Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Лабораторная работа № 4

«Метод Монте-Карло (ММК) на GPU»

Вариант № 5

Выполнили: Проверила:

студенты гр. 050503 ассист. каф. ЭВМ

Кириллов В. И. Русакович А.В.

Русель А. А.

Минск 2023

**1. Исходные данные**

Для точек A(0.6, 0.7, −0.1), B(0.1, −1.3, −0.27), C(−2.1, 0.4, −0.5), D(0.3, 0.1, 0.4) посчитать объем фигуры (рис. 1) ограниченной плоскостями △ABC, △ABD, △BCD, △CAD. Использовать метод агрегации на уровне блоков - parallel reduce, на уровне сетки – atomic add. Для GPU должна быть использована разделяемая память;

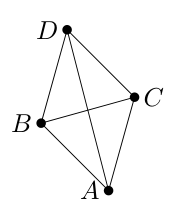


Рисунок 1 – Фигура

**2. Ход работы**

#include <iostream>

#include <algorithm>

#include <chrono>

#include <curand\_kernel.h>

using namespace std;

using namespace chrono;

/\*

\* Var 5

\*/

void PrintGPUsInfo() {

int devices;

cudaGetDeviceCount(&devices);

if (devices == 0) {

std::cout << "You have no GPUs!\n";

return;

}

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

cudaDeviceProp cudaDeviceProp{};

cudaGetDeviceProperties(&cudaDeviceProp, i);

std::cout << "Device '" << cudaDeviceProp.name << "':\n\n";

std::cout << "max threads per block: "

<< cudaDeviceProp.maxThreadsPerBlock << "\n";

std::cout << "totalGlobalMem: "

<< cudaDeviceProp.totalGlobalMem / 1024 / 1024

<< " mb\n";

std::cout << "sharedMemoryPerBlock: "

<< cudaDeviceProp.sharedMemPerBlock / 1024

<< " kb.\n";

std::cout << "compute capabilities: " << cudaDeviceProp.major << " (major), " <<

cudaDeviceProp.minor << " (minor)\n";

auto maxGridSize = cudaDeviceProp.maxGridSize;

std::cout << "maxGridSize: "

<< maxGridSize[0] << " x "

<< maxGridSize[1] << " x "

<< maxGridSize[2] << "\n";

std::cout << "compute units: "

<< cudaDeviceProp.multiProcessorCount << "\n\n";

}

}

using Point = float3;

constexpr Point A{0.6, 0.7, -0.1};

constexpr Point B{0.1, -1.3, -0.27};

constexpr Point C{-2.1, 0.4, -0.5};

constexpr Point D{0.3, 0.1, 0.4};

constexpr auto X\_SET = {A.x, B.x, C.x, D.x};

constexpr auto Y\_SET = {A.y, B.y, C.y, D.y};

constexpr auto Z\_SET = {A.z, B.z, C.z, D.z};

constexpr float MIN\_X = std::min(X\_SET);

constexpr float MAX\_X = std::max(X\_SET);

constexpr float MIN\_Y = std::min(Y\_SET);

constexpr float MAX\_Y = std::max(Y\_SET);

constexpr float MIN\_Z = std::min(Z\_SET);

constexpr float MAX\_Z = std::max(Z\_SET);

constexpr float CUBE\_VOLUME = abs(

(MAX\_X - MIN\_X) \*

(MAX\_Y - MIN\_Y) \*

(MAX\_Z - MIN\_Z)

);

constexpr auto ITERATIONS = 100'000'000;

constexpr auto ITERATIONS\_PER\_THREAD = 1000;

constexpr auto TOTAL\_USED\_THREADS = ITERATIONS / ITERATIONS\_PER\_THREAD;

constexpr int MAX\_THREADS\_PER\_BLOCK = 1024;

constexpr int THREADS\_IN\_WARP = 32;

constexpr int WARPS\_PER\_BLOCK = MAX\_THREADS\_PER\_BLOCK / THREADS\_IN\_WARP;

class Plane {

public:

constexpr Plane(

Point p1,

Point p2,

Point p3) :

a(GetDeterminant2D({p2.y - p1.y, p2.z - p1.z},

{p3.y - p1.y, p3.z - p1.z})),

b(-GetDeterminant2D({p2.x - p1.x, p2.z - p1.z},

{p3.x - p1.x, p3.z - p1.z})),

c(GetDeterminant2D({p2.x - p1.x, p2.y - p1.y},

{p3.x - p1.x, p3.y - p1.y})),

d(-p1.x \* a - p1.y \* b - p1.z \* c) {

if (d < 0) {

a \*= -1;

b \*= -1;

c \*= -1;

d \*= -1;

}

}

static constexpr float GetDeterminant2D(float2 r1, float2 r2) {

return r1.x \* r2.y -

r1.y \* r2.x;

}

static constexpr float GetDeterminant3D(float3 r1, float3 r2, float3 r3) {

return r1.x \* GetDeterminant2D({r2.y, r2.z}, {r3.y, r3.z}) -

r2.x \* GetDeterminant2D({r1.y, r1.z}, {r3.y, r3.z}) +

r3.x \* GetDeterminant2D({r1.y, r1.z}, {r2.y, r2.z});

}

[[nodiscard]]

\_\_host\_\_

\_\_device\_\_

constexpr bool ContainsPoint(Point point) const {

return a \* point.x +

b \* point.y +

c \* point.z +

d >= 0;

}

public:

[[maybe\_unused]]

void Info() const {

std::cout << "A: " << a << "\n"

<< "B: " << b << "\n"

<< "C: " << c << "\n"

<< "D: " << d << "\n";

std::cout << "Contains {0, 0, 0}: " << std::boolalpha

<< ContainsPoint({0, 0, 0})

<< std::noboolalpha << "\n\n";

}

private:

float a, b, c, d;

};

\_\_device\_\_

constexpr Plane ABC(A, B, C);

\_\_device\_\_

constexpr Plane ABD(A, B, D);

\_\_device\_\_

constexpr Plane BCD(B, C, D);

\_\_device\_\_

constexpr Plane CAD(C, A, D);

\_\_host\_\_

\_\_device\_\_

constexpr float RandomNumber(float initial, float min, float max) {

return initial \* (max - min) + min;

}

constexpr float AnalyticComputeVolume() {

constexpr Point AB{B.x - A.x, B.y - A.y, B.z - A.z};

constexpr Point AC{C.x - A.x, C.y - A.y, C.z - A.z};

constexpr Point AD{D.x - A.x, D.y - A.y, D.z - A.z};

return abs(Plane::GetDeterminant3D(AB, AC, AD)) / 6;

}

float CPUComputeVolume() {

int hits{};

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

const auto x = RandomNumber(((float) rand() / (float) RAND\_MAX),

MIN\_X, MAX\_X);

const auto y = RandomNumber(((float) rand() / (float) RAND\_MAX),

MIN\_Y, MAX\_Y);

const auto z = RandomNumber(((float) rand() / (float) RAND\_MAX),

MIN\_Z, MAX\_Z);

const Point point{x, y, z};

if (ABC.ContainsPoint(point) &&

ABD.ContainsPoint(point) &&

BCD.ContainsPoint(point) &&

CAD.ContainsPoint(point)) {

hits++;

}

}

return CUBE\_VOLUME \* (float) hits / (float) ITERATIONS;

}

\_\_managed\_\_

unsigned long long totalHits = 0;

\_\_device\_\_

int ShuffleDownWarp(int value) {

constexpr auto MASK = 0xffffffff;

for (int offset = THREADS\_IN\_WARP / 2; offset > 0; offset /= 2) {

value += \_\_shfl\_down\_sync(MASK, value, offset);

}

return value;

}

\_\_device\_\_

int SumBlock(int value) {

\_\_shared\_\_ int reduceBuffer[WARPS\_PER\_BLOCK];

const auto warpLine = threadIdx.x % THREADS\_IN\_WARP;

const auto warpId = threadIdx.x / THREADS\_IN\_WARP;

value = ShuffleDownWarp(value);

if (warpLine == 0) {

reduceBuffer[warpId] = value;

}

\_\_syncthreads();

// Loads values back only in the first warp of the first block

value = (threadIdx.x < blockDim.x / THREADS\_IN\_WARP) ? reduceBuffer[warpLine] : 0;

if (warpId == 0) {

value = ShuffleDownWarp(value);

}

return value;

}

\_\_global\_\_

void KernelCountHits() {

const auto threadId = blockIdx.x \* blockDim.x + threadIdx.x;

if (threadId >= TOTAL\_USED\_THREADS) {

return;

}

curandState randomState;

curand\_init(1234, threadId, 0, &randomState);

int hits = 0;

for (int i = 0; i < ITERATIONS\_PER\_THREAD; ++i) {

const Point point{

RandomNumber(curand\_uniform(&randomState), MIN\_X, MAX\_X),

RandomNumber(curand\_uniform(&randomState), MIN\_Y, MAX\_Y),

RandomNumber(curand\_uniform(&randomState), MIN\_Z, MAX\_Z),

};

if (ABC.ContainsPoint(point) &&

ABD.ContainsPoint(point) &&

BCD.ContainsPoint(point) &&

CAD.ContainsPoint(point)) {

++hits;

}

}

const int blockHits = SumBlock(hits);

if (threadIdx.x == 0) {

atomicAdd(&totalHits, blockHits);

}

}

float GPUComputeVolume() {

constexpr int THREADS\_PER\_BLOCK = 128;

constexpr int BLOCKS\_PER\_GRID = (TOTAL\_USED\_THREADS + THREADS\_PER\_BLOCK - 1) / THREADS\_PER\_BLOCK;

KernelCountHits<<<BLOCKS\_PER\_GRID, THREADS\_PER\_BLOCK>>>();

cudaDeviceSynchronize();

return CUBE\_VOLUME \* (float) totalHits / (float) ITERATIONS;

}

bool SameResults(float analyticVolume, float CPUVolume, float GPUVolume) {

return abs(GPUVolume - analyticVolume) < 0.1 &&

abs(GPUVolume - CPUVolume) < 0.001;

}

int main() {

srand(time(nullptr));

PrintGPUsInfo();

// ABC.Info();

// ABD.Info();

// BCD.Info();

// CAD.Info();

auto analyticVolume = AnalyticComputeVolume();

std::cout << "AnalyticComputeVolume: " << analyticVolume << "\n";

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

auto cpuVolume = CPUComputeVolume();

std::cout << "CPUComputeVolume: " << cpuVolume << "\n";

std::cout << "CPU time: "

<< duration\_cast<milliseconds>(high\_resolution\_clock::now() - start).count()

<< " ms\n";

cudaEvent\_t eventStart, eventStop;

cudaEventCreate(&eventStart);

cudaEventCreate(&eventStop);

cudaEventRecord(eventStart);

auto gpuVolume = GPUComputeVolume();

std::cout << "GPUComputeVolume: " << gpuVolume << "\n";

cudaEventSynchronize(eventStop);

float milliseconds;

cudaEventElapsedTime(&milliseconds, eventStart, eventStop);

std::cout << "GPU time: " << milliseconds << " ms\n";

std::cout << "Same volumes: " << std::boolalpha

<< SameResults(analyticVolume, cpuVolume, gpuVolume) << " \n";

cudaEventDestroy(eventStart);

cudaEventDestroy(eventStop);

return 0;

}

**3. Вывод**

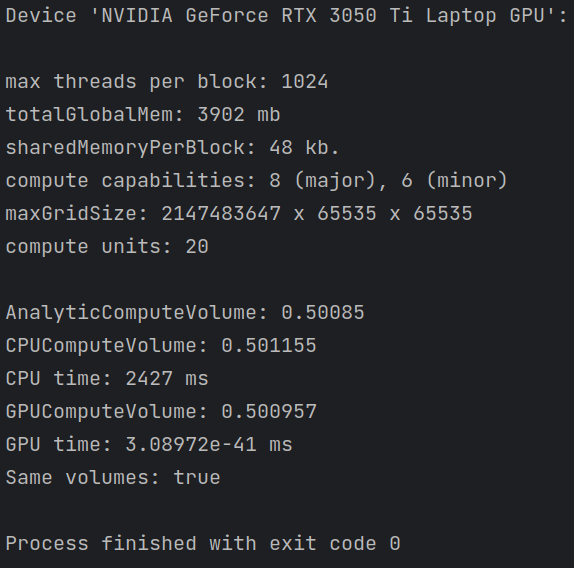


Рисунок 2 – Результат работы программы

Результат работы программы представлен на рисунке 2.

В ходе выполнения лабораторной работы был изучен Метод Монте-Карло для приближенного вычисления объема фигуры с заданными вершинами. По результатам работы можно сказать, что нам удалось ускорить вычисления, применяя GPU, относительно вычислений на CPU. Использовали метод агрегации на уровне блоков - parallel reduce, на уровне сетки – atomic add. Для нитей использовали разделяемую память;