Министерство науки и высшего образования РФ

Федеральное государственное автономное

образовательное учреждение высшего образования

**«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**

Институт космических и информационных технологий

Кафедра вычислительной техники

**ОТЧЁТ О ПРАКТИЧЕСКОЙ РАБОТЕ №1**

**по дисциплине**

**«Гибридные вычислительные системы»**

«Изучение модели программирования CUDA»

|  |  |  |
| --- | --- | --- |
| Преподаватель | \_\_\_\_\_\_\_\_\_\_\_\_ | С. А. Тарасов |
|  | подпись, дата | инициалы, фамилия |
| Студент КИ22-07Б, 032212677 | \_\_\_\_\_\_\_\_\_\_\_\_ | Л. А. Глушков |
| номер группы, зачетной книжкой | подпись, дата | инициалы, фамилия |
| Студент КИ22-07Б, 032215583 | \_\_\_\_\_\_\_\_\_\_\_\_ | А. М. Коробков |
| номер группы, зачетной книжкой | подпись, дата | инициалы, фамилия |
| Студент КИ22-07Б, 032214653 | \_\_\_\_\_\_\_\_\_\_\_\_ | И. О. Бердин |
| номер группы, зачетной книжкой | подпись, дата | инициалы, фамилия |

Красноярск 2025

**СОДЕРЖАНИЕ**

[1 Задание 3](#_Toc209010684)

[2 Исходный код 4](#_Toc209010685)

[2.1 Core 4](#_Toc209010686)

[2.2 benchmarks 5](#_Toc209010687)

[2.3 Tests 7](#_Toc209010688)

[3 Выполнение программы 9](#_Toc209010689)

[ЗАКЛЮЧЕНИЕ 10](#_Toc209010690)

# **1 Задание**

1. Написать программу на языке CUDA C/C++, вычисляющую функцию от вещественных аргументов в двух реализациях – для CPU и GPU.
2. Добавить в программу функциональность для сравнения результатов работы этих реализаций по сложности вычисления и возвращаемым значениями.
3. Выполнить вычислительный эксперимент, результатом которого должны быть графики реальных вычислительных сложностей двух реализаций.

# **2 Исходный код**

## **2.1 Core**

#include <iostream>

#include "gpu\_addVect.cuh"

#include <cuda\_runtime.h>

namespace AddVect {

\_\_global\_\_ void GpuAddVect(float\* vect1, float\* vect2, float\* resultVect) {

int i = threadIdx.x;

resultVect[i] = vect1[i] + vect2[i];

};

void FullGpuAddVect(float\* vect1, float\* vect2, float\* resultVect, int gridSize, float\* kernel\_ms) {

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start, 0);

GpuAddVect << < 1, gridSize >> > (vect1, vect2, resultVect);

cudaEventRecord(stop);

cudaEventSynchronize(stop);

cudaEventElapsedTime(kernel\_ms, start, stop);

};

float\* AddingVectors::CpuAddVect(float\* vect1, float\* vect2, float\* resultVect, int gridSize) {

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

resultVect[i] = vect1[i] + vect2[i];

}

return resultVect;

};

float\* AddingVectors::RunGpu(float\* vect1, float\* vect2, float\* resultVect, int gridSize) {

float\* devVect1, \* devVect2, \* devResult;

cudaMalloc((void\*\*)&devVect1, sizeof(float) \* gridSize);

cudaMalloc((void\*\*)&devVect2, sizeof(float) \* gridSize);

cudaMalloc((void\*\*)&devResult, sizeof(float) \* gridSize);

cudaMemcpy(devVect1, vect1, sizeof(float) \* gridSize, cudaMemcpyHostToDevice);

cudaMemcpy(devVect2, vect2, sizeof(float) \* gridSize, cudaMemcpyHostToDevice);

cudaMemcpy(devResult, resultVect, sizeof(float) \* gridSize, cudaMemcpyHostToDevice);

GpuAddVect << < 1, gridSize >> > (devVect1, devVect2, devResult);

cudaMemcpy(resultVect, devResult, sizeof(float) \* gridSize, cudaMemcpyDeviceToHost);

cudaFree(devVect1);

cudaFree(devVect2);

cudaFree(devResult);

return resultVect;

};

};

## **2.2 benchmarks**

#include <benchmark/benchmark.h>

#include <vector>

#include <random>

#include <algorithm>

#include <cmath>

#include "gpu\_addVect.cuh"

static void Bench\_CPU(benchmark::State& state) {

int N = state.range(0);

std::vector<float> a(N), b(N), cpu\_res(N);

std::mt19937 rng(12345);

std::uniform\_real\_distribution<float> dist(-100.0f, 100.0f);

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

a[i] = dist(rng);

b[i] = dist(rng);

}

for (auto \_ : state) {

AddVect::AddingVectors::CpuAddVect(a.data(), b.data(), cpu\_res.data(), N);

benchmark::DoNotOptimize(cpu\_res.data());

}

}

static void Bench\_GPU(benchmark::State& state) {

int N = state.range(0);

std::vector<float> a(N), b(N), gpu\_res(N);

std::mt19937 rng(12345);

std::uniform\_real\_distribution<float> dist(-100.0f, 100.0f);

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

a[i] = dist(rng);

b[i] = dist(rng);

}

for (auto \_ : state) {

AddVect::AddingVectors::RunGpu(a.data(), b.data(), gpu\_res.data(), N);

benchmark::DoNotOptimize(gpu\_res.data());

}

}

static void Bench\_GPU\_Esh(benchmark::State& state) {

int N = state.range(0);

std::vector<float> a(N), b(N), gpu\_res(N);

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

a[i] = 0.1f \* i;

b[i] = 0.2f \* i;

}

float\* devVect1, \* devVect2, \* devResult;

cudaMalloc((void\*\*)&devVect1, sizeof(float) \* N);

cudaMalloc((void\*\*)&devVect2, sizeof(float) \* N);

cudaMalloc((void\*\*)&devResult, sizeof(float) \* N);

cudaMemcpy(devVect1, a.data(), sizeof(float) \* N, cudaMemcpyHostToDevice);

cudaMemcpy(devVect2, b.data(), sizeof(float) \* N, cudaMemcpyHostToDevice);

cudaMemcpy(devResult, gpu\_res.data(), sizeof(float) \* N, cudaMemcpyHostToDevice);

float\* kms = new float;

for (auto \_ : state) {

AddVect::FullGpuAddVect(devVect1, devVect2, devResult, N, kms);

state.SetIterationTime(\*kms);

benchmark::DoNotOptimize(devResult);

}

cudaFree(devVect1);

cudaFree(devVect2);

cudaFree(devResult);

delete kms;

}

BENCHMARK(Bench\_CPU)

->RangeMultiplier(2)

->Range(1 << 10, 1 << 20);

BENCHMARK(Bench\_GPU)

->RangeMultiplier(2)

->Range(1 << 10, 1 << 20);

BENCHMARK(Bench\_GPU\_Esh)

->RangeMultiplier(2)

->Range(1 << 10, 1 << 20);

BENCHMARK\_MAIN();

## **2.3 Tests**

#include <gtest/gtest.h>

#include <vector>

#include <random>

#include <algorithm>

#include <cmath>

#include "gpu\_addVect.cuh"

std::vector<int> g\_vector\_sizes = { 256 };

class AddVectParamTest : public ::testing::TestWithParam<int> {};

std::vector<int> ParseVectorSizes(const std::string& arg) {

std::vector<int> result;

size\_t pos = arg.find("=");

if (pos == std::string::npos) return result;

std::string values = arg.substr(pos + 1);

std::stringstream ss(values);

std::string item;

while (std::getline(ss, item, ',')) {

try {

result.push\_back(std::stoi(item));

}

catch (...) {

std::cerr << "Invalid integer in --vector\_sizes: " << item << "\n";

}

}

return result;

}

TEST\_P(AddVectParamTest, CpuVsGpuAccuracy) {

const int N = GetParam();

std::cout << "N -> " << N;

std::vector<float> a(N), b(N), cpu\_res(N), gpu\_res(N);

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

a[i] = 0.1f \* i;

b[i] = 0.2f \* i;

}

AddVect::AddingVectors::CpuAddVect(a.data(), b.data(), cpu\_res.data(), N);

AddVect::AddingVectors::RunGpu(a.data(), b.data(), gpu\_res.data(), N);

cudaDeviceSynchronize();

const float abs\_error = 1e-6f;

float max\_diff = 0.0f;

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

float d = std::fabs(cpu\_res[i] - gpu\_res[i]);

if (d > max\_diff) max\_diff = d;

}

ASSERT\_LE(max\_diff, abs\_error) << "Max absolute difference (" << max\_diff << ") exceeds abs\_error " << abs\_error;

std::cout << "Metrics test complete successfull!\n";

}

INSTANTIATE\_TEST\_SUITE\_P(

VectorSizes,

AddVectParamTest,

::testing::ValuesIn(g\_vector\_sizes)

);

int main(int argc, char\*\* argv)

{

for (int i = 1; i < argc; ++i) {

std::string arg = argv[i];

if (arg.rfind("--vector\_sizes=", 0) == 0) {

g\_vector\_sizes = ParseVectorSizes(arg);

for (int j = i; j < argc - 1; ++j) {

argv[j] = argv[j + 1];

}

--argc;

break;

}

}

::testing::InitGoogleTest(&argc, argv);

return RUN\_ALL\_TESTS();

}

# **3 Выполнение программы**

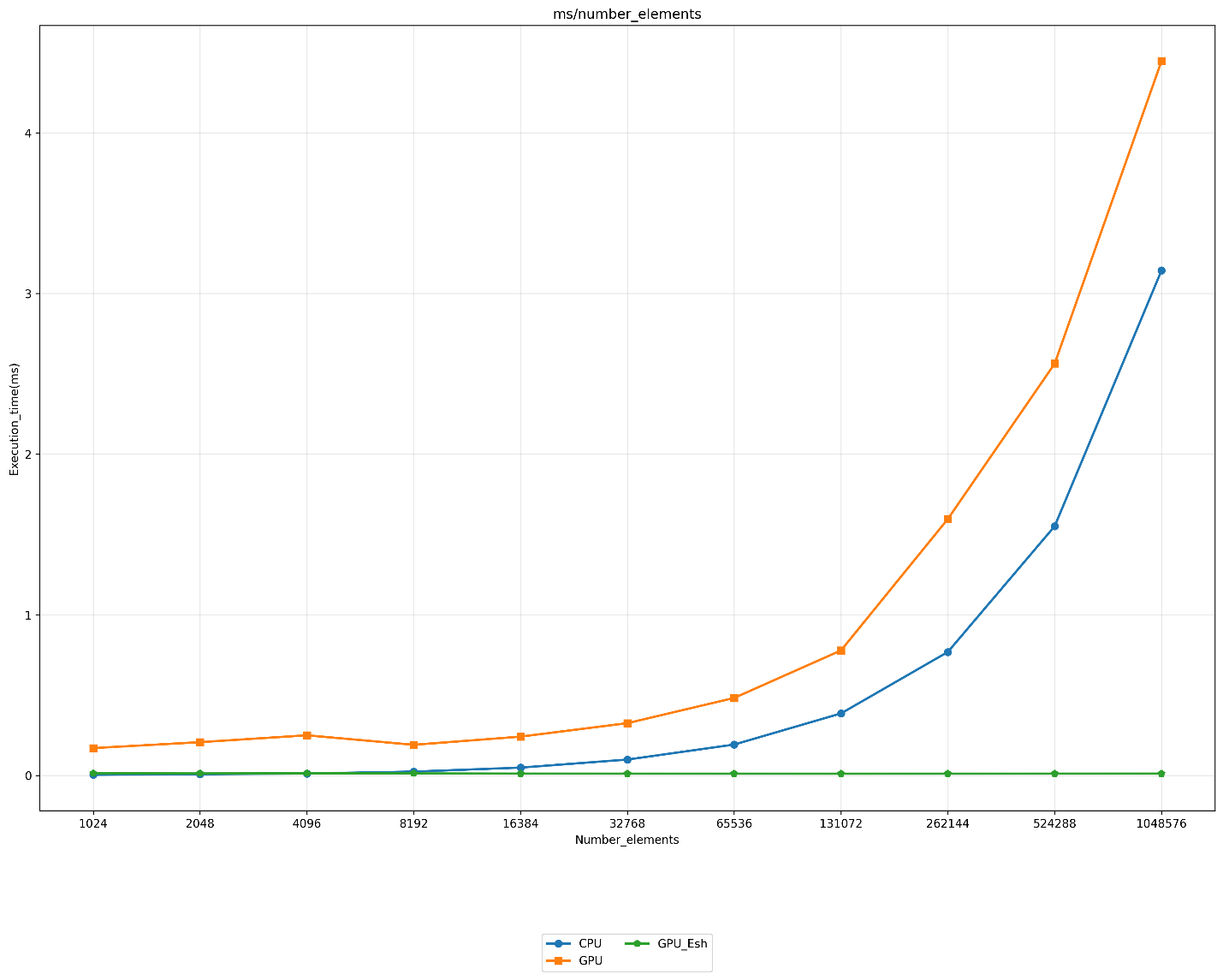


Рисунок 1 – Графики вычислительных сложностей.

На рисунке 1 изображены графики вычислительных сложностей. Полученные результаты, где время вычислений на CPU превышает время вычислений на GPU, являются ожидаемыми, т.к. графический процессор имеет большее количество вычислительных ядер.

# **ЗАКЛЮЧЕНИЕ**

При выполнении данной работы были изучены модели программирования CUDA.