Министерство цифрового развития, связи
и массовых коммуникаций Российской Федерации

Сибирский государственный университет
телекоммуникаций и информатики

Кафедра прикладной математики и кибернетики

# ЛАБОРАТОРНАЯ РАБОТА №2

По дисциплине: «Программирование графических процессоров»

Выполнили:

Студенты 3 курса группы ИП-211
Оганесян А.С.
Лацук А.Ю.
Проверил:

Профессор кафедры ПМиК
Малков Е.А.

Новосибирск 2025

**Задание:** 1. Определить при какой длине векторов имеет смысл распараллеливать операцию сложения, используя потоки CPU или GPU.

2. Определить оптимальное количество потоков POSIX для распараллеливания.

3. Определить зависимость времени выполнения операции сложения на GPU от длины векторов (выбирать количество нитей равным длине вектора).

**Цель:** начальное знакомство с распараллеливанием кода на GPU .

**Выполнение работы:** Для первого задания были написаны программы для сложения n векторов для одного потока, нескольких потоков CPU и с использованием GPU.

```
#include <iostream>
#include <vector>
#include <thread>
#include <chrono>
using namespace std;
const int n = 100000000;
typedef std::chrono::milliseconds ms;
typedef std::chrono::nanoseconds ns;


int main() {
    vector<float> a(n), b(n), c(n);
    chrono::time_point<chrono::system_clock> start, end;

    for (int i = 0; i < n; ++i) {
        a[i] = i;
        b[i] = i * 2;
    }

    start = chrono::system_clock::now();
    for (int i = 0; i < n; ++i) {
        c[i] = a[i] + b[i];
    }
    end = chrono::system_clock::now();

    cout << "Wasted time: " << chrono::duration_cast<ms>(end -
start).count() <<"ms" << endl
         << chrono::duration_cast<ns>(end - start).count() <<
"ns";
    return 0;
}
```

Листинг 1 – программа main.cpp

Результат работы программы:

```
Wasted time: 1195ms
1195286800ns
```

```
#include <iostream>
#include <vector>
#include <thread>
#include <chrono>
using namespace std;
const int n = 100000000;
typedef std::chrono::milliseconds ms;
typedef std::chrono::nanoseconds ns;

void vectorAdd(const vector<float> &a, const vector<float> &b,
vector<float> &c, int start, int end) {
    for (int i = start; i < end; ++i) {
        c[i] = a[i] + b[i];
    }
}

int main() {
    vector<float> a(n), b(n), c(n);
    int numThreads = thread::hardware_concurrency();
    //int numThreads = 4;

    chrono::time_point<chrono::system_clock> start, end;

    for (int i = 0; i < n; ++i) {
        a[i] = i;
        b[i] = i * 2;
    }

    vector<thread> threads;
    for (int i = 0; i < numThreads; ++i) {
        int start = i * (n / numThreads);
        int end = (i == numThreads - 1) ? n : (i + 1) * (n /
numThreads);
        threads.emplace_back(vectorAdd, ref(a), ref(b), ref(c),
start, end);
    }

    start = chrono::system_clock::now();
```

```
    for (auto &thread : threads) {
        thread.join();
    }
    end = chrono::system_clock::now();

    cout << "Wasted time: " << chrono::duration_cast<ms>(end -
start).count() << "ms" << endl
        << chrono::duration_cast<ns>(end - start).count() <<
"ns";
    return 0;
}
```

Листинг 2 – программа main_2.cpp

Результат работы программы:

```
Wasted time: 211ms
211703100ns
```

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <iostream>
#include <chrono>
using namespace std;
const int n = 100000000;
typedef std::chrono::milliseconds ms;
typedef std::chrono::nanoseconds ns;

__global__ void vectorAdd(const float* a, const float* b, float*
c, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        c[i] = a[i] + b[i];
    }
}

int main() {
```

```
    float elapsedTime;
    cudaEvent_t start, stop;
    chrono::time_point<chrono::system_clock> start_chrono,
end_chrono;

    float* d_a, * d_b, * d_c;
    cudaMalloc((void**)&d_a, n * sizeof(float));
    cudaMalloc((void**)&d_b, n * sizeof(float));
    cudaMalloc((void**)&d_c, n * sizeof(float));

    float* h_a = new float[n];
    float* h_b = new float[n];
    for (int i = 0; i < n; ++i) {
        h_a[i] = i;
        h_b[i] = i * 2;
    }

    cudaMemcpy(d_a, h_a, n * sizeof(float),
cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, n * sizeof(float),
cudaMemcpyHostToDevice);

    // Вычисляем количество блоков и нитей на блок
    int blockSize = 1024;
    int numBlocks = n;

    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaEventRecord(start, 0);
    start_chrono = chrono::system_clock::now();
    vectorAdd <<< numBlocks, blockSize >>> (d_a, d_b, d_c, n);
    cudaEventRecord(stop, 0);
    end_chrono = chrono::system_clock::now();

    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&elapsedTime, start, stop);
```

```
    float* h_c = new float[n];
    cudaMemcpy(h_c, d_c, n * sizeof(float),
cudaMemcpyDeviceToHost);

    cout <<"CUDA Event time: "<< elapsedTime << endl
         <<"Chrono time: "<<
chrono::duration_cast<ms>(end_chrono - start_chrono).count() <<
"ms"
        << endl << chrono::duration_cast<ns>(end_chrono -
start_chrono).count() << "ns";


    delete[] h_a;
    delete[] h_b;
    delete[] h_c;
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

Листинг 3 – программа main.cu

Результат работы программы:

```
CUDA Event time: 0.06544
Chrono time: 0ms
84800ns
```

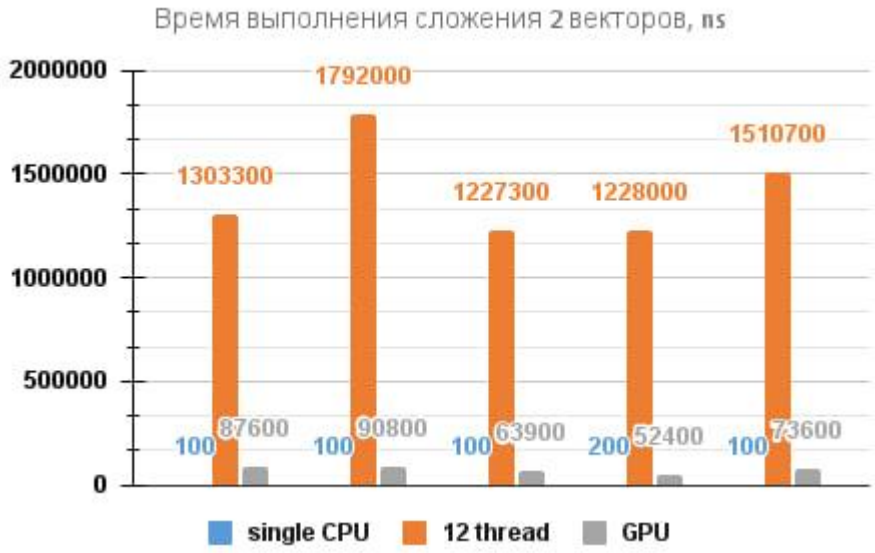
Измерим время работы программ на разном векторов:

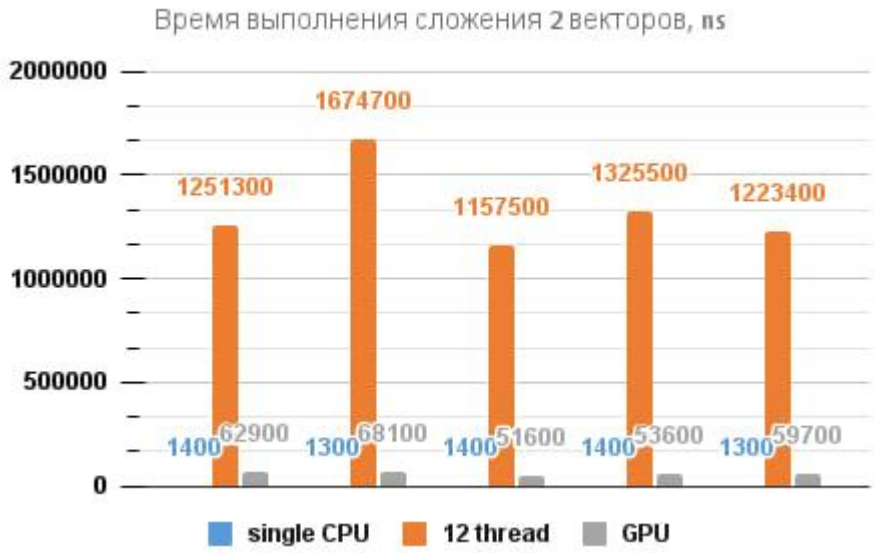
| | single CPU | | 12 thread | | GPU | | |
|---|---|---|---|---|---|---|---|
| | ms | ns | ms | ns | event | ms | ns |
| 2 | 0 | 100 | 1 | 1.510.700 | 0,079872 | 0 | 73.600 |
| | 0 | 200 | 1 | 1.228.000 | 0,058368 | 0 | 52.400 |
| | 0 | 100 | 1 | 1.227.300 | 0,067584 | 0 | 63.900 |
| | 0 | 100 | 1 | 1.792.000 | 0,095232 | 0 | 90.800 |
| | 0 | 100 | 1 | 1.303.300 | 0,092192 | 0 | 87.600 |
| 100 | 0 | 1.300 | 1 | 1.223.400 | 0,065536 | 0 | 59.700 |
| | 0 | 1.400 | 1 | 1.325.500 | 0,057504 | 0 | 53.600 |
| | 0 | 1.400 | 1 | 1.157.500 | 0,546816 | 0 | 51.600 |

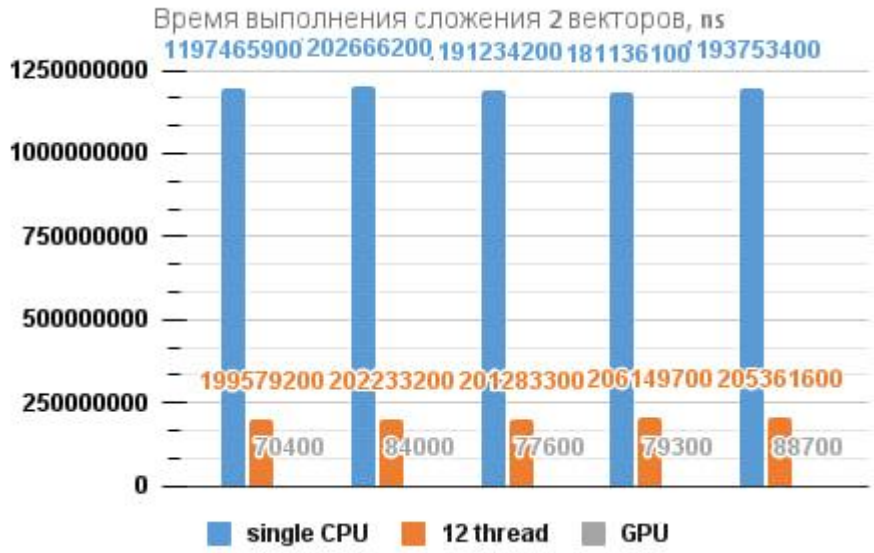
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 0 | 1.300 | 1 | 1.674.700 | 0,007168 | 0 | 68.100 |
| | 0 | 1.400 | 1 | 1.251.300 | 0,067584 | 0 | 62.900 |
| 10.000 | 0 | 183.000 | 1 | 1.701.100 | 0,058368 | 0 | 53.700 |
| | 0 | 168.000 | 1 | 1.459.400 | 0,06144 | 0 | 57.100 |
| | 0 | 127.700 | 1 | 1.486.100 | 0,06144 | 0 | 55.900 |
| | 0 | 194.200 | 1 | 1.430.200 | 0,056288 | 0 | 51.500 |
| | 0 | 117.800 | 1 | 1.302.200 | 0,060416 | 0 | 56.100 |
| 100.000.000 | 1193 | 1.193.753.400 | 205 | 205.361.600 | 0,099648 | 0 | 88.700 |
| | 1181 | 1.181.136.100 | 206 | 206.149.700 | 0,067264 | 0 | 79.300 |
| | 1191 | 1.191.234.200 | 201 | 201.283.300 | 0,064896 | 0 | 77.600 |
| | 1202 | 1.202.666.200 | 202 | 202.233.200 | 0,067968 | 0 | 84.000 |
| | 1197 | 1197465900 | 199 | 199.579.200 | 0,043424 | 0 | 70.400 |

Таблица 1 – Замеры программ с разным количеством векторов.

В графиках ниже приведены визуальные сравнения работы трех программ по времени:




По графикам видно что использовать один поток эффективнее на небольшом количестве векторов (<1.000.000), после потом эффективней использовать вычисления с и пользованием многопоточности. GPU же показывает наибольшую эффективность уже больше 10.000 векторов.

Вывод: Мы познакомились с распараллеливанием кода на GPU и определили, что она наиболее эффективен на большом количестве данных