys\_tensor->getNumElements() ||

size != vals\_tensor->getNumElements()) {

return TWML\_ERR\_SIZE;

}

HashKey\_t \*kptr = keys\_tensor->getData<HashKey\_t>();

HashVal\_t \*vptr = vals\_tensor->getData<HashVal\_t>();

HashKey\_t key, i = 0;

HashKey\_t val;

kh\_foreach(h, key, val, {

kptr[i] = key;

vptr[i] = val;

i++;

});

return TWML\_ERR\_NONE;

}