МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение   
высшего образования

«Сибирский государственный университет науки и технологий   
имени академика М.Ф. Решетнева»

Институт информатики и телекоммуникаций

Кафедра информатики и вычислительной техники

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №5

Параллельные вычисления

Алгоритмы обработки массивов

|  |
| --- |
| и их реализация с использованием Intel oneAPI |

Руководитель А. Г. Зотин

подпись, дата инициалы, фамилия

Обучающийся БПИ19-01, 191219044 И.А.Фролов

номер группы, зачетной книжки подпись, дата инициалы, фамилия

Красноярск 2021 г.

Цель работы: Изучение различных видов реализации алгоритмов обработки массивов с использованием возможностей Intel Cilk Plus. Изучение алгоритмов обработки изображения с использованием параллельных реализаций, учитывающих разные подходы к оптимизации обработки (на примере фильтра Гаусса, Медианного фильтра). Разработка алгоритмов вычисления текстурных признаков.

Задания:

**Задание 5.1.** Разработайте консольное приложение, выполняющее следующие действия над матрицами:

–сложение (по элементам);

–перемножение (по элементам);

–вычислением суммы элементов(для двух исходных матриц);

–поиск максимального элемента (для двух исходных матриц).

При реализации использовать возможности OpenMP и Intel Cilk Plus. В частности, для Intel Cilk Plus необходимо выполнить реализацию с использованием cilk\_for, расширенной индексной нотации и их комбинации.

Проведите экспериментальное исследования по обработке данных двух типов (int и double) с разными реализациями алгоритмов (не менее 50 запусков для каждой реализации). При проведении экспериментов используйте матрицы от 3000×3000 до 8000×8000. Выполните исследование для четырех наборов данных в каждом типе. Результаты эксперимента для каждой функции оформить в виде таблицы.

**Задание5.2.** Разработайте консольное приложение, реализующее фильтрации изображения (фильтр Гаусса и медианный). При реализации параллельных вариантов алгоритмов3используйтевозможности OpenMPи Intel Cilk Plus. Для Intel Cilk Plus необходимо использовать cilk\_for, расширенную индексную нотацию и их комбинации. В приложение должен загружаться файл input.bmp и формироваться выходные файлы output\_alg.bmp, где alg означает имя используемого алгоритма обработки (или его номер). Проведите экспериментальное исследование по обработке данных с разными реализациями (не менее 15запусков для каждой реализации) для трех изображений (от 640×480 до 1920×1080) с размерами окрестности Ksize×Ksize. Результаты исследования оформить в виде таблицы.

**Задание5.3.** Разработайте консольное приложение, выполняющее расчет текстурных признаков. Необходимо рассчитать следующие текстурные признаки:

−моменты μ второго порядка;

−относительную гладкость;

−однородность;

−энтропию;

Текстурные признаки рассчитывать для локальной окрестности указываемой пользователем (от 5×5 до 9×9). На вход программе подается изображение (файл input.bmp) и на выходе формируются карты текстурных признаков (выходные файлы output\_ver\_size.bmp, где ver означает имя версии используемой реализации алгоритма, size–размерность локальной окрестности).

**Задание5.4.** Проведите экспериментальное исследование по обработке данных с разными реализациями алгоритмов вычисления текстурных признаков (не менее 10 запусков для каждой реализации) для двух изображений с размерами 1280×720 и 1920×1080. Результаты эксперимента для расчета данных для построения карты текстурных признаков оформить в виде таблицы.

Результаты проделанной работы:

Вычисления проходили на компьютере с ОС Windows 10, с процессором ЦП Intel(R) Core(TM) i5-6400 CPU @ 2.70GHz.

**Задание 1.**

Результаты работы программы приведены в таблице под названием Task1Results.csv. Выполнение задания представлено на рисунке 1.

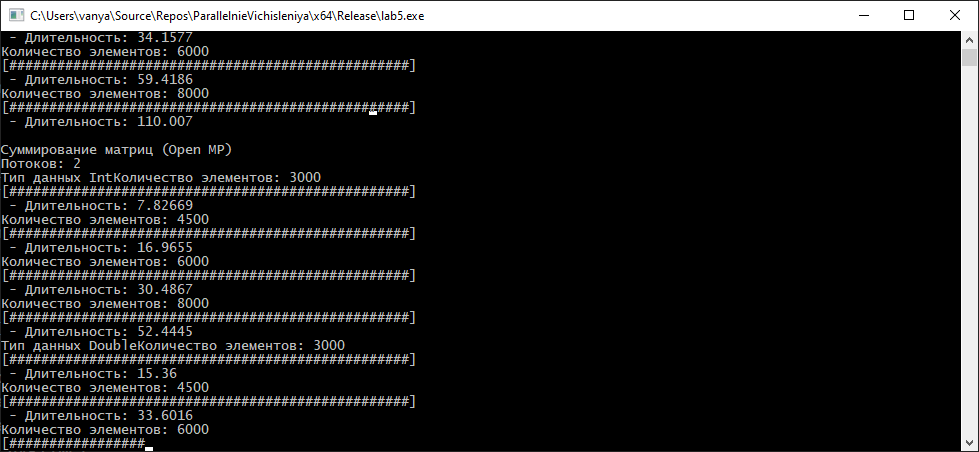


Рисунок 1 – выполнение первого задания

**Задание 2.**

Результаты программы в табличном виде представлены в файле под названием Task2Results.xlsx.

Ниже представлен пример обработки изображения медианным фильтром и фильтром Гаусса.



Рисунок 2 – Исходное изображение размером 1280x720 пикселей



Рисунок 3 – Изображение, обработанное медианным фильтром с размером окрестности в 19 пикселей



Рисунок 4 – Изображение, обработанное фильтром Гаусса с размером окрестности в 19 пикселей

**Задание 3.**

Результаты оаботы программы представлены в таблице под названием Task3Results.xlsx. Ниже представлены примеры вычисления текстурных признаков.



Рисунок 5 – Исходное изображение размером 1920x1080 пикселей

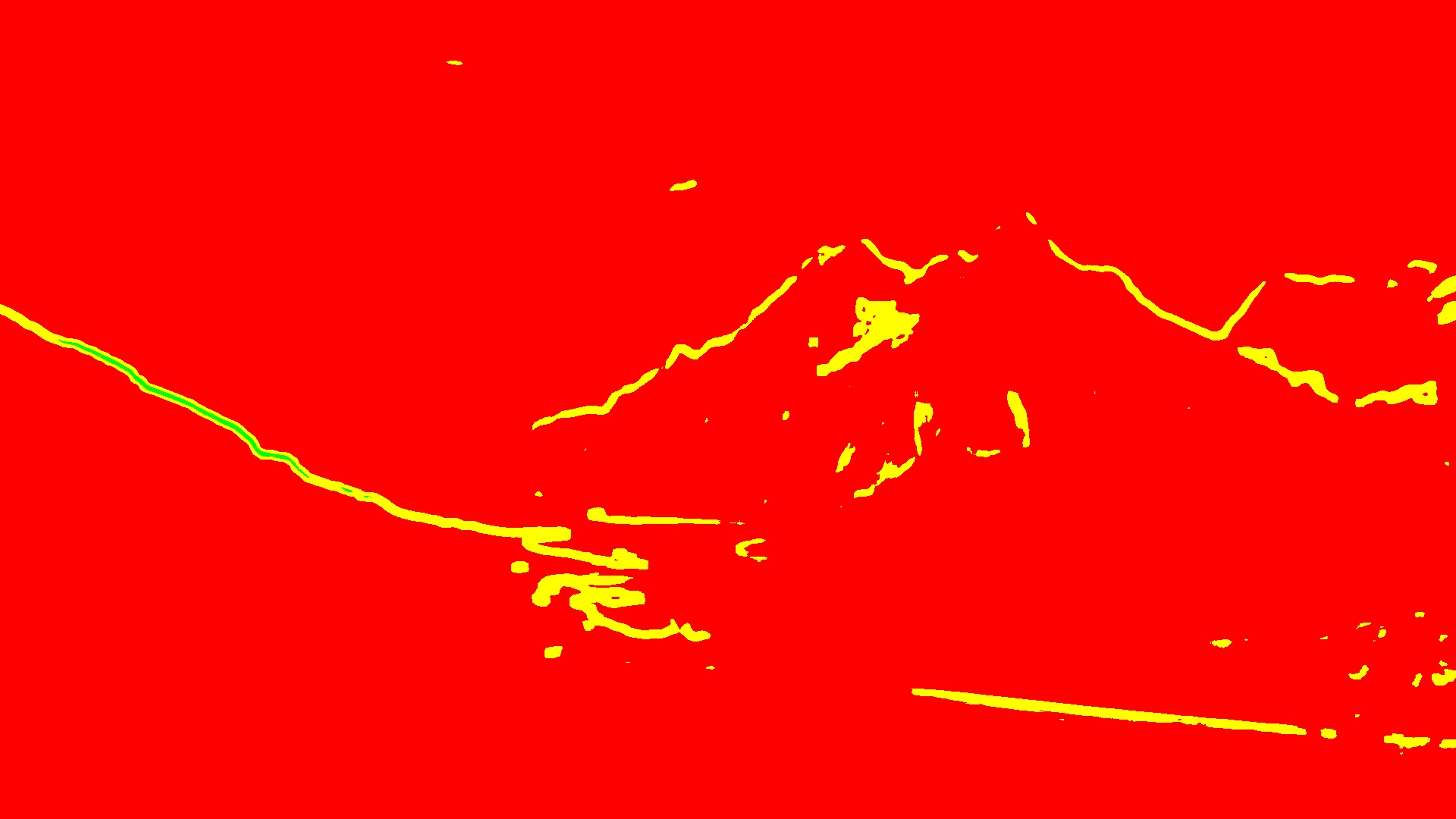


Рисунок 6 – Текстурные признаки по моментам второго порядка (M) на окрестности 7



Рисунок 7 – Текстурные признаки по однородности (U) на окрестности 7



Рисунок 7 – Текстурные признаки по относительной гладкости (R) на окрестности 7

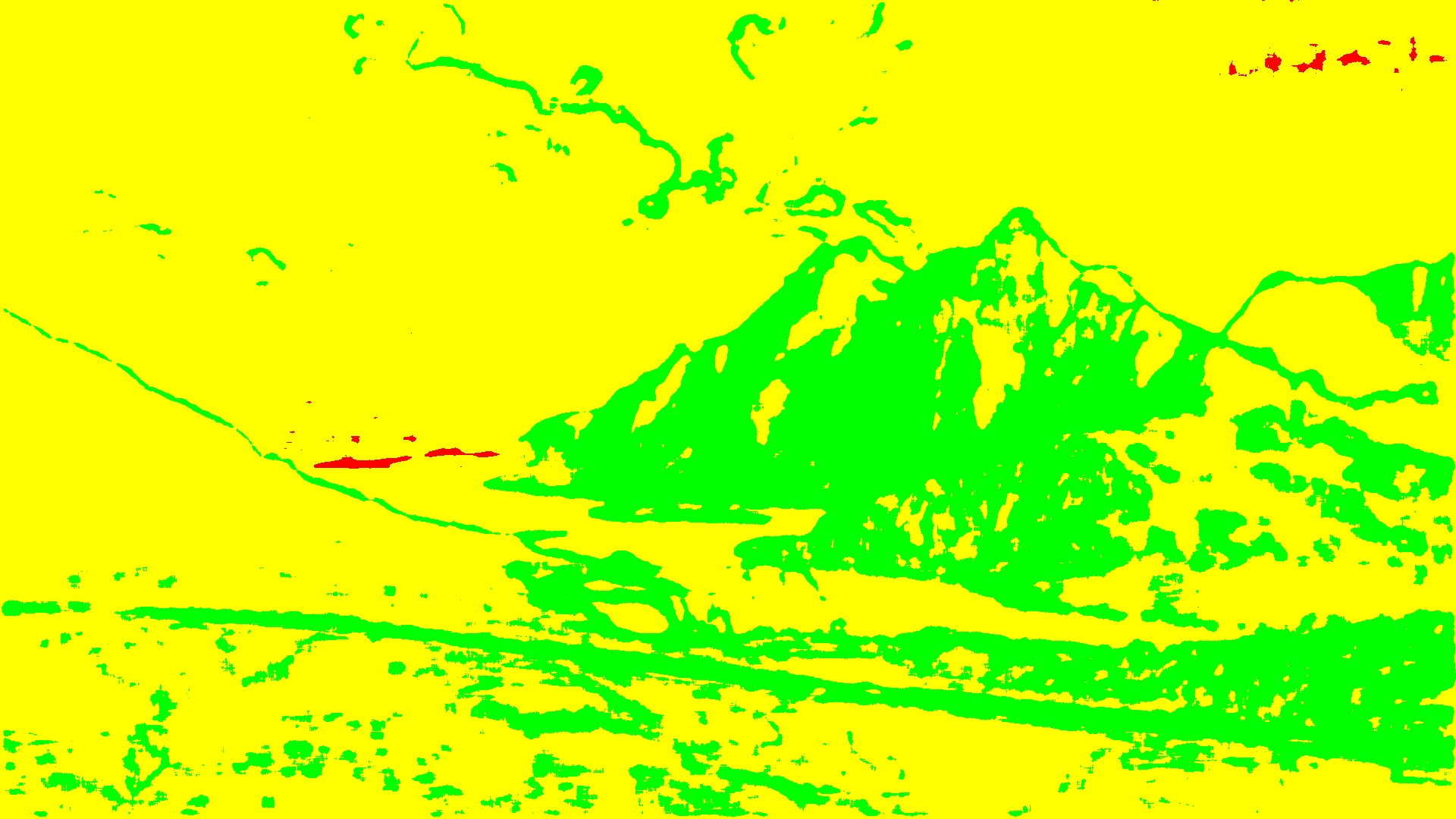


Рисунок 7 – Текстурные признаки по энтропии (E) на окрестности 7

Код программы:

#include<tbb/tbb.h>

#include <omp.h>

#include <stdio.h>

#include <iostream>

#include <String>

#include <ctime>

#include <Windows.h>

#include <Math.h>

#include <fstream>

#include <sstream>

#include "BMPFileRW.h"

# define PI 3.14159265358979323846

using namespace tbb;

using namespace std;

#pragma region Classes

typedef void(\*IntMatrix)(int\*\*&, int\*\*&, int\*\*&, int);

typedef void(\*DoubleMatrix)(double\*\*&, double\*\*&, double\*\*&, int);

typedef void(\*TextureFilterMethod)(RGBQUAD \*\*&image, int height, int width, int rh, int rw, float \*\*&M, float \*\*&U, float \*\*&R, float \*\*&E);

//Обмен значениями

template<class T>

void Swap(T& one, T& two)

{

T temp = one;

one = two;

two = temp;

}

// Класс для reduce для подсчета суммы элементов в 2 матрицах

template<class T>

class reduce\_total {

T\* m1;

T\* m2;

int size;

public:

double sumRes1, sumRes2;

void operator() (blocked\_range<size\_t>& r) {

T sumRes1\_local = sumRes1, sumRes2\_local = sumRes2;

T\* m1\_local = m1;

T\* m2\_local = m2;

size\_t end = r.end();

for (size\_t i = r.begin(); i != end; ++i) {

sumRes1\_local += m1\_local[i];

sumRes2\_local += m2\_local[i];

}

sumRes1 = sumRes1\_local;

sumRes2 = sumRes2\_local;

}

reduce\_total(reduce\_total& r, split) : m1(r.m1), m2(r.m2), size(r.size), sumRes1(0.0), sumRes2(0.0) {} // Split

void join(const reduce\_total& r) { sumRes1 += r.sumRes1; sumRes2 += r.sumRes2; } // Join

reduce\_total(T\* orig\_m1, T\* orig\_m2, int orig\_size) : m1(orig\_m1), m2(orig\_m2), size(orig\_size), sumRes1(0.0), sumRes2(0.0) {} // Основной конструктор

};

//Класс для reduce для нахождения максимального значения в 2 матрицах

template<class T>

class reduce\_max {

private:

T\* ArrayA;

T\* ArrayB;

public:

T MaxValue1, MaxValue2;

void operator() (const blocked\_range<size\_t>& r) {

T\* a = ArrayA;

T\* b = ArrayB;

T val1, val2;

for (size\_t i = r.begin(); i != r.end(); ++i) {

val1 = a[i];

val2 = b[i];

if (MaxValue1 < val1) MaxValue1 = val1;

if (MaxValue2 < val2) MaxValue2 = val2;

}

}

reduce\_max(reduce\_max& x, tbb::split) : ArrayA(x.ArrayA), ArrayB(x.ArrayB), MaxValue1(-1000), MaxValue2(-1000) {}

void join(const reduce\_max& y) {

if (y.MaxValue1 > MaxValue1) MaxValue1 = y.MaxValue1;

if (y.MaxValue2 > MaxValue2) MaxValue2 = y.MaxValue2;

}

reduce\_max(T\* A, T\* B) : ArrayA(A), ArrayB(B), MaxValue1(-1000), MaxValue2(-1000) {}

};

#pragma endregion

#pragma region SumMatrix

//Функция суммирует матрицы

template<class T>

void SumMatrix(T\*\*& matrix1, T\*\*& matrix2, T\*\*& result, int size)

{

//double time\_start = omp\_get\_wtime();

for (int i = 0; i < size; i++)

for (int j = 0; j < size; j++)

{

result[i][j] = matrix1[i][j] + matrix2[i][j];

}

//double time\_stop = omp\_get\_wtime();

//return time\_stop - time\_start;

}

//Функция суммирует матрицы с OMP

template<class T>

void SumMatrixOMP(T\*\*& matrix1, T\*\*& matrix2, T\*\*& result, int size)

{

//double time\_start = omp\_get\_wtime();

#pragma omp parallel for

for (int i = 0; i < size; i++)

for (int j = 0; j < size; j++)

{

result[i][j] = matrix1[i][j] + matrix2[i][j];

}

//double time\_stop = omp\_get\_wtime();

//return time\_stop - time\_start;

}

//Функция суммирует матрицы с TBB

template<class T>

void SumMatrixTBB(T\*\*& matrix1, T\*\*& matrix2, T\*\*& result, int size)

{

//double time\_start = omp\_get\_wtime();

tbb::parallel\_for(tbb::blocked\_range2d<int>(0, size, 0, size), [&](tbb::blocked\_range2d<int> r)

{

for (int i = r.rows().begin(); i < r.rows().end(); i++)

for (int j = r.cols().begin(); j < r.cols().end(); j++)

{

//matrix[i][j] = (cos(i) \* sin(j / 2.0));

result[i][j] = matrix1[i][j] + matrix2[i][j];

}

});

//double time\_stop = omp\_get\_wtime();

//return time\_stop - time\_start;

}

#pragma endregion

#pragma region ProductMatrix

//Функция вычисляет произведение матриц

template<class T>

void ProductMatrix(T\*\*& matrix1, T\*\*& matrix2, T\*\*& result, int size)

{

//double time\_start = omp\_get\_wtime();

for (int i = 0; i < size; i++)

for (int j = 0; j < size; j++)

{

//matrix[i][j] = (cos(i) \* sin(j / 2.0));

result[i][j] = matrix1[i][j] \* matrix2[i][j];

}

//double time\_stop = omp\_get\_wtime();

//return time\_stop - time\_start;

}

//Функция вычисляет произведение матриц с OMP

template<class T>

void ProductMatrixOMP(T\*\*& matrix1, T\*\*& matrix2, T\*\*& result, int size)

{

//double time\_start = omp\_get\_wtime();

#pragma omp parallel for

for (int i = 0; i < size; i++)

for (int j = 0; j < size; j++)

{

//matrix[i][j] = (cos(i) \* sin(j / 2.0));

result[i][j] = matrix1[i][j] \* matrix2[i][j];

}

//double time\_stop = omp\_get\_wtime();

//return time\_stop - time\_start;

}

//Функция вычисляет произведение матриц с TBB

template<class T>

void ProductMatrixTBB(T\*\*& matrix1, T\*\*& matrix2, T\*\*& result, int size)

{

//double time\_start = omp\_get\_wtime();

tbb::parallel\_for(tbb::blocked\_range2d<int>(0, size, 0, size), [&](tbb::blocked\_range2d<int> r)

{

for (int i = r.rows().begin(); i < r.rows().end(); i++)

for (int j = r.cols().begin(); j < r.cols().end(); j++)

{

//matrix[i][j] = (cos(i) \* sin(j / 2.0));

result[i][j] = matrix1[i][j] \* matrix2[i][j];

}

});

//double time\_stop = omp\_get\_wtime();

//return time\_stop - time\_start;

}

#pragma endregion

#pragma region TotalSumMatrix

//Функция суммирует элементы матриц

template<class T>

void TotalSum(T\*\*& matrix1, T\*\*& matrix2, T\*\*& unused, int size) {

//double time\_start = omp\_get\_wtime();

T total1 = 0, total2 = 0;

for (int j = 0; j < size; j++)

{

for (int i = 0; i < size; i++)

{

total1 += matrix1[i][j];

total2 += matrix2[i][j];

}

}

cout << total1 << " " << total2;

//double time\_stop = omp\_get\_wtime();

//return time\_stop - time\_start;

}

//Функция суммирует элементы матриц с OMP reduction

template<class T>

void TotalSumOMP(T\*\*& matrix1, T\*\*& matrix2, T\*\*& unused, int size) {

//double time\_start = omp\_get\_wtime();

T total1 = 0, total2 = 0;

for (int j = 0; j < size; j++)

{

#pragma omp parallel

#pragma omp for reduction(+:total1)

for (int i = 0; i < size; i++)

{

total1 += matrix1[i][j];

}

#pragma omp parallel

#pragma omp for reduction(+:total2)

for (int i = 0; i < size; i++)

{

total2 += matrix2[i][j];

}

}

cout << total1 << " " << total2;

//double time\_stop = omp\_get\_wtime();

//return time\_stop - time\_start;

}

//Функция суммирует элементы матриц с TBB

template<class T>

void TotalSumTBB(T\*\*& matrix1, T\*\*& matrix2, T\*\*& unused, int size) {

//double time\_start = omp\_get\_wtime();

T total1 = 0, total2 = 0;

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

reduce\_total<T> r(matrix1[i], matrix2[i], size);

parallel\_reduce(blocked\_range<size\_t>(0, size), r);

total1 += r.sumRes1;

total2 += r.sumRes2;

}

cout << total1 << " " << total2;

//double time\_stop = omp\_get\_wtime();

//return time\_stop - time\_start;

}

#pragma endregion

#pragma region MaxElemMatrix

//Функция находит максимальный элемент в 2 матрицах с OMP for

template<class T>

void MaxElem(T\*\*& matrix1, T\*\*& matrix2, T\*\*& unused, int size) {

//double time\_start = omp\_get\_wtime();

double max1 = -1000, max2 = -1000;

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

for (int j = 0; j < size; j++) {

if (max1 < matrix1[i][j]) max1 = matrix1[i][j];

if (max2 < matrix1[i][j]) max2 = matrix1[i][j];

}

}

cout << max1 << " " << max2;

//double time\_stop = omp\_get\_wtime();

//return time\_stop - time\_start;

}

//Функция находит максимальный элемент в 2 матрицах с OMP for

template<class T>

void MaxElemOMP(T\*\*& matrix1, T\*\*& matrix2, T\*\*& unused, int size) {

//double time\_start = omp\_get\_wtime();

double max1 = -1000, max2 = -1000;

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

int j = 0;

#pragma omp parallel for private(j)

for (j = 0; j < size; j++) {

if (max1 < matrix1[i][j]) max1 = matrix1[i][j];

if (max2 < matrix1[i][j]) max2 = matrix1[i][j];

}

}

cout << max1 << " " << max2;

//double time\_stop = omp\_get\_wtime();

//return time\_stop - time\_start;

}

//Функция находит максимальный элемент в 2 матрицах с TBB

template<class T>

void MaxElemTBB(T\*\*& matrix1, T\*\*& matrix2, T\*\*& unused, int size) {

//double time\_start = omp\_get\_wtime();

double max1 = -1000, max2 = -1000;

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

reduce\_max<T> r(matrix1[i], matrix2[i]);

parallel\_reduce(blocked\_range<size\_t>(0, size), r);

if (max1 < r.MaxValue1) max1 = r.MaxValue1;

if (max2 < r.MaxValue2) max2 = r.MaxValue2;

}

cout << max1 << " " << max2;

//double time\_stop = omp\_get\_wtime();

//return time\_stop - time\_start;

}

#pragma endregion

#pragma region specFunc

//Заполнение матрицы

template<class T>

void Fill2DMatrix(T\*\* matrix, int size)

{

tbb::parallel\_for(tbb::blocked\_range2d<int>(0, size, 0, size), [&](tbb::blocked\_range2d<int> r)

{

for (int i = r.rows().begin(); i < r.rows().end(); i++)

for (int j = r.cols().begin(); j < r.cols().end(); j++)

{

//matrix[i][j] = (cos(i) \* sin(j / 2.0));

matrix[i][j] = -1000 \* pow(i + 1, 3 / 4) \* cos(i) / atan(i + 1) + 1000 \* pow(j + 1, 3 / 4)\* cos(j) / atan(j + 1);

}

});

}

//Вывести матрицы

template<class T>

void Print2DMatrix(T\*\* matrix, int size)

{

for (int i = 0; i < size; i++)

{

for (int j = 0; j < size; j++)

{

//matrix[i][j] = (cos(i) \* sin(j / 2.0));

cout << matrix[i][j] << " ";

}

cout << endl;

}

}

double AvgTrustedInterval(double& avg, double\*& times, int& cnt)

{

double sd = 0, newAVg = 0;

int newCnt = 0;

for (int i = 0; i < cnt; i++)

{

sd += (times[i] - avg) \* (times[i] - avg);

}

sd /= (cnt - 1.0);

sd = sqrt(sd);

for (int i = 0; i < cnt; i++)

{

if (avg - sd <= times[i] && times[i] <= avg + sd)

{

newAVg += times[i];

newCnt++;

}

}

if (newCnt == 0) newCnt = 1;

return newAVg / newCnt;

}

#pragma endregion

#pragma region Task1\_CalcMatrixFuncsAvgTime

void CalcMatrixAvgTimeInt(IntMatrix func, int size, double& time, int iterations)

{

int\*\* matrix1 = new int\*[size];

int\*\* matrix2 = new int\*[size];

int\*\* result = new int\*[size];

for (int i = 0; i < size; i++)

{

matrix1[i] = new int[size];

matrix2[i] = new int[size];

result[i] = new int[size];

}

Fill2DMatrix<int>(matrix1, size);

Fill2DMatrix<int>(matrix2, size);

double avgTime = 0, avgTimeT = 0, correctAVG = 0;

double startTime, curTime;

double\* Times = new double[iterations];

std::cout << "[";

for (int i = 0; i < iterations; i++)

{

startTime = omp\_get\_wtime();

func(matrix1, matrix2, result, size);

curTime = omp\_get\_wtime() - startTime;

Times[i] = curTime;

avgTime += curTime;

std::cout << "#";

}

std::cout << "]\n";

avgTime /= iterations;

avgTimeT = AvgTrustedInterval(avgTime, Times, iterations);

time = avgTimeT \* 1000;

for (int i = 0; i < size; i++)

{

delete[] matrix1[i];

delete[] matrix2[i];

delete[] result[i];

}

delete matrix1;

delete matrix2;

delete result;

}

void CalcMatrixAvgTimeDouble(DoubleMatrix func, int size, double& time, int iterations)

{

double\*\* matrix1 = new double\*[size];

double\*\* matrix2 = new double\*[size];

double\*\* result = new double\*[size];

for (int i = 0; i < size; i++)

{

matrix1[i] = new double[size];

matrix2[i] = new double[size];

result[i] = new double[size];

}

Fill2DMatrix<double>(matrix1, size);

Fill2DMatrix<double>(matrix2, size);

double avgTime = 0, avgTimeT = 0, correctAVG = 0;

double startTime, curTime;

double\* Times = new double[iterations];

std::cout << "[";

for (int i = 0; i < iterations; i++)

{

startTime = omp\_get\_wtime();

func(matrix1, matrix2, result, size);

curTime = omp\_get\_wtime() - startTime;

Times[i] = curTime;

avgTime += curTime;

std::cout << "#";

}

std::cout << "]\n";

avgTime /= iterations;

avgTimeT = AvgTrustedInterval(avgTime, Times, iterations);

time = avgTimeT \* 1000;

for (int i = 0; i < size; i++)

{

delete[] matrix1[i];

delete[] matrix2[i];

delete[] result[i];

}

delete matrix1;

delete matrix2;

delete result;

}

void CalcMatrixFuncs()

{

std::ofstream resultsFile;

int\* dataAmount = new int[4]{ 3000, 4500, 6000, 8000 };

double time;

string\* funcsNames = new string[12]{ "Суммирование матриц (последовательно)", "Суммирование матриц (Open MP)", "Суммирование матриц (TBB)", "Перемножение матриц (последовательно)", "Перемножение матриц (Open MP)", "Перемножение матриц (TBB)", "Вычисление суммы элементов 2 матриц (последовательно)", "Вычисление суммы элементов 2 матриц (Open MP)", "Вычисление суммы элементов 2 матриц (TBB)", "Поиск максимального элемента в 2 матрицах (последовательно)", "Поиск максимального элемента в 2 матрицах (Open MP)", "Поиск максимального элемента в 2 матрицах (TBB)" };

//IntMatrix\* funcInt = new IntMatrix[3]{ SumMatrix<int>, SumMatrixOMP<int>, SumMatrixTBB<int> };

DoubleMatrix\* funcDouble = new DoubleMatrix[12]{ SumMatrix<double>, SumMatrixOMP<double>, SumMatrixTBB<double>, ProductMatrix<double>, ProductMatrixOMP<double>, ProductMatrixTBB<double>, TotalSum<double>, TotalSumOMP<double>, TotalSumTBB<double>, MaxElem<double>, MaxElemOMP<double>, MaxElemTBB<double> };

IntMatrix\* funcInt = new IntMatrix[12]{ SumMatrix<int>, SumMatrixOMP<int>, SumMatrixTBB<int>, ProductMatrix<int>, ProductMatrixOMP<int>, ProductMatrixTBB<int>, TotalSum<int>, TotalSumOMP<int>, TotalSumTBB<int>, MaxElem<int>, MaxElemOMP<int>, MaxElemTBB<int> };

double T1[3][2];

resultsFile.open("Task1Results.csv", std::ios\_base::app);

resultsFile << "Int;;;;;;;;;;Double;\n";

resultsFile << "Функция сортировки;Потоки;Время;Sp(n);Время;Sp(n);Время;Sp(n);Время;Sp(n);Время;Sp(n);Время;Sp(n);Время;Sp(n);Время;Sp(n);\n";

resultsFile << ";;";

for (int cr = 0; cr < 4; cr++)

resultsFile << "НД" << cr + 1 << ": " << dataAmount[cr] << ";;";

for (int cr = 0; cr < 4; cr++)

resultsFile << "НД" << cr + 1 << ": " << dataAmount[cr] << ";;";

resultsFile << endl;

for (int i = 0; i < 12; i++)

{

resultsFile << funcsNames[i];

std::cout << funcsNames[i] << endl;

for (int t = 2; t < 5; t++)

{

if (i == 0 || i == 3 || i == 6 || i == 9)

t = 1;

std::cout << "Потоков: " << t << endl;

resultsFile << ";" << t << ";";

omp\_set\_num\_threads(t);

global\_control global\_limit(global\_control::max\_allowed\_parallelism, t);

for (int j = 0; j < 2; j++)

{

if (j == 0)

{

cout << "Тип данных Int";

}

else

{

cout << "Тип данных Double";

}

for (int d = 0; d < 4; d++)

{

std::cout << "Количество элементов: " << dataAmount[d] << endl;

if (j == 0)

CalcMatrixAvgTimeInt(funcInt[i], dataAmount[d], time, 50);

else

CalcMatrixAvgTimeDouble(funcDouble[i], dataAmount[d], time, 50);

if (i == 0 || i == 3 || i == 6 || i == 9)

T1[d][j] = time;

resultsFile << time << ";" << T1[d][j] / time << ";";

std::cout << " - Длительность: " << time << endl;

}

}

std::cout << endl;

resultsFile << endl;

if (i == 0 || i == 3 || i == 6 || i == 9)

break;

}

}

resultsFile << endl;

}

#pragma endregion

#pragma region MedianFiltering

//Сортировка Шелла

template<class T>

void ShellSortConsistently(T \*arr, int length)

{

int step, i, j;

for (step = length / 2; step > 0; step /= 2)

{

// Перечисление элементов, которые сортируются на определённом шаге

for (i = step; i < length; i++)

{

// Перестановка элементов внутри подсписка, пока i-тый не будет отсортирован

for (j = i - step; j >= 0 && arr[j] > arr[j + step]; j -= step)

{

Swap(arr[j], arr[j + step]);

}

}

}

}

//заполнение медиального массива

RGBQUAD\* getMedial(RGBQUAD \*\*&image, int width, int height, int x, int y, int kSize)

{

int index = 0;

RGBQUAD\* barray = new RGBQUAD[(2 \* kSize + 1) \* (2 \* kSize + 1)];

int coordX;

int coordY;

for (int dy = -kSize; dy <= kSize; dy++)

{

coordY = y + dy;

for (int dx = -kSize; dx <= kSize; dx++)

{

coordX = x + dx;

if (coordX < 0)

coordX = 0;

if (coordX >= width)

coordX = width - 1;

if (coordY < 0)

coordY = 0;

if (coordY >= height)

coordY = height - 1;

barray[index] = image[coordY][coordX];

index++;

}

}

return barray;

}

//Сортировка массива РГБ

RGBQUAD\* sortRGB(RGBQUAD\* arr, long length)

{

BYTE \*red = new BYTE[length];

BYTE \*green = new BYTE[length];

BYTE \*blue = new BYTE[length];

for (int i = 0; i < length; i++)

{

red[i] = arr[i].rgbRed;

green[i] = arr[i].rgbGreen;

blue[i] = arr[i].rgbBlue;

}

ShellSortConsistently(red, length);

ShellSortConsistently(green, length);

ShellSortConsistently(blue, length);

RGBQUAD\* resultRGBArr = new RGBQUAD[length];

for (int i = 0; i < length; i++)

resultRGBArr[i] = { blue[i], green[i],red[i], 0 };

delete[] red;

delete[] green;

delete[] blue;

return resultRGBArr;

}

//сортировка массива РГБ с Omp sections и for

RGBQUAD\* sortRGB\_OMP(RGBQUAD\* arr, long length)

{

BYTE \*red = new BYTE[length];

BYTE \*green = new BYTE[length];

BYTE \*blue = new BYTE[length];

#pragma omp parallel for

for (int i = 0; i < length; i++)

{

red[i] = arr[i].rgbRed;

green[i] = arr[i].rgbGreen;

blue[i] = arr[i].rgbBlue;

}

#pragma omp parallel sections

{

#pragma omp section

{

ShellSortConsistently(red, length);

}

#pragma omp section

{

ShellSortConsistently(green, length);

}

#pragma omp section

{

ShellSortConsistently(blue, length);

}

}

RGBQUAD\* narr = new RGBQUAD[length];

for (int i = 0; i < length; i++)

narr[i] = { blue[i], green[i],red[i], 0 };

delete[] red;

delete[] green;

delete[] blue;

return narr;

}

//сортировка массива РГБ с Omp sections и for

RGBQUAD\* sortRGB\_TBB(RGBQUAD\* arr, long length)

{

BYTE \*red = new BYTE[length];

BYTE \*green = new BYTE[length];

BYTE \*blue = new BYTE[length];

tbb::parallel\_for(tbb::blocked\_range<int>(0, length), [&](tbb::blocked\_range<int> r)

{

for (int i = r.begin(); i < r.end(); i++)

{

red[i] = arr[i].rgbRed;

green[i] = arr[i].rgbGreen;

blue[i] = arr[i].rgbBlue;

}

});

tbb::task\_group g;

g.run([&] {ShellSortConsistently(red, length); });

g.run([&] {ShellSortConsistently(blue, length); });

g.run([&] {ShellSortConsistently(green, length); });

g.wait();

RGBQUAD\* narr = new RGBQUAD[length];

for (int i = 0; i < length; i++)

narr[i] = { blue[i], green[i],red[i], 0 };

delete[] red;

delete[] green;

delete[] blue;

return narr;

}

//медианная фильтрация

void MedianFiltering(RGBQUAD\*\* &RGB, int height, int width, int kSize, RGBQUAD\*\* &RGBresult)

{

RGBQUAD \*temp1, \*temp2;

int size = (2 \* kSize + 1) \* (2 \* kSize + 1);

for (int y = 0; y < height; y++)

{

for (int x = 0; x < width; x++)

{

//в окне H x W кладу пиксели в массив temp

temp1 = getMedial(RGB, width, height, x, y, kSize); //заполняю медиальный массив

temp2 = sortRGB(temp1, size); // сортирую каждую из компонент

RGBresult[y][x] = temp2[size / 2]; // вытаскиваю срединный элемент

delete[] temp1;

delete[] temp2;

}

}

}

//медианная фильтрация (OMP)

void MedianFilteringOMP(RGBQUAD\*\* &RGB, int height, int width, int kSize, RGBQUAD\*\* &RGBresult)

{

int size = (2 \* kSize + 1) \* (2 \* kSize + 1);

#pragma omp parallel for

for (int y = 0; y < height; y++)

{

RGBQUAD \*temp1, \*temp2;

for (int x = 0; x < width; x++)

{

//в окне H x W ложу пиксели в массив temp

temp1 = getMedial(RGB, width, height, x, y, kSize); //заполняю медиальный массив

temp2 = sortRGB\_OMP(temp1, size);

RGBresult[y][x] = temp2[size / 2]; // вытаскиваю срединный элемент

delete[] temp1;

delete[] temp2;

}

}

}

//медианная фильтрация (TBB)

void MedianFilteringTBB(RGBQUAD\*\* &RGB, int height, int width, int kSize, RGBQUAD\*\* &RGBresult)

{

int size = (2 \* kSize + 1) \* (2 \* kSize + 1);

tbb::parallel\_for(tbb::blocked\_range2d<int>(0, height, 0, width), [&](tbb::blocked\_range2d<int> r)

{

for (int y = r.rows().begin(); y < r.rows().end(); y++)

{

RGBQUAD \*temp1, \*temp2;

for (int x = r.cols().begin(); x < r.cols().end(); x++)

{

//в окне H x W ложу пиксели в массив temp

temp1 = getMedial(RGB, width, height, x, y, kSize); //заполняю медиальный массив

temp2 = sortRGB\_TBB(temp1, size);

RGBresult[y][x] = temp2[size / 2]; // вытаскиваю срединный элемент

delete[] temp1;

delete[] temp2;

}

}

});

}

#pragma endregion

#pragma region GaussFiltering

//Формирование матрицы коэффициентов для фильтрации Гаусса

double\*\* GaussMatrixCoefficients(int kSize, double q) {

double\*\* Result = new double\*[kSize \* 2 + 1];

for (int i = 0; i < kSize \* 2 + 1; i++)

Result[i] = new double[kSize \* 2 + 1];

double SUM = 0;

for (int Y = -kSize; Y <= kSize; Y++)

for (int X = -kSize; X <= kSize; X++) {

double CF = (1 / (2 \* PI \* q \* q)) \* exp(-1 \* (X \* X + Y \* Y) / (2 \* q \* q));

Result[Y + kSize][X + kSize] = CF;

SUM += CF;

}

for (int Y = -kSize; Y <= kSize; Y++)

for (int X = -kSize; X <= kSize; X++)

Result[Y + kSize][X + kSize] /= SUM;

return Result;

}

//Формирование матрицы коэффициентов для фильтрации Гаусса (Open MP)

double\*\* GaussMatrixCoefficientsOMP(int kSize, double q) {

double\*\* Result = new double\*[kSize \* 2 + 1];

for (int i = 0; i < kSize \* 2 + 1; i++)

Result[i] = new double[kSize \* 2 + 1];

double SUM = 0;

#pragma omp parallel for reduction(+:SUM) shared(Result)

for (int Y = -kSize; Y <= kSize; Y++)

for (int X = -kSize; X <= kSize; X++) {

double CF = (1 / (2 \* PI \* q \* q)) \* exp(-1 \* (X \* X + Y \* Y) / (2 \* q \* q));

Result[Y + kSize][X + kSize] = CF;

SUM += CF;

}

#pragma omp parallel for shared(Result)

for (int Y = -kSize; Y <= kSize; Y++)

for (int X = -kSize; X <= kSize; X++) {

Result[Y + kSize][X + kSize] /= SUM;

}

return Result;

}

//Линейный фильтр Гаусса последовательный;

//Возвращает RGBresult указывающий на выходную картинку

void LineGaussFiltering(RGBQUAD\*\* &RGB, int height, int width, int kSize, RGBQUAD\*\* &RGBresult)

{

double\*\* CoefMatrix = GaussMatrixCoefficients(kSize, kSize / 3.0); //Сигма тут

for (int Y = 0; Y < height; Y++)

{

for (int X = 0; X < width; X++)

{

double rgbBlue = 0, rgbGreen = 0, rgbRed = 0;

for (int DY = -kSize; DY <= kSize; DY++)

{

int KY = Y + DY;

if (KY < 0)

KY = 0;

if (KY > height - 1)

KY = height - 1;

for (int DX = -kSize; DX <= kSize; DX++)

{

int KX = X + DX;

if (KX < 0)

KX = 0;

if (KX > width - 1)

KX = width - 1;

double tmp = CoefMatrix[DY + kSize][DX + kSize];

rgbBlue += RGB[KY][KX].rgbBlue \* tmp;

rgbGreen += RGB[KY][KX].rgbGreen \* tmp;

rgbRed += RGB[KY][KX].rgbRed \* tmp;

}

}

if (rgbBlue < 0) rgbBlue = 0;

if (rgbBlue > 255) rgbBlue = 255;

if (rgbGreen < 0) rgbGreen = 0;

if (rgbGreen > 255) rgbGreen = 255;

if (rgbRed < 0) rgbRed = 0;

if (rgbRed > 255) rgbRed = 255;

RGBresult[Y][X].rgbBlue = rgbBlue;

RGBresult[Y][X].rgbGreen = rgbGreen;

RGBresult[Y][X].rgbRed = rgbRed;

}

}

for (int i = 0; i < kSize; i++)

delete[] CoefMatrix[i];

delete[] CoefMatrix;

}

//Линейный фильтр Гаусса (Open MP);

//Возвращает RGBresult указывающий на выходную картинку

void LineGaussFilteringOMP(RGBQUAD\*\* &RGB, int height, int width, int kSize, RGBQUAD\*\* &RGBresult)

{

double\*\* CoefMatrix = GaussMatrixCoefficientsOMP(kSize, kSize / 3.0); //Сигма тут

#pragma omp parallel for

for (int Y = 0; Y < height; Y++)

{

for (int X = 0; X < width; X++)

{

double rgbBlue = 0, rgbGreen = 0, rgbRed = 0;

for (int DY = -kSize; DY <= kSize; DY++)

{

int KY = Y + DY;

if (KY < 0)

KY = 0;

if (KY > height - 1)

KY = height - 1;

for (int DX = -kSize; DX <= kSize; DX++)

{

int KX = X + DX;

if (KX < 0)

KX = 0;

if (KX > width - 1)

KX = width - 1;

double tmp = CoefMatrix[DY + kSize][DX + kSize];

rgbBlue += RGB[KY][KX].rgbBlue \* tmp;

rgbGreen += RGB[KY][KX].rgbGreen \* tmp;

rgbRed += RGB[KY][KX].rgbRed \* tmp;

}

}

if (rgbBlue < 0) rgbBlue = 0;

if (rgbBlue > 255) rgbBlue = 255;

if (rgbGreen < 0) rgbGreen = 0;

if (rgbGreen > 255) rgbGreen = 255;

if (rgbRed < 0) rgbRed = 0;

if (rgbRed > 255) rgbRed = 255;

RGBresult[Y][X].rgbBlue = rgbBlue;

RGBresult[Y][X].rgbGreen = rgbGreen;

RGBresult[Y][X].rgbRed = rgbRed;

}

}

for (int i = 0; i < kSize; i++)

delete[] CoefMatrix[i];

delete[] CoefMatrix;

}

//Линейный фильтр Гаусса (TBB);

//Возвращает RGBresult указывающий на выходную картинку

void LineGaussFilteringTBB(RGBQUAD\*\* &RGB, int height, int width, int kSize, RGBQUAD\*\* &RGBresult)

{

// Заполнение матрицы коэффициентов

double\*\* CoefMatrix;

tbb::task\_group g;

g.run([&] { CoefMatrix = GaussMatrixCoefficients(kSize, kSize / 3.0); }); //Сигма тут

g.wait();

tbb::parallel\_for(tbb::blocked\_range2d<int>(0, height, 0, width), [&](tbb::blocked\_range2d<int> r)

{

for (int Y = r.rows().begin(); Y < r.rows().end(); Y++)

{

for (int X = r.cols().begin(); X < r.cols().end(); X++)

{

double rgbBlue = 0, rgbGreen = 0, rgbRed = 0;

for (int DY = -kSize; DY <= kSize; DY++)

{

int KY = Y + DY;

if (KY < 0)

KY = 0;

if (KY > height - 1)

KY = height - 1;

for (int DX = -kSize; DX <= kSize; DX++)

{

int KX = X + DX;

if (KX < 0)

KX = 0;

if (KX > width - 1)

KX = width - 1;

double tmp = CoefMatrix[DY + kSize][DX + kSize];

rgbBlue += RGB[KY][KX].rgbBlue \* tmp;

rgbGreen += RGB[KY][KX].rgbGreen \* tmp;

rgbRed += RGB[KY][KX].rgbRed \* tmp;

}

}

if (rgbBlue < 0) rgbBlue = 0;

if (rgbBlue > 255) rgbBlue = 255;

if (rgbGreen < 0) rgbGreen = 0;

if (rgbGreen > 255) rgbGreen = 255;

if (rgbRed < 0) rgbRed = 0;

if (rgbRed > 255) rgbRed = 255;

RGBresult[Y][X].rgbBlue = rgbBlue;

RGBresult[Y][X].rgbGreen = rgbGreen;

RGBresult[Y][X].rgbRed = rgbRed;

}

}

});

for (int i = 0; i < kSize; i++)

delete[] CoefMatrix[i];

delete[] CoefMatrix;

}

#pragma endregion

#pragma region Task2\_CalcFilteringMethods

void FilteringFuncAverageTime(int filtheringMethodIndex, string fileName, string methodFullName, int kSize, double& time, int iterations)

{

double avgTime = 0, avgTimeT = 0, correctAVG = 0;

double startTime, curTime;

double\* Times = new double[iterations];

std::cout << "[";

for (int i = 0; i < iterations; i++)

{

RGBQUAD\*\* sourceImage;

RGBQUAD\*\* resultImage;

BITMAPFILEHEADER head;

BITMAPINFOHEADER info;

BMPRead(sourceImage, head, info, fileName.c\_str());

resultImage = new RGBQUAD\*[info.biHeight];

for (int i = 0; i < info.biHeight; i++)

resultImage[i] = new RGBQUAD[info.biWidth];

startTime = omp\_get\_wtime();

switch (filtheringMethodIndex)

{

case 0:

{

MedianFiltering(sourceImage, info.biHeight, info.biWidth, kSize, resultImage);

break;

}

case 1:

{

MedianFilteringOMP(sourceImage, info.biHeight, info.biWidth, kSize, resultImage);

break;

}

case 2:

{

MedianFilteringTBB(sourceImage, info.biHeight, info.biWidth, kSize, resultImage);

break;

}

case 3:

{

LineGaussFiltering(sourceImage, info.biHeight, info.biWidth, kSize, resultImage);

break;

}

case 4:

{

LineGaussFilteringOMP(sourceImage, info.biHeight, info.biWidth, kSize, resultImage);

break;

}

default:

{

LineGaussFilteringTBB(sourceImage, info.biHeight, info.biWidth, kSize, resultImage);

break;

}

}

curTime = omp\_get\_wtime() - startTime;

Times[i] = curTime;

avgTime += curTime;

std::cout << "#";

if (i == 0)

{

BMPWrite(resultImage, head, info, methodFullName.c\_str());

}

for (int i = 0; i < info.biHeight; i++)

{

delete[] sourceImage[i];

}

delete[] sourceImage;

for (int i = 0; i < info.biHeight; i++)

{

delete[] resultImage[i];

}

delete[] resultImage;

}

std::cout << "]\n";

avgTime /= iterations;

avgTimeT = AvgTrustedInterval(avgTime, Times, iterations);

time = avgTimeT;

}

void TaskFilteringMethods()

{

std::ofstream resultsFile;

string\* filteringFuncsNames = new string[6]{ "Медианный фильтр (последовательный)", "Медианный фильтр (Open MP)", "Медианный фильтр (TBB)", "Фильтр Гаусса (последовательный)", "Фильтр Гаусса (Open MP)", "Фильтр Гаусса(TBB)" };

string\* inputFiles = new string[4]{ "500x500.bmp", "840x480.bmp", "1280x720.bmp", "1920x1080.bmp" };

int\* kSize = new int[3]{ 3, 6, 9 };

int iterations = 20;

stringstream ss;

double time;

double\*\* T1 = new double\*[3];

for (int i = 0; i < 3; i++)

{

T1[i] = new double[4];

}

resultsFile.open("Task2Results.csv", std::ios\_base::app);

resultsFile << "Функция;Потоки;Ksize;НД1;;НД2;;НД3;;НД4\n";

resultsFile << ";;;Время;Sp(n);Время;Sp(n);Время;Sp(n);Время;Sp(n);\n";

for (int j = 0; j < 6; j++)

{

resultsFile << filteringFuncsNames[j] << ";";

cout << "\_\_\_" << filteringFuncsNames[j] << "\_\_\_" << endl;

for (int t = 2; t < 5; t++)

{

if (j == 0 || j == 3)

t = 1;

omp\_set\_num\_threads(t);

global\_control global\_limit(global\_control::max\_allowed\_parallelism, t);

if (t > 2)

resultsFile << ";";

resultsFile << t << ";";

std::cout << "Потоков: " << t << endl;

for (int k = 0; k < 3; k++)

{

if (k != 0)

resultsFile << ";;";

resultsFile << kSize[k] \* 2 + 1 << ";";

std::cout << "Ksize = " << kSize[k] \* 2 + 1 << endl;

for (int d = 0; d < 4; d++)

{

ss = stringstream();

ss << "2\_Задание\_" << inputFiles[d] << "\_ouput\_" << filteringFuncsNames[j] << "\_k" << kSize[k] << ".bmp";

std::cout << "Input file: " << inputFiles[d] << endl;

FilteringFuncAverageTime(j, inputFiles[d], ss.str(), kSize[k], time, 20);

if (j == 0 || j == 3)

T1[k][d] = time;

std::cout << "Длительность: " << time << " с." << endl;

resultsFile << time << ";" << (double)T1[k][d] / time << ";";

}

resultsFile << endl;

}

if (j == 0 || j == 3)

break;

}

}

}

#pragma endregion

#pragma region Texturing

// Формирует гистограмму на основе карты яркости в рамке с радиусами RH, RW на позиции (x,y)

vector<float> formHist(BYTE\*\* &BrMap, int height, int width, int x, int y, int RH, int RW)

{

int index = 0;

int coordX;

int coordY;

//инициализирую нулями

vector<float> hist = vector<float>(256);

//прохожу по рамке

for (int Y = -RH; Y <= RH; Y++)

{

coordY = y + Y;

for (int X = -RW; X <= RW; X++)

{

coordX = x + X;

if (coordX < 0)

coordX = 0;

if (coordX >= width)

coordX = width - 1;

if (coordY < 0)

coordY = 0;

if (coordY >= height)

coordY = height - 1;

//инкрементирует элемент гистограммы, соответствующий яркости текущего пикселя

hist[BrMap[coordY][coordX]] += 1;

}

}

//определю количество пикселей в рамке

int size = (RH \* 2 + 1)\*(RW \* 2 + 1);

//нормирую гистограмму

for (int i = 0; i < 256; i++)

hist[i] /= size;

return hist;

}

// Формирует гистограмму на основе карты яркости в рамке с радиусами RH, RW на позиции (x,y)

//Параллельный вариант с использованием циклов из ОМП

vector<float> formHistOMP(BYTE\*\* &BrMap, int height, int width, int x, int y, int RH, int RW)

{

int index = 0;

int coordX;

int coordY;

//инициализирую нулями

vector<float> hist = vector<float>(256);

//прохожу по рамке

#pragma omp parallel for shared(BrMap, hist) firstprivate(x, y, width, height, RH, RW) schedule(dynamic, 35)

for (int Y = -RH; Y <= RH; Y++)

{

coordY = y + Y;

for (int X = -RW; X <= RW; X++)

{

coordX = x + X;

if (coordX < 0)

coordX = 0;

if (coordX >= width)

coordX = width - 1;

if (coordY < 0)

coordY = 0;

if (coordY >= height)

coordY = height - 1;

//инкрементирует элемент гистограммы, соответствующий яркости текущего пикселя

#pragma omp critical

{

hist[BrMap[coordY][coordX]] += 1;

}

}

}

//определю количество пикселей в рамке

int size = (RH \* 2 + 1)\*(RW \* 2 + 1);

//нормирую гистограмму

for (int i = 0; i < 256; i++)

hist[i] /= size;

return hist;

}

//Создаёт метрики по текущей гистограмме

void getMetrics(float &m2, float &u, float &r, float &e, vector<float> &hist)

{

float m = 0;

for (int i = 0; i < 256; i++)

m += hist[i] \* i;

for (int i = 0; i < 256; i++)

{

m2 += pow((i - m), 2)\*hist[i];

e += (hist[i] != 0) ? hist[i] \* log2(hist[i]) : 0;

u += pow(hist[i], 2);

}

r = 1 - (1 / (1 + m2));

e \*= -1;

}

//Создаёт метрики по текущей гистограмме

//Параллельный вариант с использованием циклов и секций из ОМП

void getMetricsOmp(float &m2, float &u, float &r, float &e, vector<float> &hist)

{

float m = 0;

#pragma omp parallel for reduction(+:m) schedule(dynamic, 45)

for (int i = 0; i < 256; i++)

m += hist[i] \* i;

float step = 256 / 4;

int mod = 256 % 4;

int t1 = 0;

int t2 = step \* 1 + mod;

int t3 = t2 + step;

int t4 = t3 + step;

int t5 = t4 + step;

#pragma omp parallel sections reduction(+:m2, e, u) shared(t1, t2, t3, t4, t5)

{

#pragma omp section

{

for (int i = t1; i < t2; i++)

{

m2 += pow((i - m), 2)\*hist[i];

e += (hist[i] != 0) ? hist[i] \* log2(hist[i]) : 0;

u += pow(hist[i], 2);

}

}

#pragma omp section

{

for (int i = t2; i < t3; i++)

{

m2 += pow((i - m), 2)\*hist[i];

e += (hist[i] != 0) ? hist[i] \* log2(hist[i]) : 0;

u += pow(hist[i], 2);

}

}

#pragma omp section

{

for (int i = t3; i < t4; i++)

{

m2 += pow((i - m), 2)\*hist[i];

e += (hist[i] != 0) ? hist[i] \* log2(hist[i]) : 0;

u += pow(hist[i], 2);

}

}

#pragma omp section

{

for (int i = t4; i < t5; i++)

{

m2 += pow((i - m), 2)\*hist[i];

e += (hist[i] != 0) ? hist[i] \* log2(hist[i]) : 0;

u += pow(hist[i], 2);

}

}

}

r = 1 - (1 / (1 + m2));

e \*= -1;

}

//Определение текстурных признаков

void textureFilter(RGBQUAD \*\*&image, int height, int width, int rh, int rw, float \*\*&M, float \*\*&U, float \*\*&R, float \*\*&E)

{

vector<float> hist;//256

BYTE \*\*Brightness = new BYTE\*[height]();

for (int y = 0; y < height; y++)

{

Brightness[y] = new BYTE[width]();

//перевожу цветную картинку в карту яркости

for (int x = 0; x < width; x++)

Brightness[y][x] = image[y][x].rgbRed\*0.299 + image[y][x].rgbGreen\*0.587 + image[y][x].rgbBlue\*0.114;

}

for (int y = 0; y < height; y++)

for (int x = 0; x < width; x++)

{

//получаю гистограмму для окна

hist = formHist(Brightness, height, width, x, y, rh, rw);

//получаю метрики для текущего положения окна

getMetrics(M[y][x], U[y][x], R[y][x], E[y][x], hist);

}

//возвращаю метрики наверх

}

//Параллельная реализация, использует распараллеливание внешнего цикла на Omp For

void textureFilterOMP(RGBQUAD \*\*&image, int height, int width, int rh, int rw, float \*\*&M, float \*\*&U, float \*\*&R, float \*\*&E)

{

BYTE \*\*Brightness = new BYTE\*[height];

#pragma omp parallel for shared(Brightness, image) schedule(dynamic, 50)

for (int y = 0; y < height; y++)

{

Brightness[y] = new BYTE[width];

for (int x = 0; x < width; x++)

//перевожу цветную картинку в карут яркости

Brightness[y][x] = image[y][x].rgbRed\*0.299 + image[y][x].rgbGreen\*0.587 + image[y][x].rgbBlue\*0.114;

}

#pragma omp parallel for shared(Brightness, image, M, U, R, E) schedule(dynamic, 50)

for (int y = 0; y < height; y++)

for (int x = 0; x < width; x++)

{

//получаю гистограмму для окна

vector<float> hist = formHist(Brightness, height, width, x, y, rh, rw);

//получаю метрики для текущего положения окна

getMetrics(M[y][x], U[y][x], R[y][x], E[y][x], hist);

}

//возвращаю метрики наверх

}

//Паралелльный вариант с внешним распараллеливанием parallel\_for

void textureFilterTBB(RGBQUAD \*\*&image, int height, int width, int rh, int rw, float \*\*&M, float \*\*&U, float \*\*&R, float \*\*&E)

{

BYTE \*\*Brightness = new BYTE\*[height];

tbb::parallel\_for(tbb::blocked\_range<int>(0, height), [&](tbb::blocked\_range<int> r)

{

for (int y = r.begin(); y < r.end(); y++)

{

Brightness[y] = new BYTE[width];

//перевожу цветную картинку в карут яркости

for (int x = 0; x < width; x++)

Brightness[y][x] = image[y][x].rgbRed\*0.299 + image[y][x].rgbGreen\*0.587 + image[y][x].rgbBlue\*0.114;

}

});

tbb::parallel\_for(tbb::blocked\_range<int>(0, height), [&](tbb::blocked\_range<int> r)

{

for (int y = r.begin(); y < r.end(); y++)

{

for (int x = 0; x < width; x++)

{

vector<float> hist = formHist(Brightness, height, width, x, y, rh, rw);

getMetrics(M[y][x], U[y][x], R[y][x], E[y][x], hist);

}

}

});

}

#pragma endregion

#pragma region Task3\_Texturing

//Оценивает карту метрик, возвращает минимальное и максимальное значение признаков

void getInterval(float\*\* &map, int Height, int Width, float& \_min, float& \_max)

{

\_min = map[0][0];

\_max = map[0][0];

for (int y = 0; y < Height; y++)

for (int x = 0; x < Width; x++)

{

\_max = max(\_max, map[y][x]);

\_min = min(\_min, map[y][x]);

}

}

//Функция формирующая картинку по карте признаков

void formImage(BITMAPFILEHEADER head, BITMAPINFOHEADER info, float\*\* &T, int Height, int Width, string fname, float t1, float t2)

{

float Max = 0, Min = 0;

getInterval(T, Height, Width, Min, Max);

float T1 = (Max - Min)\*t1 + Min;

float T2 = (Max - Min)\*t2 + Min;

RGBQUAD\*\* out = new RGBQUAD\*[Height];

for (int y = 0; y < Height; y++)

{

out[y] = new RGBQUAD[Width]();

for (int x = 0; x < Width; x++)

{

//Если в первом диапазоне, то крашу в зеленый

if (T[y][x] >= T2 && T[y][x] <= Max)

out[y][x].rgbGreen = 255;

//Если во втором диапазоне, то крашу в желтый

if (T[y][x] >= T1 && T[y][x] < T2)

{

out[y][x].rgbRed = 255;

out[y][x].rgbGreen = 255;

}

//Если в первом диапазоне, то крашу в красный

if (T[y][x] >= Min && T[y][x] < T1)

out[y][x].rgbRed = 255;

}

}

//Сохраняю в файл

BMPWrite(out, head, info, fname.c\_str());

for (int i = 0; i < Height; i++)

delete[] out[i];

delete[] out;

}

void CalcTexturingFunc(TextureFilterMethod func, string fileName, string methodFullName, int window, double& time, int iterations)

{

double avgTime = 0, avgTimeT = 0, correctAVG = 0;

double startTime, curTime;

double\* Times = new double[iterations];

float \*\*M, \*\*U, \*\*R, \*\*E;

stringstream str;

RGBQUAD\*\* sourceImage;

BITMAPFILEHEADER head;

BITMAPINFOHEADER info;

BMPRead(sourceImage, head, info, fileName.c\_str());

M = new float\*[info.biHeight];

U = new float\*[info.biHeight];

R = new float\*[info.biHeight];

E = new float\*[info.biHeight];

for (int i = 0; i < info.biHeight; i++)

{

M[i] = new float[info.biWidth]();

U[i] = new float[info.biWidth]();

R[i] = new float[info.biWidth]();

E[i] = new float[info.biWidth]();

}

std::cout << "[";

for (int i = 0; i < iterations; i++)

{

startTime = omp\_get\_wtime();

func(sourceImage, info.biHeight, info.biWidth, window, window, M, U, R, E);

std::cout << "#";

curTime = omp\_get\_wtime() - startTime;

Times[i] = curTime;

avgTime += curTime;

if (i == 0)

{

//сохранение получившихся изображений в файлы

str = stringstream();

str << "3\_Задание\_" << fileName << "\_M\_" << "(" << methodFullName << ")[" << window << "].bmp";

formImage(head, info, M, info.biHeight, info.biWidth, str.str().c\_str(), 0.2, 0.8);

str = stringstream();

str << "3\_Задание\_" << fileName << "\_U\_" << "(" << methodFullName << ")[" << window << "].bmp";

formImage(head, info, U, info.biHeight, info.biWidth, str.str().c\_str(), 0.2, 0.8);

str = stringstream();

str << "3\_Задание\_" << fileName << "\_R\_" << "(" << methodFullName << ")[" << window << "].bmp";

formImage(head, info, R, info.biHeight, info.biWidth, str.str().c\_str(), 0.2, 0.8);

str = stringstream();

str << "3\_Задание\_" << fileName << "\_E\_" << "(" << methodFullName << ")[" << window << "].bmp";

formImage(head, info, E, info.biHeight, info.biWidth, str.str().c\_str(), 0.2, 0.8);

}

}

std::cout << "]\n";

avgTime /= iterations;

avgTimeT = AvgTrustedInterval(avgTime, Times, iterations);

time = avgTimeT;

for (int i = 0; i < info.biHeight; i++)

{

delete[] sourceImage[i];

delete[] M[i];

delete[] U[i];

delete[] R[i];

delete[] E[i];

}

delete[] sourceImage;

delete[] M;

delete[] U;

delete[] R;

delete[] E;

}

void CalcTexturingFuncs()

{

std::ofstream resultsFile;

resultsFile.open("Task3Results.csv", std::ios\_base::app);

string inputFiles[2] = { "1280x720.bmp", "1920x1080.bmp" };

string\* funcsName = new string[3]{ "Вычисление текстурных признаков (последовательно)", "Вычисление текстурных признаков (Open MP)", "Вычисление текстурных признаков (TBB)" };

int\* windowSizes = new int[3]{5, 7, 9};

TextureFilterMethod\* texturingFuncs = new TextureFilterMethod[3]{ textureFilter, textureFilterOMP, textureFilterTBB };

double posled[3][2];

double time, spd;

int iterations = 10;

resultsFile << "Название функции;Кол-во потоков;Окрестность;НД1;;НД2;;\n";

resultsFile << ";;;Время;Sp;Время;Sp;\n";

for (int m = 0; m < 3; m++)

{

cout << "Функция: " << funcsName[m] << endl;

resultsFile << funcsName[m] << ";";

for (int t = 2; t < 5; t++)

{

if (t != 2)

{

resultsFile << ";";

}

if (m == 0)

{

t = 1;

}

resultsFile << t << ";";

cout << "Кол-во потоков: " << t << endl;

omp\_set\_num\_threads(t);

global\_control global\_limit(global\_control::max\_allowed\_parallelism, t);

for (int k = 0; k < 3; k++)

{

if (k != 0)

resultsFile << ";;";

cout << "Окно размером: " << windowSizes[k] << "x" << windowSizes[k] << endl;

resultsFile << windowSizes[k] << "x" << windowSizes[k] << ";";

for (int img = 0; img < 2; img++)

{

cout << "Изображение: " << inputFiles[img] << endl;

CalcTexturingFunc(texturingFuncs[m], inputFiles[img], funcsName[m], windowSizes[k], time, iterations);

if (m == 0)

{

posled[k][img] = time;

}

cout << "Длительность: " << time << endl;

resultsFile << time << ";" << posled[k][img] / time << ";";

}

resultsFile << endl;

}

if (m == 0)

{

break;

}

}

cout << "\n=\n\n";

}

}

#pragma endregion

int main()

{

setlocale(LC\_ALL, "Russian");

int wait, choice;

cout << "[1]: Задание 1\n[2]: Задание 2\n[3]: задание 3\n";

cin >> choice;

switch (choice)

{

case 1:

{

CalcMatrixFuncs();

break;

}

case 2:

{

TaskFilteringMethods();

break;

}

default:

{

CalcTexturingFuncs();

break;

}

}

cout << "Конец программы...\n";

std::cin >> wait;

return 0;

}

Вывод: в результате проделанной работы были получены практические навыки по написанию алгоритмов обработки массивов, а также их реализации с помощью Intel oneAPI.