#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <CL/cl.h>

typedef struct {

float x;

float y;

float z;

} Point;

typedef struct {

Point pos;

int idx;

} HashEntry;

#define MAX\_HASH\_ENTRY 1000

typedef struct {

HashEntry entries[MAX\_HASH\_ENTRY];

int count;

} HashTable;

void hash\_init(HashTable\* table) {

table->count = 0;

for (int i = 0; i < MAX\_HASH\_ENTRY; i++) {

table->entries[i].idx = -1;

}

}

void hash\_insert(HashTable\* table, Point pos, int idx) {

int hash = ((int)pos.x + (int)pos.y + (int)pos.z) % MAX\_HASH\_ENTRY;

while (table->entries[hash].idx != -1) {

hash = (hash + 1) % MAX\_HASH\_ENTRY;

}

table->entries[hash].pos = pos;

table->entries[hash].idx = idx;

table->count++;

}

int hash\_lookup(HashTable\* table, Point pos) {

int hash = ((int)pos.x + (int)pos.y + (int)pos.z) % MAX\_HASH\_ENTRY;

while (table->entries[hash].idx != -1) {

if (table->entries[hash].pos.x == pos.x &&

table->entries[hash].pos.y == pos.y &&

table->entries[hash].pos.z == pos.z) {

return table->entries[hash].idx;

}

hash = (hash + 1) % MAX\_HASH\_ENTRY;

}

return -1;

}

const char\* kernel\_source =

"typedef struct {\n"

" float x;\n"

" float y;\n"

" float z;\n"

"} Point;\n"

"typedef struct {\n"

" Point pos;\n"

" int idx;\n"

"} HashEntry;\n"

"#define MAX\_HASH\_ENTRY 1000\n"

"typedef struct {\n"

" HashEntry entries[MAX\_HASH\_ENTRY];\n"

" int count;\n"

"} HashTable;\n"

"void hash\_init(\_\_global HashTable\* table) {\n"

" table->count = 0;\n"

" for (int i = 0; i < MAX\_HASH\_ENTRY; i++) {\n"

" table->entries[i].idx = -1;\n"

" }\n"

"}\n"

"void hash\_insert(\_\_global HashTable\* table, Point pos, int idx) {\n"

" int hash = ((int)pos.x + (int)pos.y + (int)pos.z) % MAX\_HASH\_ENTRY;\n"

" while (table->entries[hash].idx != -1) {\n"

" hash = (hash + 1) % MAX\_HASH\_ENTRY;\n"

" }\n"

" table->entries[hash].pos = pos;\n"

" table->entries[hash].idx = idx;\n"

" table->count++;\n"

"}\n"

"int hash\_lookup(\_\_global HashTable\* table, Point pos) {\n"

" int hash = ((int)pos.x + (int)pos.y + (int)pos.z) % MAX\_HASH\_ENTRY;\n"

" while (table->entries[hash].idx != -1) {\n"

" if (table->entries[hash].pos.x == pos.x &&\n"

" table->entries[hash].pos.y == pos.y &&\n"

" table->entries[hash].pos.z == pos.z) {\n"

" return table->entries[hash].idx;\n"

" }\n"

" hash = (hash + 1) % MAX\_HASH\_ENTRY;\n"

" }\n"

" return -1;\n"

"}\n"

"\_\_kernel void hash\_build(\_\_global Point\* cloud, \_\_global HashTable\* table) {\n"

" int gid = get\_global\_id(0);\n"

" Point pos = cloud[gid];\n"

" hash\_insert(table, pos, gid);\n"

"}\n"

"\_\_kernel void hash\_lookup(\_\_global Point\* cloud, \_\_global HashTable\* table, \_\_global int\* result) {\n"

" int gid = get\_global\_id(0);\n"

" Point pos = cloud[gid];\n"

" result[gid] = hash\_lookup(table, pos);\n"

"}\n";

int main() {

cl\_int err;

cl\_platform\_id platform;

cl\_device\_id device;

cl\_context context;

cl\_command\_queue queue;

cl\_program program;

cl\_kernel kernel;

// Find the first available platform

err = clGetPlatformIDs(1, &platform, NULL);

if (err != CL\_SUCCESS) {

printf("Error: Failed to find a platform!\n");

return EXIT\_FAILURE;

}

// Find the first available GPU device

err = clGetDeviceIDs(platform, CL\_DEVICE\_TYPE\_GPU, 1, &device, NULL);

if (err != CL\_SUCCESS) {

printf("Error: Failed to find a GPU device!\n");

return EXIT\_FAILURE;

}

// Create a context for the device

context = clCreateContext(NULL, 1, &device, NULL, NULL, &err);

if (err != CL\_SUCCESS) {

printf("Error: Failed to create a context for the device!\n");

return EXIT\_FAILURE;

}

// Create a command queue for the device

queue = clCreateCommandQueue(context, device, 0, &err);

if (err != CL\_SUCCESS) {

printf("Error: Failed to create a command queue for the device!\n");

return EXIT\_FAILURE;

}

// Create a program from kernel source

program = clCreateProgramWithSource(context, 1, &kernel\_source, NULL, &err);

if (err != CL\_SUCCESS) {

printf("Error: Failed to create a program from kernel source!\n");

return EXIT\_FAILURE;

}

// Build the program

err = clBuildProgram(program, 1, &device, NULL, NULL, NULL);

if (err != CL\_SUCCESS) {

printf("Error: Failed to build the program!\n");

return EXIT\_FAILURE;

}

// Generate random point cloud data

const int num\_points = 1000;

Point\* cloud = (Point\*)malloc(num\_points \* sizeof(Point));

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

cloud[i].x = rand() % 100;

cloud[i].y = rand() % 100;

cloud[i].z = rand() % 100;

}

// Create a hash table on the host

HashTable table;

hash\_init(&table);

// Insert points into the hash table on the host

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

hash\_insert(&table, cloud[i], i);

}

// Create buffers for the input and output data on the device

cl\_mem cloud\_buf = clCreateBuffer(context, CL\_MEM\_READ\_ONLY, num\_points \* sizeof(Point), NULL, &err);

if (err != CL\_SUCCESS) {

printf("Error: Failed to create a buffer for cloud data!\n");

return EXIT\_FAILURE;

}

cl\_mem table\_buf = clCreateBuffer(context, CL\_MEM\_READ\_WRITE, sizeof(HashTable), NULL, &err);

if (err != CL\_SUCCESS) {

printf("Error: Failed to create a buffer for the hash table!\n");

return EXIT\_FAILURE;

}

cl\_mem result\_buf = clCreateBuffer(context, CL\_MEM\_WRITE\_ONLY, num\_points \* sizeof(int), NULL, &err);

if (err != CL\_SUCCESS) {

printf("Error: Failed to create a buffer for the result data!\n");

return EXIT\_FAILURE;

}

// Copy the input data to the device

err = clEnqueueWriteBuffer(queue, cloud\_buf, CL\_TRUE, 0, num\_points \* sizeof(Point), cloud, 0, NULL, NULL);

if (err != CL\_SUCCESS) {

printf("Error: Failed to copy cloud data to the device!\n");

return EXIT\_FAILURE;

}

// Copy the hash table to the device

err = clEnqueueWriteBuffer(queue, table\_buf, CL\_TRUE, 0, sizeof(HashTable), &table, 0, NULL, NULL);

if (err != CL\_SUCCESS) {

printf("Error: Failed to copy the hash table to the device!\n");

return EXIT\_FAILURE;

}

// Create the kernel

kernel = clCreateKernel(program, "hash\_lookup", &err);

if (err != CL\_SUCCESS) {

printf("Error: Failed to create the kernel!\n");

return EXIT\_FAILURE;

}

// Set the kernel arguments

err = clSetKernelArg(kernel, 0, sizeof(cl\_mem), &cloud\_buf);

if (err != CL\_SUCCESS) {

printf("Error: Failed to set kernel argument 0!\n");

return EXIT\_FAILURE;

}

err = clSetKernelArg(kernel, 1, sizeof(cl\_mem), &table\_buf);

if (err != CL\_SUCCESS) {

printf("Error: Failed to set kernel argument 1!\n");

return EXIT\_FAILURE;

}

err = clSetKernelArg(kernel, 2, sizeof(cl\_mem), &result\_buf);

if (err != CL\_SUCCESS) {

printf("Error: Failed to set kernel argument 2!\n");

return EXIT\_FAILURE;

}

// Execute the kernel

size\_t global\_size = num\_points;

err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global\_size, NULL, 0, NULL, NULL);

if (err != CL\_SUCCESS) {

printf("Error: Failed to execute the kernel!\n");

return EXIT\_FAILURE;

}

// Read the output data from the device

int\* results = (int\*)malloc(num\_points \* sizeof(int));

err = clEnqueueReadBuffer(queue, result\_buf, CL\_TRUE, 0, num\_points \* sizeof(int), results, 0, NULL, NULL);

if (err != CL\_SUCCESS) {

printf("Error: Failed to read result data from the device!\n");

return EXIT\_FAILURE;

}

// Verify the results

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

int idx = hash\_lookup(&table, cloud[i]);

if (results[i] != idx) {

printf("Error: mismatch at index %d (%d != %d)!\n", i, results[i], idx);

return EXIT\_FAILURE;

}

}

printf("Success: results match host implementation!\n");

// Release resources

clReleaseMemObject(cloud\_buf);

clReleaseMemObject(table\_buf);

clReleaseMemObject(result\_buf);

clReleaseKernel(kernel);

clReleaseProgram(program);

clReleaseCommandQueue(queue);

clReleaseContext(context);

free(cloud);

free(results);

return EXIT\_SUCCESS;

}