МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра Автоматизированных систем управления



**Расчетно-графическая работа**

**По «Параллельному программированию»**

**на тему: «Численное интегрирование методом Симпсона»**

Выполнила: Проверил: Студентка гр. *АВТ - 713*, *АВТФ Доцент*

*Тягунова В.В. Ландовский В.В.*

Новосибирск,

2019

**Задание:**

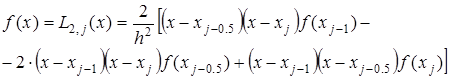
Произвести численное интегрирование методом Симпсона четырьмя различными способами - с использованием четырех различных технологий, одна из которых будет являться последовательным решением этой задачи. Сравнить реализации между собой. Получить анализ полученных данных. Оценить зависимости временных затрат от количества потоков и от объема данных. Сделать выводы о полученных результатах.

**Теоретическая часть:**

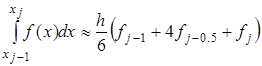
**Описание задачи:**

*Метод Симпсона*:

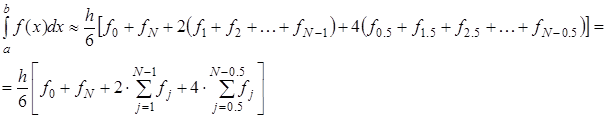
В этом методе подынтегральная функция на частичном отрезке http://aco.ifmo.ru/el_books/numerical_methods/lectures/images/image051.png аппроксимируется параболой, проходящей через три точки http://aco.ifmo.ru/el_books/numerical_methods/lectures/images/image066.png, http://aco.ifmo.ru/el_books/numerical_methods/lectures/images/image067.png, http://aco.ifmo.ru/el_books/numerical_methods/lectures/images/image068.png, то есть интерполяционным многочленом Лагранжа второй степени:

           (1)

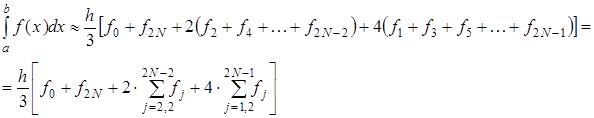
Проведя интегрирование, получим:

           (2)

Это и есть формула Симпсона или формула парабол. На отрезке http://aco.ifmo.ru/el_books/numerical_methods/lectures/images/image047.png формула Симпсона примет вид:

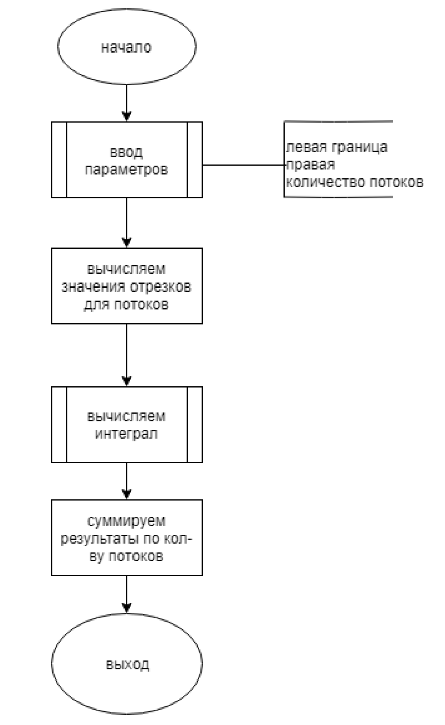
(3)

Если разбить отрезок интегрирования http://aco.ifmo.ru/el_books/numerical_methods/lectures/images/image072.png на **четное** количество 2*N* равных частей с шагом http://aco.ifmo.ru/el_books/numerical_methods/lectures/images/image073.png, то можно построить параболу на каждом сдвоенном частичном отрезке http://aco.ifmo.ru/el_books/numerical_methods/lectures/images/image051.png и переписать выражения (2.12-2.14) без дробных индексов. Тогда формула Симпсона примет вид:

             (4)

**Описание разработанных алгоритмов:**

*Описание последовательного алгоритма:*



*Описание алгоритма в C++:*



*Описание алгоритма MPI:*

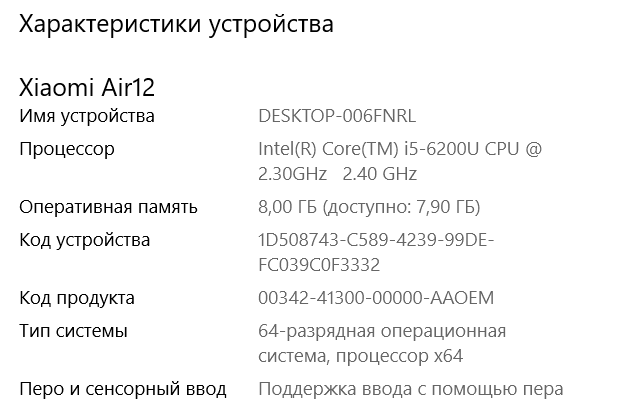
В главном процессе считываем значения из консоли. Затем с помощью функции MPI\_Bcast передаем значения переменных (начальная и конечная точки и количество шагов) из главного потока всем остальным. После этого для каждого процесса высчитываем собственную начальную точку, длину шага и количество шагов. После этого каждый процесс начинает работать со своими данными, затем вычисляет значение интеграла. В самом конце программы главной процесс с помощью функции MPI\_Reduce суммируем результаты всех процессов и выводит их пользователю.

*CUDA:*

В основном потоке считываем нужные данные из консоли, формируем два массива с данными (левая-правая границы). Запускаем ядро, при этом передаем ему массивы. После выполнения первого вызова ядра, вызывается сразу еще раз ядро с функцией сложения полученных результатов. Выводим результат и время на экран.

**Характеристики устройств:**

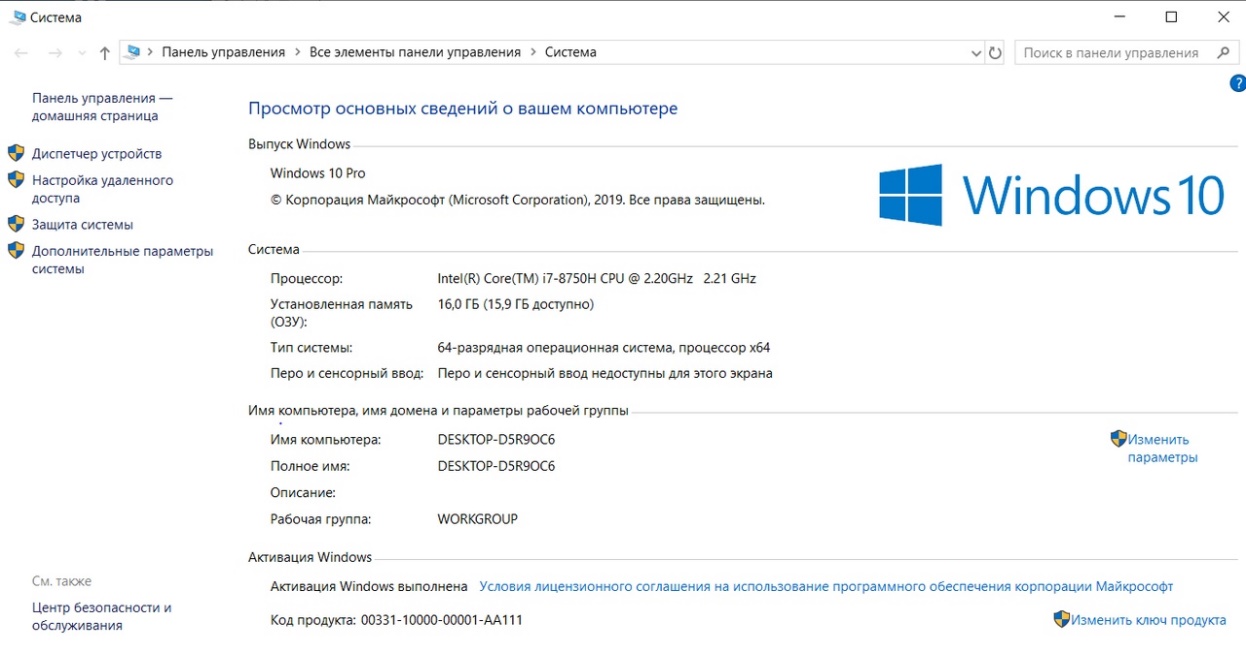
**1. Первое вычислительное устройство использовалось для программы на С++ и MPI**



*Рис. 1. Характеристики устройства*

**1. Первое вычислительное устройство использовалось для программы на CUDA**

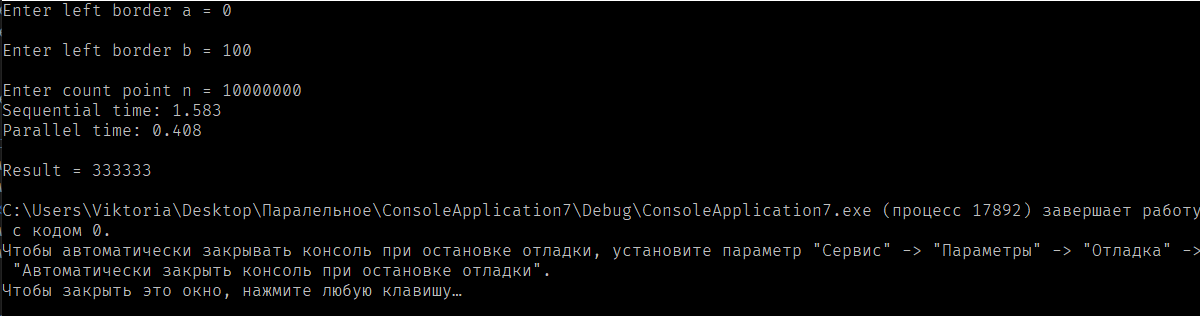
Видеокарта: NVIDIA GTX 1060 ti 6gb



*Рис. 2. Характеристики второго устройства*

**Экспериментальная часть:**

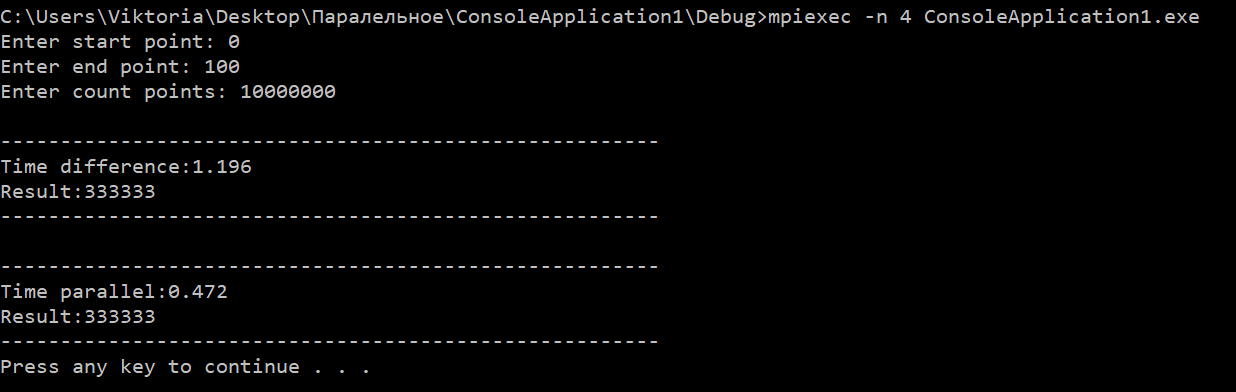
*Для последовательного и потоков: Для функции*



*Рис. 3. Пример работы последовательной программы и для программы с потоками*

Для того чтобы найти определенный интеграл с количеством шагов равным 10000000 потребовалось 1.583 мс, а для параллельного 0.408 мс.

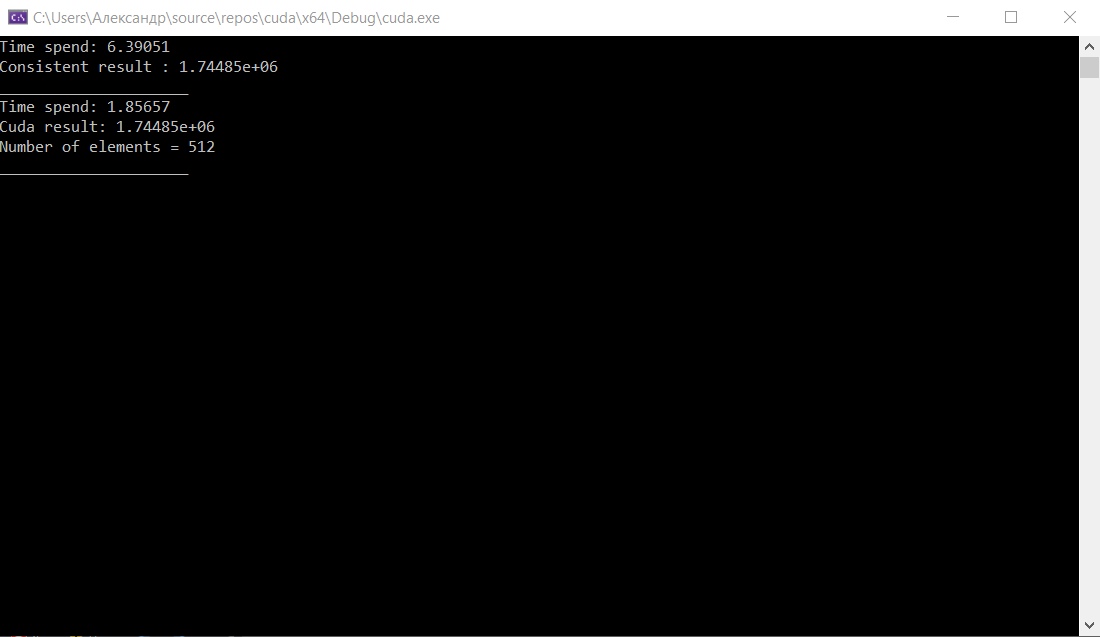
*MPI: Для функции*



*Рис. 4. Пример работы программы MPI*

*CUDA: Для функции*

Для того чтобы найти определенный интеграл с количеством шагов равным 10000000 потребовалось 0.476 мс для параллельного процесса, а для основного потребовалось 1.305 мс, при условии того что использовалось 4 процесса.



*Рис. 5. Пример работы программы CUDA*

Для того чтобы найти определенный интеграл для функции ((sin(x) + 1) / 13) + log(x + 1) + x / 1000) с диапазоном 0 до 50000 с точностью 0.00000001 потребовалось 1,39051 с для параллельного, а для последовательного потребовалось 6,39051.

Проведем эксперименты, которые будут показывать, как в параллельных программах время зависит от количества потоков и количества шагов интегрирования.

|  |  |
| --- | --- |
| Количество потоков | Время выполнения (мс) |
| 1 | 1,242 |
| 2 | 0,873 |
| 3 | 0,642 |
| 4 | 0,422 |
| 5 | 0,527 |
| 6 | 0,393 |
| 7 | 0,414 |
| 8 | 0,506 |
| 10 | 0,555 |
| 20 | 0,54 |

*Потоки в C++: Для функции*

*Таблица 1. Зависимость времени выполнения от количества потоков при 10000000 шагов*

*График 1. Зависимость времени выполнения от количества потоков при 10000000 шагов*

Как видно из графика с увеличением потоков время выполнения программы уменьшается, но при достижении 5 потоков снова начинает расти, объясняется это тем что компьютер, на котором проводился эксперимент, имеет 4 ядра процессора.

*Потоки в MPI: Для функции*

|  |  |
| --- | --- |
| Количество потоков | Время выполнения (мс) |
| 1 | 1,557 |
| 2 | 0,554 |
| 3 | 0,433 |
| 4 | 0,4 |
| 5 | 0,508 |
| 6 | 0,401 |
| 7 | 0,381 |
| 8 | 0,369 |
| 10 | 0,353 |
| 20 | 0,363 |

*Таблица 2. Зависимость времени выполнения от количества потоков при 10000000 шагов*

*График 2. Зависимость времени выполнения от количества потоков при 10000000 шагов*

Как видно из графика с увеличением процессов время выполнения программы уменьшается. Так же оно стремится к 0,4 мс, значит дальнейшее увеличение количества процессов не имеет смысла, для этого количества шагов интегрирования.

CUDA: *Для функции*

|  |  |
| --- | --- |
| Количество блоков | Время(с) |
| 8 | 10,5931 |
| 16 | 5,98074 |
| 32 | 3,91859 |
| 64 | 2,80085 |
| 128 | 2,27021 |
| 256 | 1,86668 |
| 521 | 1,86543 |

*Таблица 3. Зависимость времени выполнения от количества потоков при 10000000 шагов*

*График 2. Зависимость времени выполнения от количества потоков при 10000000 шагов*

На основе полученных графиков можно судить, что использование технологии CUDA оправданно только при необходимости обработки достаточно больших объемов данных, так как на пересылку данных затрачивается значительное время.

**Вывод:**

В ходе работы был реализован последовательный и параллельный алгоритм вычисления определенного интеграла методом Симпсона. При выполнении данной работы было изучено три инструмента для распараллеливания вычислений: POSIX, MPI и CUDA.

POSIX – это отличное средство для распараллеливания вычислений, которое не требует дополнительного времени для запуска. Данный вид может подойти к задаче, в которой требуется немного времени для выполнения программы.

MPI – это такая технология, которая уже на начальном этапе требует немного времени. Но в отличие от POSIX уже может общаться со своими процессами, не используя мьютексов. Она дает большое преимущество в вычислениях, где требуется частый обмен информации между процессами.

CUDA – это такое решение для распараллеливания процессов, которое позволяет использовать CPU и GPU, то есть помимо вычислений на процессоре, мы можем вычислять на видеокарте. Однако этот подход имеет небольшой минус: время запуска данной технологии очень большое, но при этом она может дать большой выигрыш в скорости распараллеливания вычислений, в отличие от POSIX, MPI. Таким образом можно подвести итог, что CUDA отлично подходит для длительных и сложных вычислений.

В ходе тестирования было выявлено, что самым быстрым способом является использование POSIX.

**Листинг программ:**

*Последовательная и потоки в C++:*

#include "pch.h"

#include <iostream>

#include "omp.h"

#include <time.h>

#include <math.h>

#include <cmath>

using namespace std;

double f(double x) {

return x\*x;

}

double simpson\_integral\_par(double a, double b, int n) {

const double h = (b - a) / n;

double k1 = 0, k2 = 0;

#pragma omp parallel for reduction(+:k1,k2) num\_threads(20)

for (int i = 1; i < n; i += 2) {

k1 += f(a + i \* h);

k2 += f(a + (i + 1)\*h);

}

return h / 3 \* (f(a) + 4 \* k1 + 2 \* k2);

}

double simpson\_integral(double a, double b, int n) {

const double h = (b - a) / n;

double k1 = 0, k2 = 0;

for (int i = 1; i < n; i += 2) {

k1 += f(a + i \* h); //вычисление значения с нечетными индексами, которые нужно множить на 4

k2 += f(a + (i + 1)\*h); // вычисление с четными индексами, которые нужно будет умножить на 2

}

return h / 3 \* (f(a) + 4 \* k1 + 2 \* k2);

}

int main() {

double a, b;

double result;

int n;

cout << "Enter left border a = ";

cin >> a;

cout << "\nEnter left border b = ";

cin >> b;

cout << "\nEnter count point n = ";

cin >> n;

clock\_t clock\_start = clock();

result = simpson\_integral(a, b, n);

double time\_c = (double)(clock() - clock\_start) / CLOCKS\_PER\_SEC;

cout << "Sequential time: " << time\_c << endl;

clock\_t clock\_start\_par = clock();

result = simpson\_integral\_par(a, b, n);

double time\_c\_par = (double)(clock() - clock\_start\_par) / CLOCKS\_PER\_SEC;

cout << "Parallel time: " << time\_c\_par << endl;

cout << "\nResult = " << result << endl;

}

*MPI:*

#include "pch.h"

#include <iostream>

#include <mpi.h>

#include <iostream>

#include <math.h>

#include <conio.h>

#include <stdio.h>

#include <time.h>

using namespace std;

const int MAIN\_RANK = 0;

double f(double x) {

return x \* x;

}

double simpsonIntegral(double a, double b, int n) {

const double h = (b - a) / n;

double k1 = 0, k2 = 0;

for (int i = 1; i < n; i += 2) {

k1 += f(a + i \* h);

k2 += f(a + (i + 1)\*h);

}

return h / 3 \* (f(a) + 4 \* k1 + 2 \* k2);

}

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

{

double startPoint;

double endPoint;

double result = 0;

int rank;

int size;

int countPoints;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

if (rank == MAIN\_RANK) {

cout << "Enter start point: ";

cin >> startPoint;

cout << "Enter end point: ";

cin >> endPoint;

cout << "Enter count points: ";

cin >> countPoints;

}

if (rank == MAIN\_RANK) {

cout << "\n-------------------------------------------------------" << endl;

clock\_t clockStartSeq = clock();

result = simpsonIntegral(startPoint, endPoint, countPoints);

double timeDifSeq= (double)(clock() - clockStartSeq) / CLOCKS\_PER\_SEC;

cout << "Time difference:" << timeDifSeq << "\n";

cout << "Result:" << result;

cout << "\n-------------------------------------------------------" << endl;

}

MPI\_Bcast(&startPoint, 1, MPI\_DOUBLE, MAIN\_RANK, MPI\_COMM\_WORLD);

MPI\_Bcast(&endPoint, 1, MPI\_DOUBLE, MAIN\_RANK, MPI\_COMM\_WORLD);

MPI\_Bcast(&countPoints, 1, MPI\_INT, MAIN\_RANK, MPI\_COMM\_WORLD);

clock\_t clockStartPar = clock();

double processResult;

double processEndPoint;

int processCountPoint;

double processDifRangePoints = (endPoint - startPoint) / size;

int processCountPoints = countPoints / size;

double processStartPoint = startPoint + processDifRangePoints \* rank;

if (rank == (size - 1)) {

processEndPoint = endPoint;

processCountPoint = countPoints - processCountPoints \* (size-1);

} else {

processEndPoint = startPoint + processDifRangePoints \* (rank + 1);

processCountPoint = processCountPoints;

}

processResult = simpsonIntegral(processStartPoint, processEndPoint, processCountPoint);

double timeDifPar = (double)(clock() - clockStartPar) / CLOCKS\_PER\_SEC;

MPI\_Reduce(&processResult, &result, 1, MPI\_DOUBLE, MPI\_SUM, MAIN\_RANK, MPI\_COMM\_WORLD);

if (rank == MAIN\_RANK) {

cout << "\n-------------------------------------------------------" << endl;

cout << "Time parallel:" << timeDifPar << "\n";

cout << "Result:" << result;

cout << "\n-------------------------------------------------------" << endl;

}

MPI\_Finalize();

if (rank == MAIN\_RANK) {

system("pause");

}

return 0;

}

*CUDA:*

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <iostream>

#include <math.h>

#include <stdio.h>

#include <pplinterface.h>

#include <omp.h>

#define ESP 0.00000001;

#define LEFT\_BORDER 0;

#define RIGTH\_BORDER 50000;

\_\_device\_\_ double funcCuda(double x)

{

return ((sin(x) + 1) / 13) + log(x + 1) + x / 1000;

}

extern \_\_device\_\_ double result;

\_\_global\_\_ void Simpson(double\* inData, double\* outData, double\* left\_border, double\* right\_border)

{

extern \_\_shared\_\_ double sdata[];

unsigned int tid = threadIdx.x;

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

double eps = ESP;

double a = inData[i];

double b = outData[i];

double I = eps + 1, I1 = 0;

outData[i] = 0;

//I-предыдущее вычисленное значение интеграла, I1-новое, с большим N.

int counter = 0;

for (double N = 2; (N <= 4) || (abs(I1 - I) > eps); N \*= 2)

{

double h, sum2 = 0, sum4 = 0, sum = 0;

h = (b - a) / (2 \* N); //Шаг интегрирования.

for (double i = 1; i <= 2 \* N - 1; i += 2)

{

sum4 += funcCuda(a + h \* i); //Значения с нечётными индексами, которые нужно умножить на 4.

sum2 += funcCuda(a + h \* (i + 1)); //Значения с чётными индексами, которые нужно умножить на 2.

counter++;

}

sum = funcCuda(a) + 4 \* sum4 + 2 \* sum2 - funcCuda(b);

//Отнимаем sum = Integral(a) + 4 \* sum4 + 2 \* sum2 - Integral(b);

//Отнимаем sum = Integral(a) + 4 \* sum4 + 2 \* sum2 - Integral(b);

//Отнимаем значение f(b) так как ранее прибавили его дважды.

I = I1;

I1 = (h / 3) \* sum;

}

sdata[tid] = I1;

\_\_syncthreads();

for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1)

{

if (tid < s)

{

sdata[tid] += sdata[tid + s];

}

\_\_syncthreads();

}

if (tid == 0)

{

outData[blockIdx.x] = sdata[tid];

}

\_\_syncthreads();

}

\_\_global\_\_ void Summ(double\* inData, double\* outData)

{

extern \_\_shared\_\_ double sh\_data[];

int tid = threadIdx.x;

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

outData[tid] = inData[tid];

sh\_data[tid] = inData[tid];

\_\_syncthreads();

for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1)

{

if (tid < s)

{

sh\_data[tid] += sh\_data[tid + s];

}

\_\_syncthreads();

}

if (tid == 0)

{

outData[0] = sh\_data[0];

}

}

double funcCon(double x)

{

return ((sin(x) + 1) / 13) + log(x + 1) + x / 1000;

}

double integral(double a, double b) {

double eps = ESP;//Нижний и верхний пределы интегрирования (a, b), погрешность (eps).

double I = eps + 1, I1 = 0;//I-предыдущее вычисленное значение интеграла, I1-новое, с большим N.

for (int N = 2; (N <= 4) || (fabs(I1 - I) > eps); N \*= 2)

{

double h, sum2 = 0, sum4 = 0, sum = 0;

h = (b - a) / (2 \* N);//Шаг интегрирования.

for (int i = 1; i <= 2 \* N - 1; i += 2)

{

sum4 += funcCon(a + h \* i);//Значения с нечётными индексами, которые нужно умножить на 4.

sum2 += funcCon(a + h \* (i + 1));//Значения с чётными индексами, которые нужно умножить на 2.

}

sum = funcCon(a) + 4 \* sum4 + 2 \* sum2 - funcCon(b);//Отнимаем значение f(b) так как ранее прибавили его дважды.

I = I1;

I1 = (h / 3) \* sum;

}

return I1;

}

int main()

{

double left\_border = LEFT\_BORDER;

double right\_border = RIGTH\_BORDER;

double lenght;

double\* inData;

double\* outData;

int number\_of\_threads = 1;

int number\_of\_blocks = 512;

int number\_of\_elem = -1;

double\* dev\_in\_data, \* dev\_out\_data;

double\* device\_left\_border, \* device\_right\_border;

double time1, time2;

double result;

time1 = omp\_get\_wtime();

result = integral(left\_border, right\_border);

time2 = omp\_get\_wtime();

std::cout << "Time spend: " << time2 - time1 << "\n";

std::cout << "Consistent result : " << result << "\n";

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

number\_of\_elem = number\_of\_threads \* number\_of\_blocks;

inData = new double[number\_of\_elem];

outData = new double[number\_of\_elem];

lenght = (right\_border - left\_border) / number\_of\_elem;

time1 = omp\_get\_wtime();

for (int i = 0; i < number\_of\_elem; ++i)

{

inData[i] = left\_border + lenght \* i;

outData[i] = inData[i] + lenght;

}

int size\_double = number\_of\_elem \* sizeof(double);

cudaEvent\_t start\_ev, stop\_ev;

cudaEventCreate(&start\_ev);

cudaEventCreate(&stop\_ev);

cudaMalloc(&dev\_in\_data, size\_double);

cudaMalloc(&dev\_out\_data, size\_double);

cudaMalloc(&device\_left\_border, size\_double);

cudaMalloc(&device\_right\_border, size\_double);

cudaMemcpy(dev\_in\_data, inData, size\_double, cudaMemcpyHostToDevice);

cudaMemcpy(dev\_out\_data, outData, size\_double, cudaMemcpyHostToDevice);

Simpson << <number\_of\_blocks, number\_of\_threads, size\_double / number\_of\_blocks >> >(dev\_in\_data, dev\_out\_data, device\_left\_border, device\_right\_border);

Summ << <1, number\_of\_blocks, number\_of\_blocks \* sizeof(double) >> > (dev\_out\_data, dev\_in\_data);

cudaDeviceSynchronize();

time2 = omp\_get\_wtime();

cudaMemcpy(outData, dev\_in\_data, size\_double, cudaMemcpyDeviceToHost);

std::cout << "Time spend: " << time2 - time1 << "\n";

std::cout << "Cuda result: " << outData[0] << std::endl;

std::cout << "Number of elements = " << number\_of\_elem << std::endl;

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

getchar();

delete inData;

cudaFree(dev\_in\_data);

cudaFree(dev\_out\_data);

cudaFree(device\_right\_border);

cudaFree(device\_left\_border);

return 0;

}