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-111
Корнилов А.А.,
Попов М.И.,
Толкач А.А.

Проверил:

Профессор кафедры ПМиК
Малков Е.А.

Новосибирск, 2024

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, линейная (лист. 1)
- для многопоточности, используя библиотеку thread (лист. 2)
- для вычисления на GPU, используя CUDA (лист. 3)

Для замеров времени используется STL библиотека Chrono, для вычисления с GPU еще и CudaEvent, в программа для примера n = 100000000, количество нитей 1024. Для вычислений с threads было взято максимальное количество потоков процесса с помощью thread::hardware_concurrency(), в нашем случае это число было 12.

```
#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 LR02_1C.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 1 – программа LR02_2C.cpp

Результат работы программы:

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

```cpp
#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 1 – программа LR02_1G.cu

Результат работы программы:

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

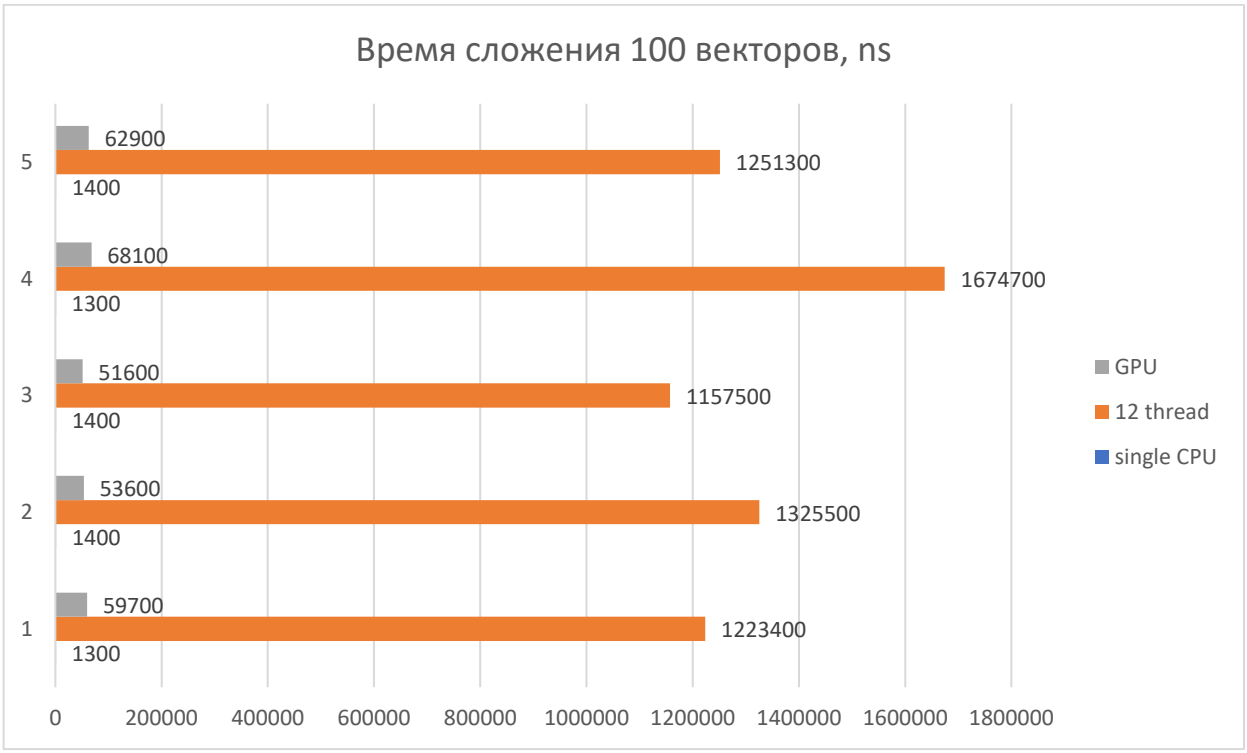
Для вычисления оптимально метода и количества потоков были проведены замеры трех программам с количеством векторов: 2, 100, 10000 и 100000000. Результаты в таблице 1

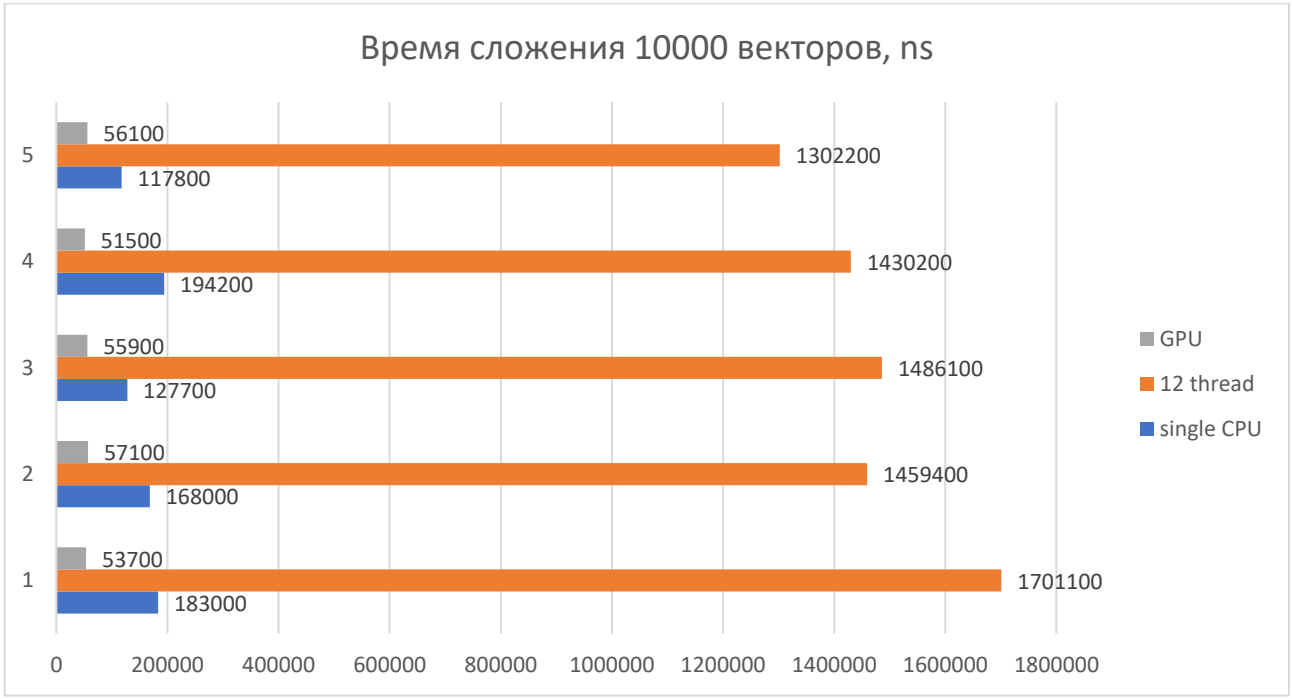
	single CPU		12 thread		GPU		
	ms	ns	ms	ns	event	ms	ns
2	0	100	1	1510700	0,079872	0	73600
	0	200	1	1228000	0,058368	0	52400
	0	100	1	1227300	0,067584	0	63900
	0	100	1	1792000	0,095232	0	90800
	0	100	1	1303300	0,092192	0	87600
100	0	1300	1	1223400	0,065536	0	59700
	0	1400	1	1325500	0,057504	0	53600
	0	1400	1	1157500	0,546816	0	51600
	0	1300	1	1674700	0,007168	0	68100
	0	1400	1	1251300	0,067584	0	62900
10000	0	183000	1	1701100	0,058368	0	53700
	0	168000	1	1459400	0,06144	0	57100
	0	127700	1	1486100	0,06144	0	55900
	0	194200	1	1430200	0,056288	0	51500
	0	117800	1	1302200	0,060416	0	56100
100000000	1193	1193753400	205	205361600	0,099648	0	88700
	1181	1181136100	206	206149700	0,067264	0	79300
	1191	1191234200	201	201283300	0,064896	0	77600
	1202	1202666200	202	202233200	0,067968	0	84000

	1197	1197465900	199	199579200	0,043424	0	70400

Таблица 1 – Замеры программ с разным количеством векторов.

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





По графикам можно понять что в случае с вычисление процессором то для небольшого количества векторов (>1.000.000) можно использовать линейные вычисления, после 1.000.000 уже эффективней использовать вычисления с и пользованием многопоточности. В случае вычисление с GPU то её преимущество уже видно после 10.000 векторов

Таблица Excel будет приведена в том-же письме что и отчет, вычисления проводились на GTX 1050ti 4Gb