#include <iostream>

#include <ctime>

#include <cuda\_runtime.h>

using namespace std;

// Helper function to allocate memory on the device

void allocateDeviceMemory(float\*\* device\_ptr, size\_t size) {

cudaMalloc((void\*\*)device\_ptr, size);

}

// Helper function to copy memory from host to device

void copyToDevice(float\* device\_ptr, const float\* host\_ptr, size\_t size) {

cudaMemcpy(device\_ptr, host\_ptr, size, cudaMemcpyHostToDevice);

}

// Helper function to copy memory from device to host

void copyFromDevice(float\* host\_ptr, const float\* device\_ptr, size\_t size) {

cudaMemcpy(host\_ptr, device\_ptr, size, cudaMemcpyDeviceToHost);

}

// Helper function to free memory on the device

void freeDeviceMemory(float\* device\_ptr) {

cudaFree(device\_ptr);

}

// CPU implementation of matrix multiplication

void matrixMultCPU(const float\* a, const float\* b, float\* c, int N) {

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

for (int j = 0; j < N; j++) {

float sum = 0;

for (int k = 0; k < N; k++) {

sum += a[i \* N + k] \* b[k \* N + j];

}

c[i \* N + j] = sum;

}

}

}

// GPU implementation of matrix multiplication

\_\_global\_\_ void matrixMultGPU(const float\* a, const float\* b, float\* c, int N) {

// correct below line

int i = blockIdx.x \* blockDim.x + threadIdx.x;

int j = blockIdx.y \* blockDim.y + threadIdx.y;

if (i < N && j < N) {

float sum = 0;

for (int k = 0; k < N; k++) {

sum += a[i \* N + k] \* b[k \* N + j];

}

c[i \* N + j] = sum;

}

}

int main() {

int N = 1024; // matrix size

size\_t size = N \* N \* sizeof(float);

// allocate memory on host

float\* h\_a = new float[N \* N];

float\* h\_b = new float[N \* N];

float\* h\_c\_cpu = new float[N \* N];

float\* h\_c\_gpu = new float[N \* N];

// initialize matrices with random data

for (int i = 0; i < N \* N; i++) {

h\_a[i] = static\_cast <float> (rand()) / static\_cast <float> (RAND\_MAX);

h\_b[i] = static\_cast <float> (rand()) / static\_cast <float> (RAND\_MAX);

}

// allocate memory on device

float\* d\_a;

float\* d\_b;

float\* d\_c;

allocateDeviceMemory(&d\_a, size);

allocateDeviceMemory(&d\_b, size);

allocateDeviceMemory(&d\_c, size);

// copy data to device

copyToDevice(d\_a, h\_a, size);

copyToDevice(d\_b, h\_b, size);

// CPU matrix multiplication

clock\_t start\_cpu = clock();

matrixMultCPU(h\_a, h\_b, h\_c\_cpu, N);

clock\_t end\_cpu = clock();

double cpu\_time = static\_cast<double>(end\_cpu - start\_cpu) / CLOCKS\_PER\_SEC;

// GPU matrix multiplication

dim3 threadsPerBlock(16, 16); // 256 threads per block

dim3 numBlocks((N + threadsPerBlock.x - 1) / threadsPerBlock.x, (N + threadsPerBlock.y - 1) / threadsPerBlock.y);

clock\_t start\_gpu = clock();

matrixMultGPU<<<numBlocks, threadsPerBlock >> > (d\_a, d\_b, d\_c, N);

cudaDeviceSynchronize();

clock\_t end\_gpu = clock();

double gpu\_time = static\_cast<double>(end\_gpu - start\_gpu) / CLOCKS\_PER\_SEC;

// copy data back to host

copyFromDevice(h\_c\_gpu, d\_c, size);

// print results

cout << "CPU time: " << cpu\_time << " s" << endl;

cout << "GPU time: " << gpu\_time << " s" << endl;

cout << "Speedup: " << cpu\_time / gpu\_time << "x" << endl;

// verify results

for (int i = 0; i < N \* N; i++) {

if (h\_c\_cpu[i] != h\_c\_gpu[i]) {

cout << "Error: matrices do not match!" << endl;

break;

}

}

// free memory

// free memory

delete[] h\_a;

delete[] h\_b;

delete[] h\_c\_cpu;

delete[] h\_c\_gpu;

freeDeviceMemory(d\_a);

freeDeviceMemory(d\_b);

freeDeviceMemory(d\_c);

return 0;

}