#include <CL/sycl.hpp>

#include <unordered\_map>

#include <chrono>

#include <iostream>

namespace sycl = cl::sycl;

class PointCloudHash {

public:

PointCloudHash(size\_t size, sycl::queue& q) : size(size), hash\_table(sycl::malloc\_shared<std::multi\_ptr<float>>(size, q), sycl::free, q) {}

void insert(const std::vector<float>& point, sycl::queue& q) {

uint32\_t hash = compute\_hash(point);

auto& bucket = hash\_table[hash];

if (!bucket) {

bucket = sycl::malloc\_shared<float>(point.size(), q);

sycl::memcpy(bucket.get(), point.data(), point.size() \* sizeof(float)).wait();

} else {

auto new\_bucket = sycl::malloc\_shared<float>(bucket\_size(bucket) + point.size(), q);

sycl::memcpy(new\_bucket.get(), bucket.get(), bucket\_size(bucket) \* sizeof(float)).wait();

sycl::memcpy(new\_bucket.get() + bucket\_size(bucket), point.data(), point.size() \* sizeof(float)).wait();

sycl::free(bucket.get(), q);

bucket = new\_bucket;

}

}

std::vector<std::vector<float>> query(const std::vector<float>& point, sycl::queue& q) {

uint32\_t hash = compute\_hash(point);

auto& bucket = hash\_table[hash];

if (!bucket) {

return {};

} else {

std::vector<float> bucket\_data(bucket\_size(bucket));

sycl::memcpy(bucket\_data.data(), bucket.get(), bucket\_size(bucket) \* sizeof(float)).wait();

std::vector<std::vector<float>> result;

for (size\_t i = 0; i < bucket\_size(bucket); i += point.size()) {

result.push\_back(std::vector<float>(bucket\_data.begin() + i, bucket\_data.begin() + i + point.size()));

}

return result;

}

}

private:

uint32\_t compute\_hash(const std::vector<float>& point) {

// TODO: compute hash

}

size\_t bucket\_size(sycl::multi\_ptr<float> bucket) {

size\_t size = 0;

while (bucket[size]) {

size += point\_size;

}

return size;

}

const size\_t size;

const size\_t point\_size = 3;

sycl::usm::shared\_ptr<sycl::multi\_ptr<float>> hash\_table;

};

void build\_hash\_table(std::vector<std::vector<float>>& points, sycl::queue& q) {

size\_t size = points.size() \* 2;

PointCloudHash hash\_table(size, q);

// Insert points into the hash table

for (size\_t i = 0; i < points.size(); i++) {

hash\_table.insert(points[i], q);

}

q.wait();

}

void query\_hash\_table(std::vector<std::vector<float>>& points, PointCloudHash& hash\_table, sycl::queue& q) {

// Query hash table for each point

for (size\_t i = 0; i < points.size(); i++) {

auto result = hash\_table.query(points[i], q);

// TODO: process result

}

q.wait();

}

int main() {

std::vector<std::vector<float>> points(1000);

// Initialize points

for (size\_t i = 0; i < 1000; i++) {

points[i] = {static\_cast<float>(i), static\_cast<float>(i + 1), static\_cast<float>(i + 2)};

}

sycl::queue q(sycl::gpu\_selector{});

auto start = std::chrono::high\_resolution\_clock::now();

build\_hash\_table(points, q);

auto end = std::chrono::high\_resolution\_clock::now();

auto build\_time = std::chrono::duration\_cast<std::chrono::microseconds>(end - start);

std::cout << "Build hash table time: " << build\_time.count() << " us" << std::endl;

PointCloudHash hash\_table(points.size() \* 2, q);

start = std::chrono::high\_resolution\_clock::now();

build\_hash\_table(points, q);

end = std::chrono::high\_resolution\_clock::now();

build\_time = std::chrono::duration\_cast<std::chrono::microseconds>(end - start);

std::cout << "Build hash table time: " << build\_time.count() << " us" << std::endl;

start = std::chrono::high\_resolution\_clock::now();

query\_hash\_table(points, hash\_table, q);

end = std::chrono::high\_resolution\_clock::now();

auto query\_time = std::chrono::duration\_cast<std::chrono::microseconds>(end - start);

std::cout << "Query hash table time: " << query\_time.count() << " us" << std::endl;

return 0;

}