**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**

Параллельные вычисления

|  |
| --- |
| Алгоритмы обработки массивов и их реализация  с использованием oneTBB |

Руководитель А.Г. Зотин

подпись, дата инициалы, фамилия

Обучающийся БПИ20-02, 201219047 Р.А. Сухачев

номер группы, зачетной книжки подпись, дата инициалы, фамилия

Красноярск 2023 г.

# ЦЕЛЬ РАБОТЫ

Изучение различных видов реализации алгоритмов обработки массивов с использованием возможностей Intel oneTBB. 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.

# порядок выполнения работы

1. Ознакомиться с теоретическими сведениями.

2. Выполнить задания 1–2.

3. Подготовить отчет по лабораторной работе.

4.  Защитить лабораторную работу перед преподавателем.

# постановка задачи

Задание 5.1. Разработайте консольное приложение, выполняющее

следующие действия над матрицами:

* сложение двух матриц (по элементам);
* перемножение матриц (по элементам, не по правилам математики);
* вычислением суммы всех элементов (для двух исходных матриц);
* вычисления значения суммы максимального и минимального элемента среди всех элементов двух исходных матриц.

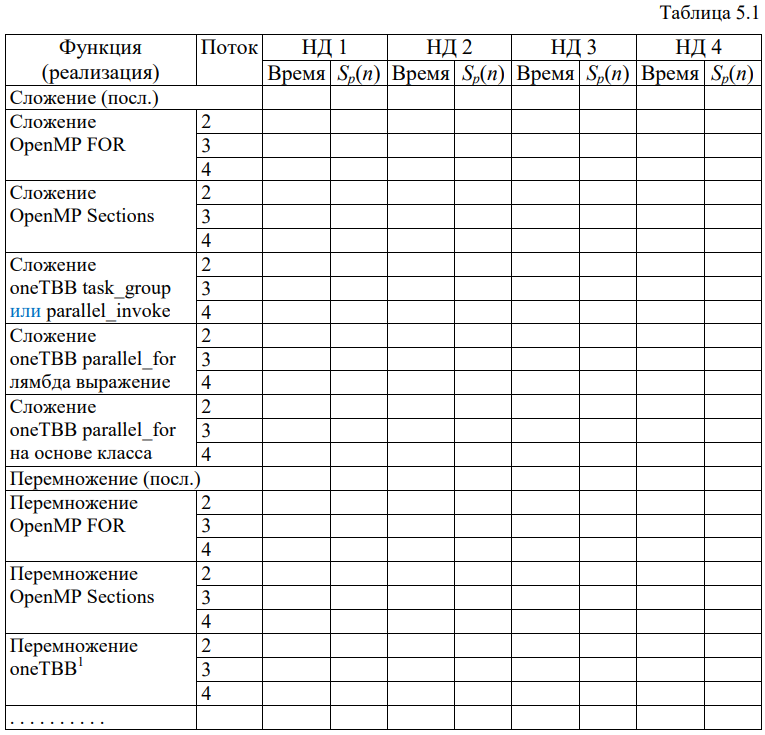
При реализации использовать различные возможности OpenMP и

Intel oneTBB. В частности, для Intel oneTBB необходимо выполнить реализацию не менее двух подходов редукционных вычислений (на выбор студента):

* с использованием классов,
* с использованием структурных объектов
* на основе лямбда выражения.

Проведите экспериментальное исследование по обработке данных

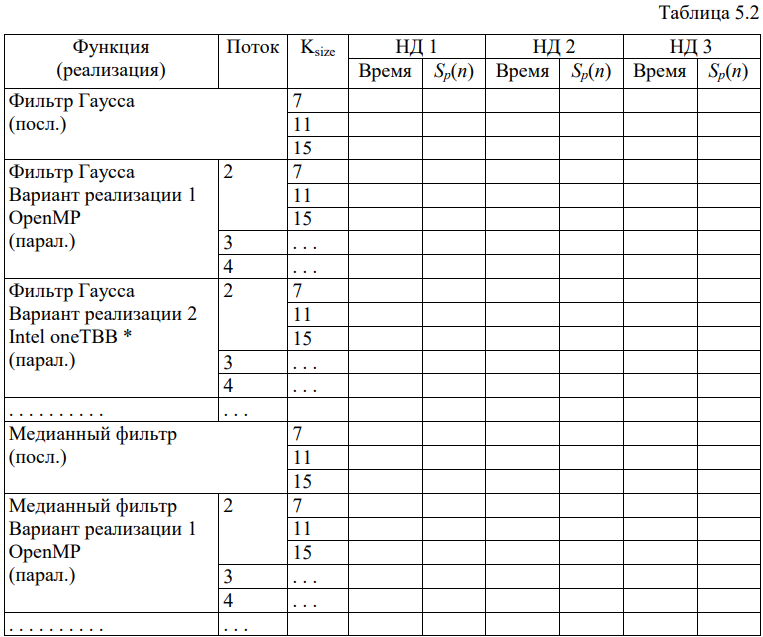
вещественного типа с разными реализациями алгоритмов (не менее 100 запусков для каждой реализации). При проведении экспериментов используйте массивы размерностью от 3000×3000 до 8000×8000. Выполните исследование для четырех наборов данных. Результаты эксперимента для каждой функции оформить в виде табл. 5.1.



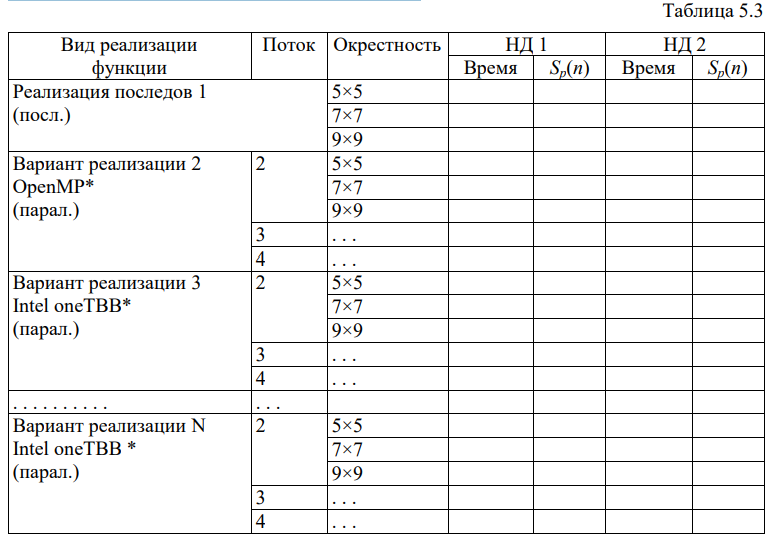
Задание 5.2. Разработайте консольное приложение, реализующее

фильтрации изображения (фильтр Гаусса и медианный). При реализации параллельных вариантов алгоритмов используйте возможности OpenMP и Intel oneTBB. В приложение должен загружаться файлы input\_X.bmp и формироваться выходные файлы output\_X\_alg.bmp, где alg означает имя используемого алгоритма обработки и его реализации (или его номер).

Проведите экспериментальное исследование по обработке данных с разными реализациями (не менее 20 запусков для каждой реализации) для трех изображений (от 1280×720 до 3840×2160) с размерами окрестности Ksize×Ksize. В рамках лабораторной работы Ksize = (RH×2+1) = (RW×2+1). Результаты исследования оформить в виде табл. 5.2.



Задание 5.3. 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 из диапазона [1600×900 , 2560×1440]. Результаты эксперимента для расчета данных для построения карты текстурных признаков оформить в виде табл. 5.3.



# ХОД РАБОТЫ

Задание 1.

Конфигурация компьютера и параметры операционной системы на котором производится выполнение данного задания:

Процессор: Intel Core i5 9400f 3.4 ГГц (4.3 ГГц) – 6 ядер 6 потоков

Операционная система: Windows 11

Код программы:

**Файл Header1.h**:

#pragma once

#include <Windows.h>

#include <fstream>

void BMPRead(RGBTRIPLE\*\*&, BITMAPFILEHEADER&, BITMAPINFOHEADER&, const char\*);

void BMPWrite(RGBTRIPLE\*\*& rgb, int imWidth, int imHeight, const char\* fout);

unsigned char get\_row\_data\_padding(unsigned int width);

unsigned int bmp24b\_file\_size\_calc(unsigned int width, unsigned int height);

unsigned char get\_row\_data\_padding(unsigned int width) {

return (width % 4 == 0) ? 0 : (4 - (width \* sizeof(RGBTRIPLE)) % 4);

}

unsigned int bmp24b\_file\_size\_calc(unsigned int width, unsigned int height) {

return sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER) + height \* (width \* sizeof(RGBTRIPLE) + get\_row\_data\_padding(width));

}

void BMPRead(RGBTRIPLE\*\*& rgb, BITMAPFILEHEADER& header, \

BITMAPINFOHEADER& bmiHeader, const char\* fin)

{

std::ifstream InFile(fin, std::ios::binary);

InFile.read((char\*)(&header), sizeof(BITMAPFILEHEADER));

InFile.read((char\*)(&bmiHeader), sizeof(BITMAPINFOHEADER));

rgb = new RGBTRIPLE \* [bmiHeader.biHeight];

rgb[0] = new RGBTRIPLE[bmiHeader.biWidth \* bmiHeader.biHeight];

for (int i = 1; i < bmiHeader.biHeight; i++)

{

rgb[i] = &rgb[0][bmiHeader.biWidth \* i];

}

InFile.seekg(header.bfOffBits, std::ios::beg);

int padding = get\_row\_data\_padding(bmiHeader.biWidth);

char tmp[3] = { 0,0,0 };

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

{

InFile.read((char\*)(&rgb[bmiHeader.biHeight - 1 - i][0]), bmiHeader.biWidth \* sizeof(RGBTRIPLE)); // RGBTRIPLE {Blue Green bRed;}

if (padding > 0)

InFile.read((char\*)(&tmp[0]), padding);

}

InFile.close();

}

void BMPWrite(RGBTRIPLE\*\*& rgb, int imWidth, int imHeight, const char\* fout)

{

std::ofstream OutFile(fout, std::ios::binary);

BITMAPFILEHEADER header = { 0 };

header.bfType = ('M' << 8) + 'B';

header.bfSize = bmp24b\_file\_size\_calc(imWidth, imHeight);;

header.bfOffBits = 54;

BITMAPINFOHEADER bmiHeader = { 0 };

bmiHeader.biSize = 40;

bmiHeader.biWidth = imWidth;

bmiHeader.biHeight = imHeight;

bmiHeader.biPlanes = 1;

bmiHeader.biBitCount = 24;

bmiHeader.biSizeImage = header.bfSize - sizeof(BITMAPINFOHEADER) - sizeof(BITMAPFILEHEADER);

OutFile.write((char\*)(&header), sizeof(BITMAPFILEHEADER));

OutFile.write((char\*)(&bmiHeader), sizeof(BITMAPINFOHEADER));

int padding = get\_row\_data\_padding(bmiHeader.biWidth);

char tmp[3] = { 0,0,0 };

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

{

OutFile.write((char\*)&(rgb[bmiHeader.biHeight - i - 1][0]), bmiHeader.biWidth \* sizeof(RGBTRIPLE));

if (padding > 0)

OutFile.write((char\*)(&tmp[0]), padding);

}

OutFile.close();

}

**Файл Header.h**:

#pragma once

#include "Header1.h"

void texture\_posled(RGBTRIPLE\*\*& rgb\_input, BITMAPFILEHEADER& header, BITMAPINFOHEADER& bmiHeader, const char\* in, int ksize) {

BMPRead(rgb\_input, header, bmiHeader, in);

RGBTRIPLE\*\* rgbout\_m2 = new RGBTRIPLE \* [bmiHeader.biHeight];

RGBTRIPLE\*\* rgbout\_u = new RGBTRIPLE \* [bmiHeader.biHeight];

RGBTRIPLE\*\* rgbout\_r = new RGBTRIPLE \* [bmiHeader.biHeight];

RGBTRIPLE\*\* rgbout\_e = new RGBTRIPLE \* [bmiHeader.biHeight];

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

rgbout\_m2[i] = new RGBTRIPLE[bmiHeader.biWidth];

rgbout\_u[i] = new RGBTRIPLE[bmiHeader.biWidth];

rgbout\_r[i] = new RGBTRIPLE[bmiHeader.biWidth];

rgbout\_e[i] = new RGBTRIPLE[bmiHeader.biWidth];

}

int\*\* BIMAGE = new int\* [bmiHeader.biHeight];

double\*\* M2 = new double\* [bmiHeader.biHeight];

double\*\* R = new double\* [bmiHeader.biHeight];

double\*\* U = new double\* [bmiHeader.biHeight];

double\*\* E = new double\* [bmiHeader.biHeight];

double Hist[256];

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

BIMAGE[i] = new int[bmiHeader.biWidth];

M2[i] = new double[bmiHeader.biWidth];

R[i] = new double[bmiHeader.biWidth];

U[i] = new double[bmiHeader.biWidth];

E[i] = new double[bmiHeader.biWidth];

}

int size = ksize \* ksize;

int rh = ksize / 2;

int rw = ksize / 2;

for (int Y = 0; Y < bmiHeader.biHeight; Y++) {

for (int X = 0; X < bmiHeader.biWidth; X++) {

BIMAGE[Y][X] = rgb\_input[Y][X].rgbtRed \* 0.299 + rgb\_input[Y][X].rgbtGreen \* 0.587 + rgb\_input[Y][X].rgbtBlue \* 0.114;

}

}

double min\_m2 = DBL\_MAX, min\_u = DBL\_MAX, min\_r = DBL\_MAX, min\_e = DBL\_MAX;

double max\_m2 = -DBL\_MAX, max\_u = -DBL\_MAX, max\_r = -DBL\_MAX, max\_e = -DBL\_MAX;

for (int Y = 0; Y < bmiHeader.biHeight; Y++) {

for (int X = 0; X < bmiHeader.biWidth; X++) {

for (int i = 0; i <= 255; i++) {

Hist[i] = 0;

}

for (int DY = -rh; DY <= rh; DY++) {

int KY = Y + DY;

if (KY < 0) {

KY = 0;

}

if (KY > bmiHeader.biHeight - 1) {

KY = bmiHeader.biHeight - 1;

}

for (int DX = -rw; DX <= rw; DX++) {

int KX = X + DX;

if (KX < 0)

KX = 0;

if (KX > bmiHeader.biWidth - 1)

KX = bmiHeader.biWidth - 1;

Hist[BIMAGE[KY][KX]] = Hist[BIMAGE[KY][KX]] + 1;

}

}

int size = (rh \* 2 + 1) \* (rw \* 2 + 1);

double m = 0, m2 = 0, e = 0, u = 0;

for (int i = 0; i <= 255; i++) {

Hist[i] = Hist[i] / size;

}

for (int i = 0; i <= 255; i++) {

m += Hist[i] \* i;

}

for (int i = 0; i <= 255; i++) {

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

if (Hist[i] != 0) {

e += Hist[i] \* log2(Hist[i]);

}

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

}

M2[Y][X] = m2;

U[Y][X] = u;

R[Y][X] = 1 - (1 / (1 + m2));

E[Y][X] = -1 \* e;

if (Y == 0 && X == 0)

{

max\_m2 = m2;

max\_u = u;

max\_r = (1 - (1 / (1 + m2)));

max\_e = (-1 \* e);

min\_m2 = m2;

min\_u = u;

min\_r = (1 - (1 / (1 + m2)));

min\_e = (-1 \* e);

}

else

{

if (max\_m2 < m2) {

max\_m2 = m2;

}

if (max\_u < u) {

max\_u = u;

}

if (max\_r < (1 - (1 / (1 + m2)))) {

max\_r = (1 - (1 / (1 + m2)));

}

if (max\_e < (-1 \* e)) {

max\_e = (-1 \* e);

}

if (min\_m2 > m2) {

min\_m2 = m2;

}

if (min\_u > u) {

min\_u = u;

}

if (min\_r > (1 - (1 / (1 + m2)))) {

min\_r = (1 - (1 / (1 + m2)));

}

if (min\_e > (-1 \* e)) {

min\_e = (-1 \* e);

}

}

}

}

double T1\_m2 = (max\_m2 - min\_m2) \* 0.2 + min\_m2;

double T2\_m2 = (max\_m2 - min\_m2) \* 0.7 + min\_m2;

double T1\_u = (max\_u - min\_u) \* 0.2 + min\_u;

double T2\_u = (max\_u - min\_u) \* 0.7 + min\_u;

double T1\_r = (max\_r - min\_r) \* 0.2 + min\_r;

double T2\_r = (max\_r - min\_r) \* 0.7 + min\_r;

double T1\_e = (max\_e - min\_e) \* 0.2 + min\_e;

double T2\_e = (max\_e - min\_e) \* 0.7 + min\_e;

for (int Y = 0; Y < bmiHeader.biHeight; Y++) {

for (int X = 0; X < bmiHeader.biWidth; X++) {

if (M2[Y][X] < T1\_m2) {

rgbout\_m2[Y][X].rgbtRed = 0;

rgbout\_m2[Y][X].rgbtGreen = 255;

rgbout\_m2[Y][X].rgbtBlue = 0;

}

else if (M2[Y][X] >= T1\_m2 && M2[Y][X] <= T2\_m2) {

rgbout\_m2[Y][X].rgbtRed = 255;

rgbout\_m2[Y][X].rgbtGreen = 255;

rgbout\_m2[Y][X].rgbtBlue = 0;

}

else {

rgbout\_m2[Y][X].rgbtRed = 255;

rgbout\_m2[Y][X].rgbtGreen = 0;

rgbout\_m2[Y][X].rgbtBlue = 0;

}

if (U[Y][X] < T1\_u) {

rgbout\_u[Y][X].rgbtRed = 0;

rgbout\_u[Y][X].rgbtGreen = 255;

rgbout\_u[Y][X].rgbtBlue = 0;

}

else if (U[Y][X] >= T1\_u && U[Y][X] <= T2\_u) {

rgbout\_u[Y][X].rgbtRed = 255;

rgbout\_u[Y][X].rgbtGreen = 255;

rgbout\_u[Y][X].rgbtBlue = 0;

}

else {

rgbout\_u[Y][X].rgbtRed = 255;

rgbout\_u[Y][X].rgbtGreen = 0;

rgbout\_u[Y][X].rgbtBlue = 0;

}

if (R[Y][X] < T1\_r) {

rgbout\_r[Y][X].rgbtRed = 0;

rgbout\_r[Y][X].rgbtGreen = 255;

rgbout\_r[Y][X].rgbtBlue = 0;

}

else if (R[Y][X] >= T1\_r && R[Y][X] <= T2\_r) {

rgbout\_r[Y][X].rgbtRed = 255;

rgbout\_r[Y][X].rgbtGreen = 255;

rgbout\_r[Y][X].rgbtBlue = 0;

}

else {

rgbout\_r[Y][X].rgbtRed = 255;

rgbout\_r[Y][X].rgbtGreen = 0;

rgbout\_r[Y][X].rgbtBlue = 0;

}

if (E[Y][X] < T1\_e) {

rgbout\_e[Y][X].rgbtRed = 0;

rgbout\_e[Y][X].rgbtGreen = 255;

rgbout\_e[Y][X].rgbtBlue = 0;

}

else if (E[Y][X] >= T1\_e && M2[Y][X] <= T2\_e) {

rgbout\_e[Y][X].rgbtRed = 255;

rgbout\_e[Y][X].rgbtGreen = 255;

rgbout\_e[Y][X].rgbtBlue = 0;

}

else {

rgbout\_e[Y][X].rgbtRed = 255;

rgbout\_e[Y][X].rgbtGreen = 0;

rgbout\_e[Y][X].rgbtBlue = 0;

}

}

}

BMPWrite(rgbout\_m2, bmiHeader.biWidth, bmiHeader.biHeight, "posl\_\_output\_m2.bmp");

BMPWrite(rgbout\_u, bmiHeader.biWidth, bmiHeader.biHeight, "posl\_\_output\_u.bmp");

BMPWrite(rgbout\_r, bmiHeader.biWidth, bmiHeader.biHeight, "posl\_\_output\_r.bmp");

BMPWrite(rgbout\_e, bmiHeader.biWidth, bmiHeader.biHeight, "posl\_\_output\_e.bmp");

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

{

delete[] rgbout\_m2[i];

delete[] rgbout\_u[i];

delete[] rgbout\_r[i];

delete[] rgbout\_e[i];

delete[] BIMAGE[i];

delete[] M2[i];

delete[] R[i];

delete[] U[i];

delete[] E[i];

}

delete[] rgbout\_m2;

delete[] rgbout\_u;

delete[] rgbout\_r;

delete[] rgbout\_e;

delete[] BIMAGE;

delete[] M2;

delete[] R;

delete[] U;

delete[] E;

}

void texture\_omp(RGBTRIPLE\*\*& rgb\_input, BITMAPFILEHEADER& header, BITMAPINFOHEADER& bmiHeader, const char\* in, int ksize) {

BMPRead(rgb\_input, header, bmiHeader, in);

RGBTRIPLE\*\* rgbout\_m2 = new RGBTRIPLE \* [bmiHeader.biHeight];

RGBTRIPLE\*\* rgbout\_u = new RGBTRIPLE \* [bmiHeader.biHeight];

RGBTRIPLE\*\* rgbout\_r = new RGBTRIPLE \* [bmiHeader.biHeight];

RGBTRIPLE\*\* rgbout\_e = new RGBTRIPLE \* [bmiHeader.biHeight];

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

{

rgbout\_m2[i] = new RGBTRIPLE[bmiHeader.biWidth];

rgbout\_u[i] = new RGBTRIPLE[bmiHeader.biWidth];

rgbout\_r[i] = new RGBTRIPLE[bmiHeader.biWidth];

rgbout\_e[i] = new RGBTRIPLE[bmiHeader.biWidth];

}

int\*\* BIMAGE = new int\* [bmiHeader.biHeight];

double\*\* M2 = new double\* [bmiHeader.biHeight];

double\*\* R = new double\* [bmiHeader.biHeight];

double\*\* U = new double\* [bmiHeader.biHeight];

double\*\* E = new double\* [bmiHeader.biHeight];

double Hist[256];

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

BIMAGE[i] = new int[bmiHeader.biWidth];

M2[i] = new double[bmiHeader.biWidth];

R[i] = new double[bmiHeader.biWidth];

U[i] = new double[bmiHeader.biWidth];

E[i] = new double[bmiHeader.biWidth];

}

int size = ksize \* ksize;

int RH = ksize / 2;

int RW = ksize / 2;

#pragma omp parallel for

for (int Y = 0; Y < bmiHeader.biHeight; Y++) {

for (int X = 0; X < bmiHeader.biWidth; X++) {

BIMAGE[Y][X] = rgb\_input[Y][X].rgbtRed \* 0.299 + rgb\_input[Y][X].rgbtGreen \* 0.587 + rgb\_input[Y][X].rgbtBlue \* 0.114;

}

}

double min\_m2 = DBL\_MAX, min\_u = DBL\_MAX, min\_r = DBL\_MAX, min\_e = DBL\_MAX;

double max\_m2 = -DBL\_MAX, max\_u = -DBL\_MAX, max\_r = -DBL\_MAX, max\_e = -DBL\_MAX;

#pragma omp parallel for reduction(max: max\_m2, max\_u, max\_r, max\_e) reduction(min: min\_m2, min\_u, min\_r, min\_e)

for (int y = 0; y < bmiHeader.biHeight; y++) {

for (int x = 0; x < bmiHeader.biWidth; x++) {

double Hist[256];

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

Hist[i] = 0;

for (int dy = -RH; dy <= RH; dy++) {

int ky = y + dy;

if (ky < 0) ky = 0;

if (ky > bmiHeader.biHeight - 1) ky = bmiHeader.biHeight - 1;

for (int dx = -RW; dx <= RW; dx++) {

int kx = x + dx;

if (kx < 0) kx = 0;

if (kx > bmiHeader.biWidth - 1) kx = bmiHeader.biWidth - 1;

Hist[BIMAGE[ky][kx]] = Hist[BIMAGE[ky][kx]] + 1;

}

}

double m = 0, m2 = 0, e = 0, u = 0;

for (int i = 0; i <= 255; i++)

Hist[i] = Hist[i] / size;

for (int i = 0; i <= 255; i++)

m += Hist[i] \* i;

for (int i = 0; i <= 255; i++) {

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

if (Hist[i] != 0) e += Hist[i] \* log2(Hist[i]);

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

}

M2[y][x] = m2;

U[y][x] = u;

R[y][x] = 1 - (1 / (1 + m2));

E[y][x] = -1 \* e;

if (y == 0 && x == 0)

{

max\_m2 = m2;

max\_u = u;

max\_r = (1 - (1 / (1 + m2)));

max\_e = (-1 \* e);

min\_m2 = m2;

min\_u = u;

min\_r = (1 - (1 / (1 + m2)));

min\_e = (-1 \* e);

}

else

{

if (max\_m2 < m2) {

max\_m2 = m2;

}

if (max\_u < u) {

max\_u = u;

}

if (max\_r < (1 - (1 / (1 + m2)))) {

max\_r = (1 - (1 / (1 + m2)));

}

if (max\_e < (-1 \* e)) {

max\_e = (-1 \* e);

}

if (min\_m2 > m2) {

min\_m2 = m2;

}

if (min\_u > u) {

min\_u = u;

}

if (min\_r > (1 - (1 / (1 + m2)))) {

min\_r = (1 - (1 / (1 + m2)));

}

if (min\_e > (-1 \* e)) {

min\_e = (-1 \* e);

}

}

}

}

double T1\_m2 = (max\_m2 - min\_m2) \* 0.2 + min\_m2;

double T2\_m2 = (max\_m2 - min\_m2) \* 0.7 + min\_m2;

double T1\_u = (max\_u - min\_u) \* 0.2 + min\_u;

double T2\_u = (max\_u - min\_u) \* 0.7 + min\_u;

double T1\_r = (max\_r - min\_r) \* 0.2 + min\_r;

double T2\_r = (max\_r - min\_r) \* 0.7 + min\_r;

double T1\_e = (max\_e - min\_e) \* 0.2 + min\_e;

double T2\_e = (max\_e - min\_e) \* 0.7 + min\_e;

for (int Y = 0; Y < bmiHeader.biHeight; Y++) {

for (int X = 0; X < bmiHeader.biWidth; X++) {

if (M2[Y][X] < T1\_m2) {

rgbout\_m2[Y][X].rgbtRed = 0;

rgbout\_m2[Y][X].rgbtGreen = 255;

rgbout\_m2[Y][X].rgbtBlue = 0;

}

else if (M2[Y][X] >= T1\_m2 && M2[Y][X] <= T2\_m2) {

rgbout\_m2[Y][X].rgbtRed = 255;

rgbout\_m2[Y][X].rgbtGreen = 255;

rgbout\_m2[Y][X].rgbtBlue = 0;

}

else {

rgbout\_m2[Y][X].rgbtRed = 255;

rgbout\_m2[Y][X].rgbtGreen = 0;

rgbout\_m2[Y][X].rgbtBlue = 0;

}

if (U[Y][X] < T1\_u) {

rgbout\_u[Y][X].rgbtRed = 0;

rgbout\_u[Y][X].rgbtGreen = 255;

rgbout\_u[Y][X].rgbtBlue = 0;

}

else if (U[Y][X] >= T1\_u && U[Y][X] <= T2\_u) {

rgbout\_u[Y][X].rgbtRed = 255;

rgbout\_u[Y][X].rgbtGreen = 255;

rgbout\_u[Y][X].rgbtBlue = 0;

}

else {

rgbout\_u[Y][X].rgbtRed = 255;

rgbout\_u[Y][X].rgbtGreen = 0;

rgbout\_u[Y][X].rgbtBlue = 0;

}

if (R[Y][X] < T1\_r) {

rgbout\_r[Y][X].rgbtRed = 0;

rgbout\_r[Y][X].rgbtGreen = 255;

rgbout\_r[Y][X].rgbtBlue = 0;

}

else if (R[Y][X] >= T1\_r && R[Y][X] <= T2\_r) {

rgbout\_r[Y][X].rgbtRed = 255;

rgbout\_r[Y][X].rgbtGreen = 255;

rgbout\_r[Y][X].rgbtBlue = 0;

}

else {

rgbout\_r[Y][X].rgbtRed = 255;

rgbout\_r[Y][X].rgbtGreen = 0;

rgbout\_r[Y][X].rgbtBlue = 0;

}

if (E[Y][X] < T1\_e) {

rgbout\_e[Y][X].rgbtRed = 0;

rgbout\_e[Y][X].rgbtGreen = 255;

rgbout\_e[Y][X].rgbtBlue = 0;

}

else if (E[Y][X] >= T1\_e && M2[Y][X] <= T2\_e) {

rgbout\_e[Y][X].rgbtRed = 255;

rgbout\_e[Y][X].rgbtGreen = 255;

rgbout\_e[Y][X].rgbtBlue = 0;

}

else {

rgbout\_e[Y][X].rgbtRed = 255;

rgbout\_e[Y][X].rgbtGreen = 0;

rgbout\_e[Y][X].rgbtBlue = 0;

}

}

}

BMPWrite(rgbout\_m2, bmiHeader.biWidth, bmiHeader.biHeight, "omp\_\_output\_m2.bmp");

BMPWrite(rgbout\_u, bmiHeader.biWidth, bmiHeader.biHeight, "omp\_\_output\_u.bmp");

BMPWrite(rgbout\_r, bmiHeader.biWidth, bmiHeader.biHeight, "omp\_\_output\_r.bmp");

BMPWrite(rgbout\_e, bmiHeader.biWidth, bmiHeader.biHeight, "omp\_\_output\_e.bmp");

#pragma omp parallel for

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

{

delete[] rgbout\_m2[i];

delete[] rgbout\_u[i];

delete[] rgbout\_r[i];

delete[] rgbout\_e[i];

delete[] BIMAGE[i];

delete[] M2[i];

delete[] R[i];

delete[] U[i];

delete[] E[i];

}

delete[] rgbout\_m2;

delete[] rgbout\_u;

delete[] rgbout\_r;

delete[] rgbout\_e;

delete[] BIMAGE;

delete[] M2;

delete[] R;

delete[] U;

delete[] E;

}

//Параллельное нахождение min/max reduce в texture на основе класса

class MinMaxTexture {

private:

int RH\_;

int RW\_;

int\*\* BIMAGE\_;

double\*\* M2\_;

double\*\* R\_;

double\*\* U\_;

double\*\* E\_;

public:

double min\_m2, min\_u, min\_r, min\_e;

double max\_m2, max\_u, max\_r, max\_e;

void operator()(const tbb::blocked\_range2d<size\_t>& r) {

for (size\_t y = r.rows().begin(); y < r.rows().end(); y++) {

for (size\_t x = r.cols().begin(); x < r.cols().end(); x++) {

double Hist[256];

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

Hist[i] = 0;

for (int dy = -RH\_; dy <= RH\_; dy++) {

int ky = y + dy;

if (ky < 0) ky = 0;

if (ky > r.rows().end() - 1) ky = r.rows().end() - 1;

for (int dx = -RW\_; dx <= RW\_; dx++) {

int kx = x + dx;

if (kx < 0) kx = 0;

if (kx > r.cols().end() - 1) kx = r.cols().end() - 1;

Hist[BIMAGE\_[ky][kx]] = Hist[BIMAGE\_[ky][kx]] + 1;

}

}

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

double m = 0, m2 = 0, e = 0, u = 0;

for (int i = 0; i <= 255; i++)

Hist[i] = Hist[i] / Size;

for (int i = 0; i <= 255; i++)

m += Hist[i] \* i;

for (int i = 0; i <= 255; i++) {

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

if (Hist[i] != 0) e += Hist[i] \* log2(Hist[i]);

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

}

M2\_[y][x] = m2;

U\_[y][x] = u;

R\_[y][x] = 1 - (1 / (1 + m2));

E\_[y][x] = -1 \* e;

if (y == 0 && x == 0)

{

max\_m2 = m2;

max\_u = u;

max\_r = (1 - (1 / (1 + m2)));

max\_e = (-1 \* e);

min\_m2 = m2;

min\_u = u;

min\_r = (1 - (1 / (1 + m2)));

min\_e = (-1 \* e);

}

else

{

if (max\_m2 < m2) {

max\_m2 = m2;

}

if (max\_u < u) {

max\_u = u;

}

if (max\_r < (1 - (1 / (1 + m2)))) {

max\_r = (1 - (1 / (1 + m2)));

}

if (max\_e < (-1 \* e)) {

max\_e = (-1 \* e);

}

if (min\_m2 > m2) {

min\_m2 = m2;

}

if (min\_u > u) {

min\_u = u;

}

if (min\_r > (1 - (1 / (1 + m2)))) {

min\_r = (1 - (1 / (1 + m2)));

}

if (min\_e > (-1 \* e)) {

min\_e = (-1 \* e);

}

}

}

}

}

MinMaxTexture(MinMaxTexture& x, tbb::split) :RH\_(x.RH\_), RW\_(x.RW\_), BIMAGE\_(x.BIMAGE\_), M2\_(x.M2\_), R\_(x.R\_), U\_(x.U\_), E\_(x.E\_),

min\_m2(DBL\_MAX), min\_u(DBL\_MAX), min\_r(DBL\_MAX), min\_e(DBL\_MAX),

max\_m2(-DBL\_MAX), max\_u(-DBL\_MAX), max\_r(-DBL\_MAX), max\_e(-DBL\_MAX) {}

void join(const MinMaxTexture& y) {

if (max\_m2 < y.max\_m2) max\_m2 = y.max\_m2;

if (max\_u < y.max\_u) max\_u = y.max\_u;

if (max\_r < (1 - (1 / (1 + y.max\_m2)))) max\_r = (1 - (1 / (1 + y.max\_m2)));

if (max\_e < (-1 \* y.max\_e)) max\_e = (-1 \* y.max\_e);

if (min\_m2 > y.min\_m2) min\_m2 = y.min\_m2;

if (min\_u > y.min\_u) min\_u = y.min\_u;

if (min\_r > (1 - (1 / (1 + y.min\_m2)))) min\_r = (1 - (1 / (1 + y.min\_m2)));

if (min\_e > (-1 \* y.min\_e)) min\_e = (-1 \* y.min\_e);

}

MinMaxTexture(int RH, int RW, int\*\* BIMAGE, double\*\* M2,

double\*\* R, double\*\* U, double\*\* E) :

RH\_(RH),

RW\_(RW),

BIMAGE\_(BIMAGE),

M2\_(M2),

R\_(R),

U\_(U),

E\_(E),

min\_m2(DBL\_MAX), min\_u(DBL\_MAX), min\_r(DBL\_MAX), min\_e(DBL\_MAX),

max\_m2(-DBL\_MAX), max\_u(-DBL\_MAX), max\_r(-DBL\_MAX), max\_e(-DBL\_MAX)

{}

};

void texture\_tbb(RGBTRIPLE\*\*& rgb\_input, BITMAPFILEHEADER& header, BITMAPINFOHEADER& bmiHeader, const char\* in, int ksize) {

BMPRead(rgb\_input, header, bmiHeader, in);

RGBTRIPLE\*\* rgbout\_m2 = new RGBTRIPLE \* [bmiHeader.biHeight];

RGBTRIPLE\*\* rgbout\_u = new RGBTRIPLE \* [bmiHeader.biHeight];

RGBTRIPLE\*\* rgbout\_r = new RGBTRIPLE \* [bmiHeader.biHeight];

RGBTRIPLE\*\* rgbout\_e = new RGBTRIPLE \* [bmiHeader.biHeight];

int size = ksize \* ksize;

int RH = ksize / 2;

int RW = ksize / 2;

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

rgbout\_m2[i] = new RGBTRIPLE[bmiHeader.biWidth];

rgbout\_u[i] = new RGBTRIPLE[bmiHeader.biWidth];

rgbout\_r[i] = new RGBTRIPLE[bmiHeader.biWidth];

rgbout\_e[i] = new RGBTRIPLE[bmiHeader.biWidth];

}

int imWidth = bmiHeader.biWidth;

int imHeight = bmiHeader.biHeight;

int\*\* BIMAGE = new int\* [bmiHeader.biHeight];

double\*\* M2 = new double\* [bmiHeader.biHeight];

double\*\* R = new double\* [bmiHeader.biHeight];

double\*\* U = new double\* [bmiHeader.biHeight];

double\*\* E = new double\* [bmiHeader.biHeight];

double Hist[256];

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

BIMAGE[i] = new int[bmiHeader.biWidth];

M2[i] = new double[bmiHeader.biWidth];

R[i] = new double[bmiHeader.biWidth];

U[i] = new double[bmiHeader.biWidth];

E[i] = new double[bmiHeader.biWidth];

}

tbb::parallel\_for(tbb::blocked\_range2d<double>(0, bmiHeader.biHeight, 0, bmiHeader.biWidth), [&](tbb::blocked\_range2d<double> r) {

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

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

BIMAGE[Y][X] = rgb\_input[Y][X].rgbtRed \* 0.299 + rgb\_input[Y][X].rgbtGreen \* 0.587 + rgb\_input[Y][X].rgbtBlue \* 0.114;

}

}

});

MinMaxTexture MMT(RH, RW, BIMAGE, M2, R, U, E);

tbb::parallel\_reduce(tbb::blocked\_range2d<size\_t>(0, imHeight, 0, imWidth), MMT);

double min\_m2 = MMT.min\_m2, min\_u = MMT.min\_u, min\_r = MMT.min\_r, min\_e = MMT.min\_e;

double max\_m2 = MMT.max\_m2, max\_u = MMT.max\_u, max\_r = MMT.max\_r, max\_e = MMT.max\_e;

double T1\_m2 = (max\_m2 - min\_m2) \* 0.2 + min\_m2;

double T2\_m2 = (max\_m2 - min\_m2) \* 0.7 + min\_m2;

double T1\_u = (max\_u - min\_u) \* 0.2 + min\_u;

double T2\_u = (max\_u - min\_u) \* 0.7 + min\_u;

double T1\_r = (max\_r - min\_r) \* 0.2 + min\_r;

double T2\_r = (max\_r - min\_r) \* 0.7 + min\_r;

double T1\_e = (max\_e - min\_e) \* 0.2 + min\_e;

double T2\_e = (max\_e - min\_e) \* 0.7 + min\_e;

for (int Y = 0; Y < bmiHeader.biHeight; Y++) {

for (int X = 0; X < bmiHeader.biWidth; X++) {

if (M2[Y][X] < T1\_m2) {

rgbout\_m2[Y][X].rgbtRed = 0;

rgbout\_m2[Y][X].rgbtGreen = 255;

rgbout\_m2[Y][X].rgbtBlue = 0;

}

else if (M2[Y][X] >= T1\_m2 && M2[Y][X] <= T2\_m2) {

rgbout\_m2[Y][X].rgbtRed = 255;

rgbout\_m2[Y][X].rgbtGreen = 255;

rgbout\_m2[Y][X].rgbtBlue = 0;

}

else {

rgbout\_m2[Y][X].rgbtRed = 255;

rgbout\_m2[Y][X].rgbtGreen = 0;

rgbout\_m2[Y][X].rgbtBlue = 0;

}

if (U[Y][X] < T1\_u) {

rgbout\_u[Y][X].rgbtRed = 0;

rgbout\_u[Y][X].rgbtGreen = 255;

rgbout\_u[Y][X].rgbtBlue = 0;

}

else if (U[Y][X] >= T1\_u && U[Y][X] <= T2\_u) {

rgbout\_u[Y][X].rgbtRed = 255;

rgbout\_u[Y][X].rgbtGreen = 255;

rgbout\_u[Y][X].rgbtBlue = 0;

}

else {

rgbout\_u[Y][X].rgbtRed = 255;

rgbout\_u[Y][X].rgbtGreen = 0;

rgbout\_u[Y][X].rgbtBlue = 0;

}

if (R[Y][X] < T1\_r) {

rgbout\_r[Y][X].rgbtRed = 0;

rgbout\_r[Y][X].rgbtGreen = 255;

rgbout\_r[Y][X].rgbtBlue = 0;

}

else if (R[Y][X] >= T1\_r && R[Y][X] <= T2\_r) {

rgbout\_r[Y][X].rgbtRed = 255;

rgbout\_r[Y][X].rgbtGreen = 255;

rgbout\_r[Y][X].rgbtBlue = 0;

}

else {

rgbout\_r[Y][X].rgbtRed = 255;

rgbout\_r[Y][X].rgbtGreen = 0;

rgbout\_r[Y][X].rgbtBlue = 0;

}

if (E[Y][X] < T1\_e) {

rgbout\_e[Y][X].rgbtRed = 0;

rgbout\_e[Y][X].rgbtGreen = 255;

rgbout\_e[Y][X].rgbtBlue = 0;

}

else if (E[Y][X] >= T1\_e && M2[Y][X] <= T2\_e) {

rgbout\_e[Y][X].rgbtRed = 255;

rgbout\_e[Y][X].rgbtGreen = 255;

rgbout\_e[Y][X].rgbtBlue = 0;

}

else {

rgbout\_e[Y][X].rgbtRed = 255;

rgbout\_e[Y][X].rgbtGreen = 0;

rgbout\_e[Y][X].rgbtBlue = 0;

}

}

}

BMPWrite(rgbout\_m2, bmiHeader.biWidth, bmiHeader.biHeight, "tbb\_\_output\_m2.bmp");

BMPWrite(rgbout\_u, bmiHeader.biWidth, bmiHeader.biHeight, "tbb\_\_output\_u.bmp");

BMPWrite(rgbout\_r, bmiHeader.biWidth, bmiHeader.biHeight, "tbb\_\_output\_r.bmp");

BMPWrite(rgbout\_e, bmiHeader.biWidth, bmiHeader.biHeight, "tbb\_\_output\_e.bmp");

tbb::parallel\_for(tbb::blocked\_range<int>(0, bmiHeader.biHeight), [&](tbb::blocked\_range<int> r) {

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

{

delete[] rgbout\_m2[i];

delete[] rgbout\_u[i];

delete[] rgbout\_r[i];

delete[] rgbout\_e[i];

delete[] BIMAGE[i];

delete[] M2[i];

delete[] R[i];

delete[] U[i];

delete[] E[i];

}

});

delete[] rgbout\_m2;

delete[] rgbout\_u;

delete[] rgbout\_r;

delete[] rgbout\_e;

delete[] BIMAGE;

delete[] M2;

delete[] R;

delete[] U;

delete[] E;

}

**Файл main.cpp**:

#include <iostream>

#include <omp.h>

#include <vector>

#include <fstream>

#include <cmath>

#include <tbb/parallel\_for.h>

#include <tbb/info.h>

#include <tbb/tbb.h>

#include <tbb/blocked\_range.h>

#include <tbb/parallel\_reduce.h>

#include <iomanip>

#include "Header.h"

#include "Header1.h"

using namespace std;

using namespace tbb;

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_НАЧАЛО СУММЫ\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//

inline void sum\_posled(double\*\* arr1, double\*\* arr2, int size1, int size2) {

double\*\* arr\_res = new double\* [size1];

for (int i = 0; i < size1; i++)

arr\_res[i] = new double[size2];

for (size\_t i = 0; i < size1; i++) {

for (size\_t j = 0; j < size2; j++) {

arr\_res[i][j] = arr1[i][j] + arr2[i][j];

}

}

for (int i = 0; i < size1; i++)

delete[] arr\_res[i];

delete[] arr\_res;

}

//Параллельное сложение openmp parallel for

inline void sum\_par\_omp(double\*\* arr1, double\*\* arr2, int size1, int size2) {

double\*\* arr\_res = new double\* [size1];

for (int i = 0; i < size1; i++)

arr\_res[i] = new double[size2];

#pragma omp parallel for

for (long i = 0; i < size1; i++) {

for (long j = 0; j < size2; j++) {

arr\_res[i][j] = arr1[i][j] + arr2[i][j];

}

}

for (int i = 0; i < size1; i++)

delete[] arr\_res[i];

delete[] arr\_res;

}

//Параллельное сложение openmp sections

inline void sum\_par\_omp\_sections(double\*\* arr1, double\*\* arr2, int size1, int size2) {

int num\_thread = omp\_get\_max\_threads();

double\*\* arr\_res = new double\* [size1];

for (int i = 0; i < size1; i++)

arr\_res[i] = new double[size2];

int p0 = 0,

p1 = size1 / num\_thread,

p2 = 2 \* size1 / num\_thread,

p3 = 3 \* size1 / num\_thread,

p4 = size1;

#pragma omp parallel sections

{

#pragma omp section

{

for (int i = p0; i < p1; i++) {

for (int j = 0; j < size1; j++) {

arr\_res[i][j] = arr1[i][j] + arr2[i][j];

}

}

}

#pragma omp section

{

for (int i = p1; i < p2; i++) {

for (int j = 0; j < size1; j++) {

arr\_res[i][j] = arr1[i][j] + arr2[i][j];

}

}

}

#pragma omp section

{

if (num\_thread > 2)

for (int i = p2; i < p3; i++)

for (int j = 0; j < size1; j++) {

arr\_res[i][j] = arr1[i][j] + arr2[i][j];

}

}

#pragma omp section

{

if (num\_thread > 3)

for (int i = p3; i < p4; i++)

for (int j = 0; j < size1; j++) {

arr\_res[i][j] = arr1[i][j] + arr2[i][j];

}

}

}

for (int i = 0; i < size1; i++)

delete[] arr\_res[i];

delete[] arr\_res;

}

//Параллельное сложение oneTBB task\_group

inline void sum\_par\_tbb\_task\_group(double\*\* arr1, double\*\* arr2, int size1, int size2) {

int num\_thread = tbb::global\_control::active\_value(tbb::global\_control::max\_allowed\_parallelism);

double\*\* arr\_res = new double\* [size1];

for (int i = 0; i < size1; i++)

arr\_res[i] = new double[size2];

int p0 = 0,

p1 = size1 / num\_thread,

p2 = 2 \* size1 / num\_thread,

p3 = 3 \* size1 / num\_thread,

p4 = size1;

tbb:task\_group g;

g.run([&] { for (long i = p0; i < p1; i++) {

for (int j = 0; j < size1; j++) {

arr\_res[i][j] = arr1[i][j] + arr2[i][j];

}

}; });

g.run([&] { for (long i = p1; i < p2; i++) {

for (int j = 0; j < size1; j++) {

arr\_res[i][j] = arr1[i][j] + arr2[i][j];

}

}; });

if (num\_thread > 2)

g.run([&] { for (long i = p2; i < p3; i++) {

for (int j = 0; j < size1; j++) {

arr\_res[i][j] = arr1[i][j] + arr2[i][j];

}

}; });

if (num\_thread > 3)

g.run([&] { for (long i = p3; i < p4; i++) {

for (int j = 0; j < size1; j++) {

arr\_res[i][j] = arr1[i][j] + arr2[i][j];

}

}; });

g.wait();

for (int i = 0; i < size1; i++)

delete[] arr\_res[i];

delete[] arr\_res;

}

//Параллельное сложение oneTBB parallel\_for лямбда выражение

inline void sum\_par\_tbb\_par\_for\_lambda(double\*\* arr1, double\*\* arr2, int size1, int size2) {

double\*\* arr\_res = new double\* [size1];

for (int i = 0; i < size1; i++)

arr\_res[i] = new double[size2];

tbb::parallel\_for(tbb::blocked\_range2d<double>(0, size1, 0, size2), [&](tbb::blocked\_range2d<double> r)

{

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

for (size\_t j = r.cols().begin(); j < r.cols().end(); ++j) {

arr\_res[i][j] = arr1[i][j] + arr2[i][j];

}

}

});

for (int i = 0; i < size1; i++)

delete[] arr\_res[i];

delete[] arr\_res;

}

//Сложение oneTBB parallel\_for на основе класса

class ArraySummer {

double\*\* arr1\_;

double\*\* arr2\_;

double\*\* arr\_res\_;

public:

ArraySummer(double\*\* arr1, double\*\* arr2, double\*\* arr\_res) :arr1\_(arr1), arr2\_(arr2), arr\_res\_(arr\_res) {}

void operator()(const tbb::blocked\_range2d<size\_t>& r) const {

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

for (size\_t j = r.cols().begin(); j < r.cols().end(); j++) {

arr\_res\_[i][j] = arr1\_[i][j] + arr2\_[i][j];

}

}

}

};

inline void sum\_par\_for\_class(double\*\* arr1, double\*\* arr2, int size1, int size2) {

double\*\* arr\_res = new double\* [size1];

for (int i = 0; i < size1; i++)

arr\_res[i] = new double[size2];

tbb::parallel\_for(tbb::blocked\_range2d<size\_t>(0, size1, 0, size2), ArraySummer(arr1, arr2, arr\_res));

for (int i = 0; i < size1; i++)

delete[] arr\_res[i];

delete[] arr\_res;

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_КОНЕЦ СУММЫ\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_НАЧАЛО УМНОЖЕНИЯ\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//

//

//Последовательное перемножение

inline void mul\_posled(double\*\* arr1, double\*\* arr2, int size1, int size2) {

double\*\* arr\_res = new double\* [size1];

for (int i = 0; i < size1; i++)

arr\_res[i] = new double[size2];

for (size\_t i = 0; i < size1; i++) {

for (size\_t j = 0; j < size2; j++) {

arr\_res[i][j] = arr1[i][j] \* arr2[i][j];

}

}

for (int i = 0; i < size1; i++)

delete[] arr\_res[i];

delete[] arr\_res;

}

//Параллельное перемножение openmp parallel for

inline void mul\_par\_omp(double\*\* arr1, double\*\* arr2, int size1, int size2) {

double\*\* arr\_res = new double\* [size1];

for (int i = 0; i < size1; i++)

arr\_res[i] = new double[size2];

#pragma omp parallel for

for (long i = 0; i < size1; i++) {

for (long j = 0; j < size2; j++) {

arr\_res[i][j] = arr1[i][j] \* arr2[i][j];

}

}

for (int i = 0; i < size1; i++)

delete[] arr\_res[i];

delete[] arr\_res;

}

//Параллельное перемножение openmp sections

inline void mul\_par\_omp\_sections(double\*\* arr1, double\*\* arr2, int size1, int size2) {

int num\_thread = omp\_get\_max\_threads();

double\*\* arr\_res = new double\* [size1];

for (int i = 0; i < size1; i++)

arr\_res[i] = new double[size2];

int p0 = 0,

p1 = size1 / num\_thread,

p2 = 2 \* size1 / num\_thread,

p3 = 3 \* size1 / num\_thread,

p4 = size1;

#pragma omp parallel sections

{

#pragma omp section

{

for (int i = p0; i < p1; i++) {

for (int j = 0; j < size1; j++) {

arr\_res[i][j] = arr1[i][j] \* arr2[i][j];

}

}

}

#pragma omp section

{

for (int i = p1; i < p2; i++) {

for (int j = 0; j < size1; j++) {

arr\_res[i][j] = arr1[i][j] \* arr2[i][j];

}

}

}

#pragma omp section

{

if (num\_thread > 2)

for (int i = p2; i < p3; i++)

for (int j = 0; j < size1; j++) {

arr\_res[i][j] = arr1[i][j] \* arr2[i][j];

}

}

#pragma omp section

{

if (num\_thread > 3)

for (int i = p3; i < p4; i++)

for (int j = 0; j < size1; j++) {

arr\_res[i][j] = arr1[i][j] \* arr2[i][j];

}

}

}

for (int i = 0; i < size1; i++)

delete[] arr\_res[i];

delete[] arr\_res;

}

//Параллельное перемножение oneTBB task\_group

inline void mul\_par\_tbb\_task\_group(double\*\* arr1, double\*\* arr2, int size1, int size2) {

int num\_thread = tbb::global\_control::active\_value(tbb::global\_control::max\_allowed\_parallelism);

double\*\* arr\_res = new double\* [size1];

for (int i = 0; i < size1; i++)

arr\_res[i] = new double[size2];

int p0 = 0,

p1 = size1 / num\_thread,

p2 = 2 \* size1 / num\_thread,

p3 = 3 \* size1 / num\_thread,

p4 = size1;

tbb:task\_group g;

g.run([&] { for (long i = p0; i < p1; i++) {

for (int j = 0; j < size1; j++) {

arr\_res[i][j] = arr1[i][j] \* arr2[i][j];

}

}; });

g.run([&] { for (long i = p1; i < p2; i++) {

for (int j = 0; j < size1; j++) {

arr\_res[i][j] = arr1[i][j] \* arr2[i][j];

}

}; });

if (num\_thread > 2)

g.run([&] { for (long i = p2; i < p3; i++) {

for (int j = 0; j < size1; j++) {

arr\_res[i][j] = arr1[i][j] \* arr2[i][j];

}

}; });

if (num\_thread > 3)

g.run([&] { for (long i = p3; i < p4; i++) {

for (int j = 0; j < size1; j++) {

arr\_res[i][j] = arr1[i][j] \* arr2[i][j];

}

}; });

g.wait();

for (int i = 0; i < size1; i++)

delete[] arr\_res[i];

delete[] arr\_res;

}

//Параллельное перемножение oneTBB parallel\_for лямбда выражение

inline void mul\_tbb\_par\_for\_lambda(double\*\* arr1, double\*\* arr2, int size1, int size2) {

double\*\* arr\_res = new double\* [size1];

for (int i = 0; i < size1; i++)

arr\_res[i] = new double[size2];

tbb::parallel\_for(tbb::blocked\_range2d<double>(0, size1, 0, size2), [&](tbb::blocked\_range2d<double> r)

{

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

for (size\_t j = r.cols().begin(); j < r.cols().end(); ++j) {

arr\_res[i][j] = arr1[i][j] \* arr2[i][j];

}

}

});

for (int i = 0; i < size1; i++)

delete[] arr\_res[i];

delete[] arr\_res;

}

//Параллельное перемножение oneTBB parallel\_for на основе класса

class ArrayMul {

double\*\* arr1\_;

double\*\* arr2\_;

double\*\* arr\_res\_;

public:

ArrayMul(double\*\* arr1, double\*\* arr2, double\*\* arr\_res) :arr1\_(arr1), arr2\_(arr2), arr\_res\_(arr\_res) {}

void operator()(const tbb::blocked\_range2d<size\_t>& r) const {

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

for (size\_t j = r.cols().begin(); j < r.cols().end(); j++) {

arr\_res\_[i][j] = arr1\_[i][j] \* arr2\_[i][j];

}

}

}

};

inline void mul\_par\_for\_class(double\*\* arr1, double\*\* arr2, int size1, int size2) {

double\*\* arr\_res = new double\* [size1];

for (int i = 0; i < size1; i++)

arr\_res[i] = new double[size2];

tbb::parallel\_for(tbb::blocked\_range2d<size\_t>(0, size1, 0, size2), ArrayMul(arr1, arr2, arr\_res));

for (int i = 0; i < size1; i++)

delete[] arr\_res[i];

delete[] arr\_res;

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_КОНЕЦ УМНОЖЕНИЯ\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_НАЧАЛО СУММЫ ЭЛЕМЕНТОВ\_\_\_\_\_\_\_\_\_//

//Последовательное сложение всех элементов

inline void sum\_all\_posled(double\*\* arr1, double\*\* arr2, int size1, int size2) {

double sum = 0;

for (size\_t i = 0; i < size1; i++) {

for (size\_t j = 0; j < size2; j++) {

sum += arr1[i][j] + arr2[i][j];

}

}

}

//Параллельное сложение всех элементов openmp parallel for

inline void sum\_all\_par\_omp(double\*\* arr1, double\*\* arr2, int size1, int size2) {

double sum = 0;

#pragma omp parallel for reduction(+:sum)

for (long i = 0; i < size1; i++) {

for (long j = 0; j < size2; j++) {

sum += arr1[i][j] + arr2[i][j];

}

}

}

//Параллельное сложение всех элементов openmp sections

inline void sum\_all\_par\_omp\_sections(double\*\* arr1, double\*\* arr2, int size1, int size2) {

int num\_thread = omp\_get\_max\_threads();

double s1 = 0, s2 = 0, s3 = 0, s4 = 0, sum = 0;

int p0 = 0,

p1 = size1 / num\_thread,

p2 = 2 \* size1 / num\_thread,

p3 = 3 \* size1 / num\_thread,

p4 = size1;

#pragma omp parallel sections

{

#pragma omp section

{

for (int i = p0; i < p1; i++) {

for (int j = 0; j < size1; j++) {

s1 += arr1[i][j] + arr2[i][j];

}

}

}

#pragma omp section

{

for (int i = p1; i < p2; i++) {

for (int j = 0; j < size1; j++) {

s2 += arr1[i][j] + arr2[i][j];

}

}

}

#pragma omp section

{

if (num\_thread > 2)

for (int i = p2; i < p3; i++)

for (int j = 0; j < size1; j++) {

s3 += arr1[i][j] + arr2[i][j];

}

}

#pragma omp section

{

if (num\_thread > 3)

for (int i = p3; i < p4; i++)

for (int j = 0; j < size1; j++) {

s4 += arr1[i][j] + arr2[i][j];

}

}

}

sum = s1 + s2 + s3 + s4;

}

//Параллельное сложение всех элементов oneTBB task\_group

inline void sum\_all\_par\_tbb\_task\_group(double\*\* arr1, double\*\* arr2, int size1, int size2) {

int num\_thread = tbb::global\_control::active\_value(tbb::global\_control::max\_allowed\_parallelism);

double s1 = 0, s2 = 0, s3 = 0, s4 = 0, sum = 0;

int p0 = 0,

p1 = size1 / num\_thread,

p2 = 2 \* size1 / num\_thread,

p3 = 3 \* size1 / num\_thread,

p4 = size1;

tbb:task\_group g;

g.run([&] { for (long i = p0; i < p1; i++) {

for (int j = 0; j < size1; j++) {

s1 += arr1[i][j] + arr2[i][j];

}

}; });

g.run([&] { for (long i = p1; i < p2; i++) {

for (int j = 0; j < size1; j++) {

s2 += arr1[i][j] + arr2[i][j];

}

}; });

if (num\_thread > 2)

g.run([&] { for (long i = p2; i < p3; i++) {

for (int j = 0; j < size1; j++) {

s3 += arr1[i][j] + arr2[i][j];

}

}; });

if (num\_thread > 3)

g.run([&] { for (long i = p3; i < p4; i++) {

for (int j = 0; j < size1; j++) {

s4 += arr1[i][j] + arr2[i][j];

}

}; });

g.wait();

sum = s1 + s2 + s3 + s4;

}

//Параллельное сложение всех элементов oneTBB parallel\_reduce лямбда выражение

inline void sum\_all\_tbb\_par\_reduce\_lambda(double\*\* arr1, double\*\* arr2, int size1, int size2) {

auto total = tbb::parallel\_reduce(tbb::blocked\_range2d<size\_t>(0, size1, 0, size2), 0.0, [&](tbb::blocked\_range2d<size\_t> r, double running\_total)

{

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

for (size\_t j = r.cols().begin(); j < r.cols().end(); ++j) {

running\_total += arr1[i][j] + arr2[i][j];

}

}

return running\_total;

}, std::plus<double>());

}

//Параллельное сложение всех элементов oneTBB parallel\_reduce на основе класс

class reduce\_par {

public:

double sum;

void operator()(const tbb::blocked\_range2d<size\_t>& r) {

double sum\_local = sum;

double\*\* arr1loc = arr1\_;

double\*\* arr2loc = arr2\_;

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

for (size\_t j = r.cols().begin(); j < r.cols().end(); j++) {

sum\_local += arr1loc[i][j] + arr2loc[i][j];

}

}

sum = sum\_local;

}

reduce\_par(reduce\_par& r, tbb::split) : sum(0.0), arr1\_(r.arr1\_), arr2\_(r.arr2\_) {}

void join(const reduce\_par& r) { sum += r.sum; }

reduce\_par(double\*\* arr1, double\*\* arr2) : sum(0.0), arr1\_(arr1), arr2\_(arr2) {}

private:

double\*\* arr1\_;

double\*\* arr2\_;

};

inline void sum\_all\_par\_reduce\_class(double\*\* arr1, double\*\* arr2, int size1, int size2) {

reduce\_par r(arr1, arr2);

parallel\_reduce(tbb::blocked\_range2d<size\_t>(0, size1, 0, size2), r);

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_КОНЕЦ СУММЫ ЭЛЕМЕНТОВ\_\_\_\_\_\_\_\_\_\_\_//

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_НАЧАЛО СУММЫ МАКС И МИН ЭЛЕМЕНТА\_\_\_\_\_\_\_//

//Последовательное нахождение min/max

inline void min\_max\_posled(double\*\* arr1, double\*\* arr2, int size1, int size2) {

double MaxValue = -DBL\_MAX;

double MinValue = DBL\_MAX;

for (size\_t i = 0; i < size1; i++) {

for (size\_t j = 0; j < size2; j++) {

if (arr1[i][j] > MaxValue) MaxValue = arr1[i][j];

if (arr2[i][j] > MaxValue) MaxValue = arr2[i][j];

if (arr1[i][j] < MinValue) MinValue = arr1[i][j];

if (arr2[i][j] < MinValue) MinValue = arr2[i][j];

}

}

}

//Параллельное нахождение min/max openmp parallel for

inline void min\_max\_par\_omp(double\*\* arr1, double\*\* arr2, int size1, int size2) {

double MaxValue = -DBL\_MAX;

double MinValue = DBL\_MAX;

#pragma omp parallel for reduction(max: MaxValue) reduction(min: MinValue)

for (int i = 0; i < size1; i++) {

for (int j = 0; j < size2; j++) {

if (arr1[i][j] > MaxValue) MaxValue = arr1[i][j];

if (arr2[i][j] > MaxValue) MaxValue = arr2[i][j];

if (arr1[i][j] < MinValue) MinValue = arr1[i][j];

if (arr2[i][j] < MinValue) MinValue = arr2[i][j];

}

}

}

//Параллельное нахождение min/max openmp sections

inline void min\_max\_par\_omp\_sections(double\*\* arr1, double\*\* arr2, int size1, int size2) {

int num\_thread = omp\_get\_max\_threads();

int p0 = 0,

p1 = size1 / num\_thread,

p2 = 2 \* size1 / num\_thread,

p3 = 3 \* size1 / num\_thread,

p4 = size1;

double MaxValue = -DBL\_MAX;

double MinValue = DBL\_MAX;

#pragma omp parallel sections

{

#pragma omp section

{

for (int i = p0; i < p1; i++) {

for (int j = 0; j < size1; j++) {

if (arr1[i][j] > MaxValue) MaxValue = arr1[i][j];

if (arr2[i][j] > MaxValue) MaxValue = arr2[i][j];

if (arr1[i][j] < MinValue) MinValue = arr1[i][j];

if (arr2[i][j] < MinValue) MinValue = arr2[i][j];

}

}

}

#pragma omp section

{

for (int i = p1; i < p2; i++) {

for (int j = 0; j < size1; j++) {

if (arr1[i][j] > MaxValue) MaxValue = arr1[i][j];

if (arr2[i][j] > MaxValue) MaxValue = arr2[i][j];

if (arr1[i][j] < MinValue) MinValue = arr1[i][j];

if (arr2[i][j] < MinValue) MinValue = arr2[i][j];

}

}

}

#pragma omp section

{

if (num\_thread > 2)

for (int i = p2; i < p3; i++)

for (int j = 0; j < size1; j++) {

if (arr1[i][j] > MaxValue) MaxValue = arr1[i][j];

if (arr2[i][j] > MaxValue) MaxValue = arr2[i][j];

if (arr1[i][j] < MinValue) MinValue = arr1[i][j];

if (arr2[i][j] < MinValue) MinValue = arr2[i][j];

}

}

#pragma omp section

{

if (num\_thread > 3)

for (int i = p3; i < p4; i++)

for (int j = 0; j < size1; j++) {

if (arr1[i][j] > MaxValue) MaxValue = arr1[i][j];

if (arr2[i][j] > MaxValue) MaxValue = arr2[i][j];

if (arr1[i][j] < MinValue) MinValue = arr1[i][j];

if (arr2[i][j] < MinValue) MinValue = arr2[i][j];

}

}

}

}

//Параллельное нахождение min/max oneTBB task\_group

inline void min\_max\_par\_tbb\_task\_group(double\*\* arr1, double\*\* arr2, int size1, int size2) {

int num\_thread = tbb::global\_control::active\_value(tbb::global\_control::max\_allowed\_parallelism);

int p0 = 0,

p1 = size1 / num\_thread,

p2 = 2 \* size1 / num\_thread,

p3 = 3 \* size1 / num\_thread,

p4 = size1;

double MaxValue = -DBL\_MAX;

double MinValue = DBL\_MAX;

tbb:task\_group g;

g.run([&] { for (long i = p0; i < p1; i++) {

for (int j = 0; j < size1; j++) {

if (arr1[i][j] > MaxValue) MaxValue = arr1[i][j];

if (arr2[i][j] > MaxValue) MaxValue = arr2[i][j];

if (arr1[i][j] < MinValue) MinValue = arr1[i][j];

if (arr2[i][j] < MinValue) MinValue = arr2[i][j];

}

}; });

g.run([&] { for (long i = p1; i < p2; i++) {

for (int j = 0; j < size1; j++) {

if (arr1[i][j] > MaxValue) MaxValue = arr1[i][j];

if (arr2[i][j] > MaxValue) MaxValue = arr2[i][j];

if (arr1[i][j] < MinValue) MinValue = arr1[i][j];

if (arr2[i][j] < MinValue) MinValue = arr2[i][j];

}

}; });

if (num\_thread > 2)

g.run([&] { for (long i = p2; i < p3; i++) {

for (int j = 0; j < size1; j++) {

if (arr1[i][j] > MaxValue) MaxValue = arr1[i][j];

if (arr2[i][j] > MaxValue) MaxValue = arr2[i][j];

if (arr1[i][j] < MinValue) MinValue = arr1[i][j];

if (arr2[i][j] < MinValue) MinValue = arr2[i][j];

}

}; });

if (num\_thread > 3)

g.run([&] { for (long i = p3; i < p4; i++) {

for (int j = 0; j < size1; j++) {

if (arr1[i][j] > MaxValue) MaxValue = arr1[i][j];

if (arr2[i][j] > MaxValue) MaxValue = arr2[i][j];

if (arr1[i][j] < MinValue) MinValue = arr1[i][j];

if (arr2[i][j] < MinValue) MinValue = arr2[i][j];

}

}; });

g.wait();

}

//Параллельное нахождение min/max oneTBB parallel\_for лямбда выражение

struct minmax\_st

{

double v\_min;

double v\_max;

minmax\_st() { v\_min = DBL\_MAX; v\_max = -DBL\_MAX; };

minmax\_st(double val) :v\_min(val), v\_max(val) {};

};

struct minmax\_join {

minmax\_st operator()(const minmax\_st& \_Left, const minmax\_st&

\_Right) const {

minmax\_st tmp(0);

tmp.v\_max = max(\_Left.v\_max, \_Right.v\_max);

tmp.v\_min = min(\_Left.v\_min, \_Right.v\_min);

return tmp;

}

};

inline void min\_max\_tbb\_par\_for\_lambda(double\*\* arr1, double\*\* arr2, int size1, int size2) {

minmax\_st total\_mm = tbb::parallel\_reduce(tbb::blocked\_range2d<double>(0, size1, 0, size2), minmax\_st(), [&](tbb::blocked\_range2d<double> r, minmax\_st running\_maxmin)

{

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

for (size\_t j = r.cols().begin(); j < r.cols().end(); ++j) {

running\_maxmin.v\_max = max(running\_maxmin.v\_max, \*arr1[i]);

running\_maxmin.v\_max = max(running\_maxmin.v\_max, \*arr2[i]);

running\_maxmin.v\_min = min(running\_maxmin.v\_min, \*arr1[i]);

running\_maxmin.v\_min = min(running\_maxmin.v\_min, \*arr2[i]);

}

}

return running\_maxmin;

}, minmax\_join());

}

//Параллельное нахождение min/max на основе класса

class MinMaxCalc {

private:

double\*\* arr1\_;

double\*\* arr2\_;

public:

double MinValue;

double MaxValue;

void operator()(const tbb::blocked\_range2d<size\_t>& r) {

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

for (size\_t j = r.cols().begin(); j < r.cols().end(); j++) {

if (arr1\_[i][j] > MaxValue) MaxValue = arr1\_[i][j];

if (arr2\_[i][j] > MaxValue) MaxValue = arr2\_[i][j];

if (arr1\_[i][j] < MinValue) MinValue = arr1\_[i][j];

if (arr2\_[i][j] < MinValue) MinValue = arr2\_[i][j];

}

}

}

MinMaxCalc(MinMaxCalc& x, tbb::split) :arr1\_(x.arr1\_), arr2\_(x.arr2\_), MinValue(DBL\_MAX), MaxValue(-DBL\_MAX) {}

void join(const MinMaxCalc& y) {

if (y.MinValue < MinValue) MinValue = y.MinValue;

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

}

MinMaxCalc(double\*\* arr1, double\*\* arr2) :

arr1\_(arr1),

arr2\_(arr2),

MinValue(DBL\_MAX),

MaxValue(-DBL\_MAX)

{}

};

inline void min\_max\_par\_reduce\_class(double\*\* arr1, double\*\* arr2, int size1, int size2) {

MinMaxCalc MMC(arr1, arr2);

tbb::parallel\_reduce(tbb::blocked\_range2d<size\_t>(0, size1, 0, size2), MMC);

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_КОНЕЦ СУММ МАКС И МИН ЭЛЕМЕНТА\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_НАЧАЛО Медианный фильтр\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//

template<typename T>

void quickSort(T\* arr, long size) {

long i = 0;

long j = size - 1;

T pivot = arr[size / 2];

do {

while (arr[i] < pivot)

i++;

while (arr[j] > pivot)

j--;

if (i <= j) {

T temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

i++;

j--;

}

} while (i < j);

if (j > 0)

{

quickSort(arr, j + 1);

}

if (size > i)

{

quickSort(arr + i, size - i);

}

}

void median\_posled(int height, int width, RGBTRIPLE\*\* rgb\_input, RGBTRIPLE\*\* rgb\_output, int ksize) {

int size = ksize \* ksize;

int RH = ksize / 2, RW = ksize / 2;

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

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

int\* medmas\_red = new int[size];

int\* medmas\_green = new int[size];

int\* medmas\_blue = new int[size];

int masind = 0;

for (int dy = -RH; dy <= RH; dy++) {

int ky = y + dy;

if (ky < 0) ky = 0;

if (ky > height - 1) ky = height - 1;

for (int dx = -RW; dx <= RW; dx++) {

int kx = x + dx;

if (kx < 0) kx = 0;

if (kx > width - 1) kx = width - 1;

medmas\_red[masind] = rgb\_input[ky][kx].rgbtRed;

medmas\_blue[masind] = rgb\_input[ky][kx].rgbtBlue;

medmas\_green[masind] = rgb\_input[ky][kx].rgbtGreen;

masind++;

}

}

quickSort(medmas\_red, size);

rgb\_output[y][x].rgbtRed = medmas\_red[size / 2];

quickSort(medmas\_green, size);

rgb\_output[y][x].rgbtGreen = medmas\_green[size / 2];

quickSort(medmas\_blue, size);

rgb\_output[y][x].rgbtBlue = medmas\_blue[size / 2];

delete[] medmas\_blue;

delete[] medmas\_red;

delete[] medmas\_green;

}

}

}

void median\_parallel\_omp(int height, int width, RGBTRIPLE\*\* rgb\_input, RGBTRIPLE\*\* rgb\_output, int ksize) {

int size = ksize \* ksize;

int RH = ksize / 2, RW = ksize / 2;

#pragma omp parallel for

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

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

int\* medmas\_red = new int[size];

int\* medmas\_green = new int[size];

int\* medmas\_blue = new int[size];

int masind = 0;

for (int dy = -RH; dy <= RH; dy++) {

int ky = y + dy;

if (ky < 0) ky = 0;

if (ky > height - 1) ky = height - 1;

for (int dx = -RW; dx <= RW; dx++) {

int kx = x + dx;

if (kx < 0) kx = 0;

if (kx > width - 1) kx = width - 1;

medmas\_red[masind] = rgb\_input[ky][kx].rgbtRed;

medmas\_blue[masind] = rgb\_input[ky][kx].rgbtBlue;

medmas\_green[masind] = rgb\_input[ky][kx].rgbtGreen;

masind++;

}

}

quickSort<int>(medmas\_red, size);

rgb\_output[y][x].rgbtRed = medmas\_red[size / 2];

quickSort<int>(medmas\_green, size);

rgb\_output[y][x].rgbtGreen = medmas\_green[size / 2];

quickSort<int>(medmas\_blue, size);

rgb\_output[y][x].rgbtBlue = medmas\_blue[size / 2];

delete[] medmas\_blue;

delete[] medmas\_red;

delete[] medmas\_green;

}

}

}

void median\_parallel\_tbb(int height, int width, RGBTRIPLE\*\* rgb\_input, RGBTRIPLE\*\* rgb\_output, int ksize) {

int size = ksize \* ksize;

int RH = ksize / 2, RW = ksize / 2;

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++) {

int\* medmas\_red = new int[size];

int\* medmas\_green = new int[size];

int\* medmas\_blue = new int[size];

int masind = 0;

for (int dy = -RH; dy <= RH; dy++) {

int ky = y + dy;

if (ky < 0) ky = 0;

if (ky > height - 1) ky = height - 1;

for (int dx = -RW; dx <= RW; dx++) {

int kx = x + dx;

if (kx < 0) kx = 0;

if (kx > width - 1) kx = width - 1;

medmas\_red[masind] = rgb\_input[ky][kx].rgbtRed;

medmas\_blue[masind] = rgb\_input[ky][kx].rgbtBlue;

medmas\_green[masind] = rgb\_input[ky][kx].rgbtGreen;

masind++;

}

}

quickSort<int>(medmas\_red, size);

rgb\_output[y][x].rgbtRed = medmas\_red[size / 2];

quickSort<int>(medmas\_green, size);

rgb\_output[y][x].rgbtGreen = medmas\_green[size / 2];

quickSort<int>(medmas\_blue, size);

rgb\_output[y][x].rgbtBlue = medmas\_blue[size / 2];

delete[] medmas\_blue;

delete[] medmas\_red;

delete[] medmas\_green;

}

}

}

);

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_КОНЕЦ Медианный фильтр\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_НАЧАЛО Фильтр Гаусса \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//

double\*\* gauss\_matrix\_posled(int size) {

int RH = size / 2, RW = size / 2;

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

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

matrix[i] = new double[size];

double SUM = 0;

for (int y = -RH; y <= RH; y++) {

for (int x = -RW; x <= RW; x++) {

int YK = y + RH;

int XK = x + RW;

double CF = (1 / (2 \* 3.14 \* pow(0.8, 2))) \* exp(-1 \* (pow(x, 2) + pow(y, 2)) / (2 \* pow(0.8, 2)));

matrix[YK][XK] = CF;

SUM += CF;

}

}

for (int y = -RH; y <= RH; y++) {

for (int x = -RW; x <= RW; x++) {

int YK = y + RH;

int XK = x + RW;

matrix[YK][XK] /= SUM;

}

}

return matrix;

}

void gauss\_filter\_posled(int height, int width, RGBTRIPLE\*\* rgb\_input, RGBTRIPLE\*\* rgb\_output, int ksize) {

int size = ksize \* ksize;

int RH = ksize / 2, RW = ksize / 2;

double\*\* matrix = gauss\_matrix\_posled(ksize);

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

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

int LinF\_Value\_Red = 0;

int LinF\_Value\_Blue = 0;

int LinF\_Value\_Green = 0;

for (int dy = -RH; dy <= RH; dy++) {

int ky = y + dy;

if (ky < 0) ky = 0;

if (ky > height - 1) ky = height - 1;

for (int dx = -RW; dx <= RW; dx++) {

int kx = x + dx;

if (kx < 0) kx = 0;

if (kx > width - 1) kx = width - 1;

LinF\_Value\_Red += rgb\_input[ky][kx].rgbtRed \* matrix[dy + RH][dx + RW];

LinF\_Value\_Blue += rgb\_input[ky][kx].rgbtBlue \* matrix[dy + RH][dx + RW];

LinF\_Value\_Green += rgb\_input[ky][kx].rgbtGreen \* matrix[dy + RH][dx + RW];

}

}

LinF\_Value\_Red = LinF\_Value\_Red < 0 ? 0 : LinF\_Value\_Red;

LinF\_Value\_Red = LinF\_Value\_Red > 255 ? 255 : LinF\_Value\_Red;

rgb\_output[y][x].rgbtRed = LinF\_Value\_Red;

LinF\_Value\_Green = LinF\_Value\_Green < 0 ? 0 : LinF\_Value\_Green;

LinF\_Value\_Green = LinF\_Value\_Green > 255 ? 255 : LinF\_Value\_Green;

rgb\_output[y][x].rgbtGreen = LinF\_Value\_Green;

LinF\_Value\_Blue = LinF\_Value\_Blue < 0 ? 0 : LinF\_Value\_Blue;

LinF\_Value\_Blue = LinF\_Value\_Blue > 255 ? 255 : LinF\_Value\_Blue;

rgb\_output[y][x].rgbtBlue = LinF\_Value\_Blue;

}

}

}

// gauss omp

double\*\* gauss\_matrix\_parallel\_omp(int size) {

int RH = size / 2, RW = size / 2;

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

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

matrix[i] = new double[size];

double SUM = 0;

#pragma omp parallel for reduction(+:SUM)

for (int y = -RH; y <= RH; y++) {

for (int x = -RW; x <= RW; x++) {

int YK = y + RH;

int XK = x + RW;

double CF = (1 / (2 \* 3.14 \* pow(RH, 2))) \* exp(-1 \* (pow(x, 2) + pow(y, 2)) / (2 \* pow(RH, 2)));

matrix[YK][XK] = CF;

SUM += CF;

}

}

#pragma omp parallel for

for (int y = -RH; y <= RH; y++) {

for (int x = -RW; x <= RW; x++) {

int YK = y + RH;

int XK = x + RW;

matrix[YK][XK] /= SUM;

}

}

return matrix;

}

void gauss\_filter\_parallel\_omp(int height, int width, RGBTRIPLE\*\* rgb\_input, RGBTRIPLE\*\* rgb\_output, int ksize) {

int size = ksize \* ksize;

int RH = ksize / 2, RW = ksize / 2;

double\*\* matrix = gauss\_matrix\_parallel\_omp(ksize);

#pragma omp parallel for

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

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

int LinF\_Value\_Red = 0;

int LinF\_Value\_Blue = 0;

int LinF\_Value\_Green = 0;

for (int dy = -RH; dy <= RH; dy++) {

int ky = y + dy;

if (ky < 0) ky = 0;

if (ky > height - 1) ky = height - 1;

for (int dx = -RW; dx <= RW; dx++) {

int kx = x + dx;

if (kx < 0) kx = 0;

if (kx > width - 1) kx = width - 1;

LinF\_Value\_Red += rgb\_input[ky][kx].rgbtRed \* matrix[dy + RH][dx + RW];

LinF\_Value\_Blue += rgb\_input[ky][kx].rgbtBlue \* matrix[dy + RH][dx + RW];

LinF\_Value\_Green += rgb\_input[ky][kx].rgbtGreen \* matrix[dy + RH][dx + RW];

}

}

LinF\_Value\_Red = LinF\_Value\_Red < 0 ? 0 : LinF\_Value\_Red;

LinF\_Value\_Red = LinF\_Value\_Red > 255 ? 255 : LinF\_Value\_Red;

rgb\_output[y][x].rgbtRed = LinF\_Value\_Red;

LinF\_Value\_Green = LinF\_Value\_Green < 0 ? 0 : LinF\_Value\_Green;

LinF\_Value\_Green = LinF\_Value\_Green > 255 ? 255 : LinF\_Value\_Green;

rgb\_output[y][x].rgbtGreen = LinF\_Value\_Green;

LinF\_Value\_Blue = LinF\_Value\_Blue < 0 ? 0 : LinF\_Value\_Blue;

LinF\_Value\_Blue = LinF\_Value\_Blue > 255 ? 255 : LinF\_Value\_Blue;

rgb\_output[y][x].rgbtBlue = LinF\_Value\_Blue;

}

}

}

//gauss\_tbb

double\*\* gauss\_matrix\_parallel\_tbb(int size) {

int RH = size / 2, RW = size / 2;

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

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

matrix[i] = new double[size];

double SUM = 0;

tbb:task\_group g;

g.run([&] { for (long Y = -RH; Y <= RH; Y++) {

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

int YK = Y + RH;

int XK = X + RW;

double CF = (1 / (2 \* 3.14 \* pow(RH, 2))) \* exp(-1 \* (pow(X, 2) + pow(Y, 2)) / (2 \* pow(RH, 2)));

matrix[YK][XK] = CF;

SUM += CF;

}

};

});

g.wait();

tbb::parallel\_for(tbb::blocked\_range2d<double>(-RH, RH, -RW, RW), [&](tbb::blocked\_range2d<double> r) {

for (size\_t Y = r.rows().begin(); Y <= r.rows().end(); Y++) {

for (size\_t X = r.cols().begin(); X <= r.cols().end(); X++) {

int YK = Y + RH;

int XK = X + RW;

matrix[YK][XK] /= SUM;

}

}

});

return matrix;

}

void gauss\_filter\_parallel\_tbb(int height, int width, RGBTRIPLE\*\* rgb\_input, RGBTRIPLE\*\* rgb\_output, int ksize) {

int size = ksize \* ksize;

int RH = ksize / 2, RW = ksize / 2;

double\*\* matrix = gauss\_matrix\_parallel\_tbb(ksize);

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++) {

int LinF\_Value\_Red = 0;

int LinF\_Value\_Blue = 0;

int LinF\_Value\_Green = 0;

for (int dy = -RH; dy <= RH; dy++) {

int ky = y + dy;

if (ky < 0) ky = 0;

if (ky > height - 1) ky = height - 1;

for (int dx = -RW; dx <= RW; dx++) {

int kx = x + dx;

if (kx < 0) kx = 0;

if (kx > width - 1) kx = width - 1;

LinF\_Value\_Red += rgb\_input[ky][kx].rgbtRed \* matrix[dy + RH][dx + RW];

LinF\_Value\_Blue += rgb\_input[ky][kx].rgbtBlue \* matrix[dy + RH][dx + RW];

LinF\_Value\_Green += rgb\_input[ky][kx].rgbtGreen \* matrix[dy + RH][dx + RW];

}

}

LinF\_Value\_Red = LinF\_Value\_Red < 0 ? 0 : LinF\_Value\_Red;

LinF\_Value\_Red = LinF\_Value\_Red > 255 ? 255 : LinF\_Value\_Red;

rgb\_output[y][x].rgbtRed = LinF\_Value\_Red;

LinF\_Value\_Green = LinF\_Value\_Green < 0 ? 0 : LinF\_Value\_Green;

LinF\_Value\_Green = LinF\_Value\_Green > 255 ? 255 : LinF\_Value\_Green;

rgb\_output[y][x].rgbtGreen = LinF\_Value\_Green;

LinF\_Value\_Blue = LinF\_Value\_Blue < 0 ? 0 : LinF\_Value\_Blue;

LinF\_Value\_Blue = LinF\_Value\_Blue > 255 ? 255 : LinF\_Value\_Blue;

rgb\_output[y][x].rgbtBlue = LinF\_Value\_Blue;

}

}

}

);

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_КОНЕЦ Фильтр Гаусса \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//

void((\*functions[24]))(double\*\*, double\*\*, int, int) = {

sum\_posled,

sum\_par\_omp,

sum\_par\_omp\_sections,

sum\_par\_tbb\_task\_group,

sum\_par\_tbb\_par\_for\_lambda,

sum\_par\_for\_class,

mul\_posled,

mul\_par\_omp,

mul\_par\_omp\_sections,

mul\_par\_tbb\_task\_group,

mul\_tbb\_par\_for\_lambda,

mul\_par\_for\_class,

sum\_all\_posled,

sum\_all\_par\_omp,

sum\_all\_par\_omp\_sections,

sum\_all\_par\_tbb\_task\_group,

sum\_all\_tbb\_par\_reduce\_lambda,

sum\_all\_par\_reduce\_class,

min\_max\_posled,

min\_max\_par\_omp,

min\_max\_par\_omp\_sections,

min\_max\_par\_tbb\_task\_group,

min\_max\_tbb\_par\_for\_lambda,

min\_max\_par\_reduce\_class

};

const char\* functions\_name[24] = {

"Сложение последовательно",

"Сложение omp for",

"Сложение omp sections",

"Сложение tbb task group",

"Сложение tbb for lambda",

"Сложение tbb for class",

"Умножение последовательно",

"Умножение omp for",

"Умножение omp sections",

"Умножение tbb task group",

"Умножение tbb for lambda",

"Умножение tbb for class",

"Сумма последовательно",

"Сумма omp for",

"Сумма omp sections",

"Сумма tbb task group",

"Сумма tbb for lambda",

"Сумма tbb reduce class",

"МинМакс последовательно",

"МинМакс omp for",

"МинМакс omp sections",

"МинМакс tbb task group",

"МинМакс tbb for lambda",

"МинМакс tbb reduce class",

};

const char\* images[3] = {

"1280x720.bmp",

"1600x900.bmp",

"2580x1080.bmp",

};

const char\* images\_out[3] = {

"1280x720\_out.bmp",

"1600x900\_out.bmp",

"2580x1080\_out.bmp",

};

void((\*filters[6]))(int, int, RGBTRIPLE\*\*, RGBTRIPLE\*\*, int) = {

gauss\_filter\_posled, gauss\_filter\_parallel\_omp, gauss\_filter\_parallel\_tbb, median\_posled, median\_parallel\_omp, median\_parallel\_tbb

};

const char\* NAMES[6] = {

"Гаусс последовательно",

"Гаусс OMP",

"Гаусс for TBB",

"Медианный последовательно",

"Медианный OMP",

"Медианный for TBB"

};

void((\*texture[3]))(RGBTRIPLE\*\*&, BITMAPFILEHEADER&, BITMAPINFOHEADER&, const char\*, int) = {

texture\_posled, texture\_omp, texture\_tbb

};

const char\* texture\_name[3] = {

"Texture posled",

"Texture omp",

"Texture tbb",

};

const char\* images\_n3[2] = {

"1600x900.bmp",

"2580x1080.bmp",

};

void sort(double\* A, int s) {

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

for (int j = 0; j < s - i - 1; j++) {

if (A[j + 1] < A[j]) {

swap(A[j], A[j + 1]);

}

}

}

}

//Задание 1

/\*

int main() {

setlocale(LC\_ALL, "en\_US.UTF-8");

ofstream n1("n1.txt");

for (int function = 0; function <= 23; function++) {

cout << functions\_name[function] << endl;

n1 << functions\_name[function] << endl;

for (int th = 2; th <= 4; th++) {

cout << "Thread-" << th << ": ";

n1 << "П-" << th << ": ";

for (int sizeA = 3000; sizeA <= 6000; sizeA += 1000) {

omp\_set\_num\_threads(th);

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

double\*\* A;

A = new double\* [sizeA];

A[0] = new double[sizeA \* sizeA];

for (long i = 1; i < sizeA; i++)

A[i] = &A[0][i \* sizeA];

for (long i = 0; i < sizeA; i++)

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

A[i][j] = sin(i) + cos(j / 2.3) \* 5;

double\*\* B;

B = new double\* [sizeA];

B[0] = new double[sizeA \* sizeA];

for (int i = 1; i < sizeA; i++)

B[i] = &B[0][i \* sizeA];

for (long i = 0; i < sizeA; i++)

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

B[i][j] = sin(i) + cos(j / 2.3) \* 3;

double\* median = new double[10]; // 100

for (int run = 0; run < 10; run++) { // 100

double start\_time = omp\_get\_wtime();

functions[function](A, B, sizeA, sizeA);

median[run] = omp\_get\_wtime() - start\_time;

}

sort(median, 10); // 100

cout << median[4] \* 1000 << endl; // 49

n1 << median[4] \* 1000 << " "; // 49

delete A[0];

delete[]A;

delete B[0];

delete[]B;

}

cout << endl;

n1 << endl;

if (function == 0 || function == 6 || function == 12 || function == 18)

break;

}

cout << endl;

n1 << endl;

}

return 0;

}

\*/

//Задание 2

/\*

int main() {

setlocale(LC\_ALL, "en\_US.UTF-8");

ofstream n2("n2.txt");

for (int function = 0; function < 6; function++) {

cout << NAMES[function] << endl;

n2 << NAMES[function] << endl;

for (int th = 2; th <= 4; th++) {

for (int k = 5; k <= 9; k += 2) {

cout << "Thread-" << th << " K" << k << ": ";

n2 << "П-" << th << " K" << k << ": ";

for (int image = 0; image < 3; image++) {

omp\_set\_num\_threads(th);

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

double\* median = new double[1]; // 20

RGBTRIPLE\*\* rgb\_input, \*\* rgb\_output;

BITMAPFILEHEADER header;

BITMAPINFOHEADER bmiHeader;

BMPRead(rgb\_input, header, bmiHeader, images[image]);

rgb\_output = new RGBTRIPLE \* [bmiHeader.biHeight];

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

rgb\_output[i] = new RGBTRIPLE[bmiHeader.biWidth];

}

for (int run = 0; run < 1; run++) { // 20

double start\_time = omp\_get\_wtime();

filters[function](bmiHeader.biHeight, bmiHeader.biWidth, rgb\_input, rgb\_output, k);

median[run] = omp\_get\_wtime() - start\_time;

}

BMPWrite(rgb\_output, bmiHeader.biWidth, bmiHeader.biHeight, images\_out[image]);

sort(median, 1); // 20

cout << median[0] \* 1000 << " "; // 9

n2 << median[0] \* 1000 << " "; // 9

}

cout << endl;

n2 << endl;

}

//Res << std::endl << std::endl;

if (function == 0 || function == 3)

break;

n2 << endl << endl;

}

}

return 0;

}

\*/

//Задание 3

int main() {

setlocale(LC\_ALL, "en\_US.UTF-8");

ofstream n3("n3.txt");

for (int filter = 0; filter < 3; filter++) {

cout << texture\_name[filter] << endl;

n3 << texture\_name[filter] << std::endl;

for (int th = 2; th <= 4; th++) {

for (int image = 0; image < 2; image++) {

for (int k = 5; k <= 9; k += 2) {

cout << "k = " << k << endl;

n3 << "k = " << k << std::endl;

for (int image = 0; image < 2; image += 1) {

RGBTRIPLE\*\* rgb\_input;

BITMAPFILEHEADER header;

BITMAPINFOHEADER bmiHeader;

omp\_set\_num\_threads(th);

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

double\* median = new double[4]; // 10

for (int run = 0; run < 4; run++) { // 10

double start\_time = omp\_get\_wtime();

texture[filter](rgb\_input, header, bmiHeader, images\_n3[image], k);

median[run] = omp\_get\_wtime() - start\_time;

}

sort(median, 4); // 10

cout << median[2] \* 1000 << " "; // 4

n3 << median[2] \* 1000 << " "; // 4

//system("pause");

}

cout << std::endl;

n3 << std::endl;

}

}

if (filter == 0)

break;

cout << std::endl << std::endl;

n3 << std::endl << std::endl;

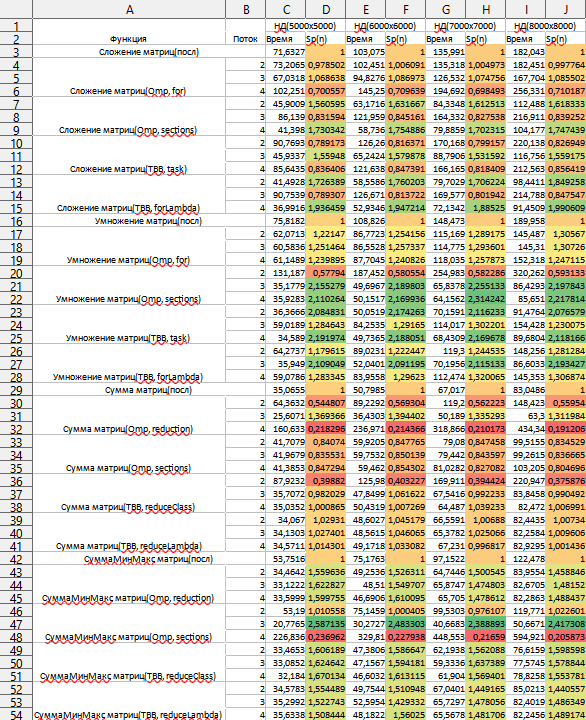
}

}

return 0;

}

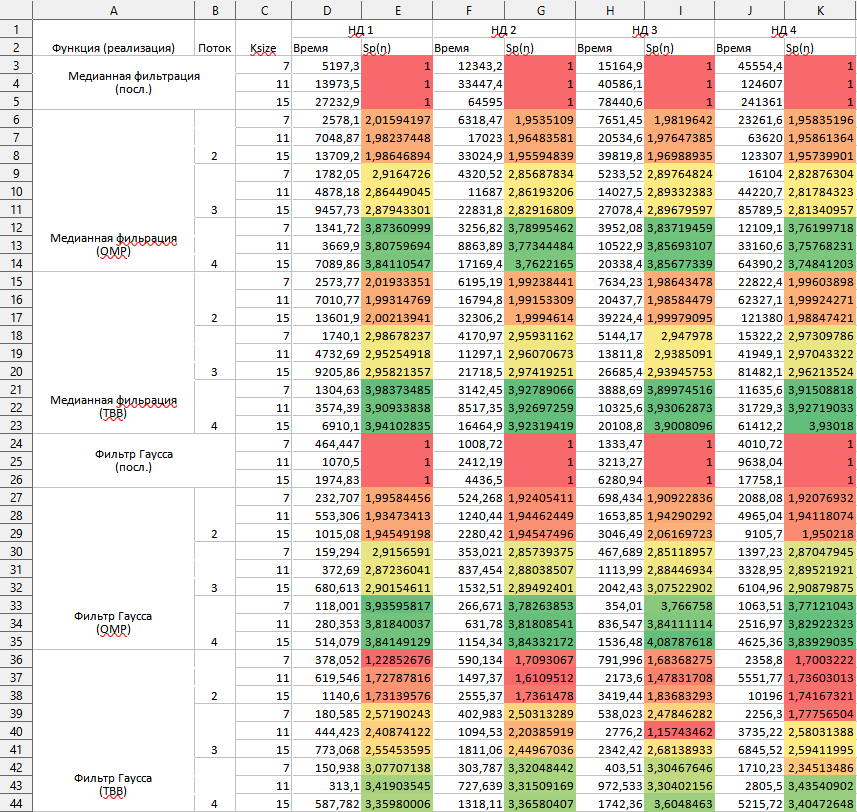
Данные полученные в ходе выполнения программы представлены в Рисунок 1.

Рисунок 1 – результаты выполнения 1-ой программы

Вывод: наибольшее ускорения для всех функций получается при реализации omp sections, реализация с помощью intel oneTBB дает ускорение, но не существенное.

Задание 2:

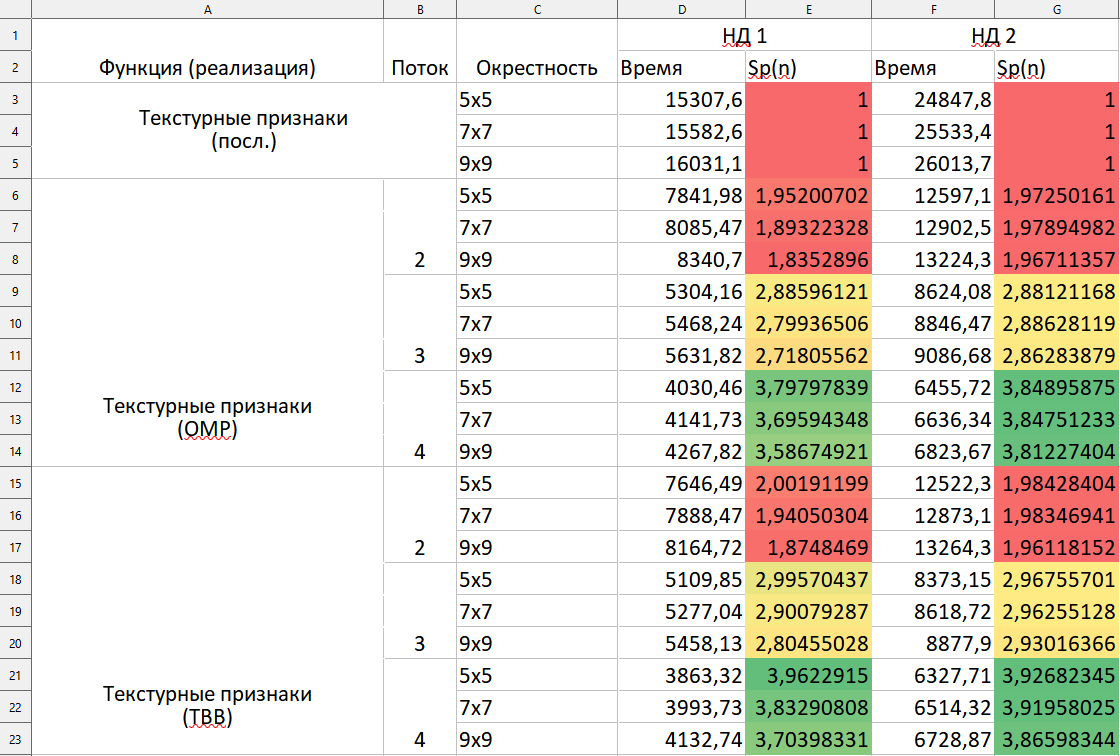
Данные полученные в ходе выполнения программы представлены в Рисунок 2.

Рисунок 2 – результаты выполнения 2-ой программы

Вывод: для медианной фильтрации наибольшее ускорение имеет функция, реализованная с помощью Intel oneTBB, а для фильтра Гаусса наоборот.

Задание 3.

Данные полученные в ходе выполнения программы представлены в Рисунок 3.

Рисунок 3 – результаты выполнения 3-ей программы

Вывод: наибольшее ускорение показала функция, реализованная с помощью Intel oneTBB, функция, реализованная с помощью OMP, имеет ускорение чуть ниже.

# ВЫВОД

Изучил различные виды реализации алгоритмов обработки массивов с использованием возможностей Intel oneTBB. Изучил алгоритмы обработки изображения с использованием параллельных реализаций, учитывающих разные подходы к оптимизации обработки (на примере фильтра Гаусса, Медианного фильтра). Разработал алгоритмы вычисления текстурных признаков.

# ОТВЕТЫ НА ВОПРОСЫ

1. Охарактеризуйте Intel oneTBB.

Intel oneTBB (Intel oneAPI Threading Building Blocks) - это библиотека параллельного программирования, разработанная компанией Intel. Она предоставляет программистам удобные инструменты для создания эффективных и масштабируемых параллельных приложений.

Intel oneTBB предоставляет высокоуровневые абстракции и шаблоны, которые позволяют разработчикам легко создавать параллельные программы. Библиотека предлагает различные алгоритмы и контейнеры, которые автоматически распределяют работу между доступными ядрами процессора или потоками выполнения. Это позволяет эффективно использовать многоядерные системы и ускоряет выполнение задач.

Intel oneTBB поддерживает различные платформы и операционные системы, включая Windows, Linux и macOS. Она также интегрируется с другими инструментами Intel для параллельного программирования, такими как Intel Advisor и Intel VTune Profiler, что облегчает профилирование и оптимизацию параллельных приложений.

Благодаря своей простоте использования и эффективности, Intel oneTBB является популярным выбором для разработчиков, которые хотят создавать быстрые и масштабируемые параллельные приложения на платформе Intel.

2. Как можно узнать максимальное количество потоков по умолча-

нию которые будут созданы по умолчанию в oneTBB?

В Intel oneTBB максимальное количество потоков, которые будут созданы по умолчанию, зависит от конфигурации вашей системы. Оно определяется количеством доступных ядер процессора и настроек операционной системы.

Вы можете узнать максимальное количество потоков, поддерживаемых вашей системой, с помощью функции tbb::info::default\_concurrency(). Эта функция возвращает количество потоков, которые будут созданы по умолчанию при инициализации планировщика задач в oneTBB.

3. Каким образом в Intel oneTBB происходит управление потоками?

В Intel oneTBB управление потоками осуществляется с помощью планировщика задач (task scheduler). Планировщик задач отвечает за распределение работы между доступными потоками выполнения и эффективное использование ресурсов процессора.

При использовании Intel oneTBB вы можете создавать задачи, которые представляют собой небольшие единицы работы. Планировщик задач автоматически распределяет эти задачи между доступными потоками выполнения. Каждый поток выполняет свою часть работы, что позволяет достичь параллелизма и ускорить выполнение программы.

Планировщик задач в Intel oneTBB динамически адаптируется к количеству доступных ядер процессора и нагрузке на систему. Он может создавать и уничтожать потоки выполнения по мере необходимости, чтобы эффективно использовать ресурсы процессора и избежать излишней нагрузки.

Кроме того, Intel oneTBB предоставляет различные средства для управления выполнением задач, такие как ограничение числа одновременно выполняемых задач (task throttling), приоритеты задач и возможность ожидания завершения выполнения задач.

В целом, Intel oneTBB обеспечивает удобный и эффективный механизм управления потоками, позволяющий разработчикам создавать параллельные приложения, которые масштабируются и эффективно используют ресурсы процессора.

4. Дайте понятие диапазона (Range), какие типы моделей диапазона

доступны в oneTBB?

В Intel oneTBB диапазон (Range) представляет собой абстракцию, которая определяет набор элементов, над которыми можно выполнять операции параллельно. Диапазон предоставляет удобный интерфейс для разделения работы между потоками выполнения и координирования их работы.

В oneTBB доступны различные типы моделей диапазона, которые предоставляют разные способы представления и обработки данных. Некоторые из них включают:

tbb::blocked\_range: Это наиболее общая модель диапазона, которая представляет непрерывный диапазон значений. Она разделяет диапазон на блоки фиксированного размера и распределяет их между потоками выполнения.

tbb::blocked\_range2d: Эта модель представляет двумерный диапазон значений. Она разделяет диапазон на блоки фиксированного размера в обоих измерениях и распределяет их между потоками выполнения.

tbb::blocked\_range3d: Эта модель представляет трехмерный диапазон значений. Она разделяет диапазон на блоки фиксированного размера в трех измерениях и распределяет их между потоками выполнения.

tbb::parallel\_for: Это модель диапазона, которая позволяет выполнять циклы параллельно. Она автоматически разделяет итерации цикла между потоками выполнения.

tbb::parallel\_reduce: Это модель диапазона, которая позволяет выполнять параллельную редукцию. Она автоматически разделяет данные и операции редукции между потоками выполнения.

5. Охарактеризуйте oneTBB parallel\_for.

oneTBB parallel\_for - это модель диапазона, предоставляемая библиотекой Intel oneTBB, которая позволяет выполнять циклы параллельно. Она автоматически разделяет итерации цикла между доступными потоками выполнения, что позволяет эффективно использовать многопоточность и ускорить выполнение цикла.

Основные характеристики и возможности parallel\_for включают:

Параллельное выполнение цикла: parallel\_for автоматически разделяет итерации цикла между потоками выполнения, что позволяет выполнять их параллельно. Это особенно полезно для циклов, в которых итерации независимы друг от друга.

Динамическое распределение работы: parallel\_for динамически распределяет итерации между потоками выполнения во время выполнения. Это позволяет более равномерно распределить нагрузку и обеспечить эффективное использование ресурсов процессора.

Удобный интерфейс: parallel\_for предоставляет удобный интерфейс для определения цикла и его итераций. Разработчики могут использовать функциональные объекты, лямбда-выражения или функции для определения операций, выполняемых в каждой итерации.

Поддержка различных типов итераций: parallel\_for поддерживает различные типы итераций, включая целочисленные, дробные и пользовательские типы. Это позволяет применять parallel\_for к различным видам циклов.

Возможность управления выполнением: parallel\_for предоставляет возможность управлять выполнением цикла, включая возможность ограничения числа одновременно выполняемых итераций (task throttling), установки приоритетов и ожидания завершения выполнения цикла.

6. Поясните аспекты управления распределением работы oneTBB

parallel\_for.

В Intel oneTBB parallel\_for предоставляет несколько аспектов управления распределением работы между потоками выполнения. Эти аспекты позволяют разработчикам более гибко контролировать и оптимизировать выполнение параллельного цикла. Некоторые из аспектов управления включают:

Размер блока итераций: Разработчики могут указать размер блока итераций, который будет обрабатываться одним потоком выполнения перед перераспределением работы. Это позволяет более точно настроить распределение работы в зависимости от характера задачи и доступных ресурсов.

Ограничение числа одновременно выполняемых итераций: parallel\_for предоставляет возможность ограничить количество одновременно выполняемых итераций. Это может быть полезно, когда требуется контролировать нагрузку на систему или ограничить использование ресурсов.

Установка приоритетов: Разработчики могут устанавливать приоритеты для различных итераций цикла. Это позволяет определить относительную важность и срочность выполнения каждой итерации и влиять на порядок их выполнения.

Ожидание завершения выполнения: parallel\_for предоставляет возможность явно ожидать завершения выполнения цикла перед продолжением работы. Это может быть полезно, когда требуется синхронизация или когда результаты выполнения цикла необходимы для дальнейшей обработки.

Пользовательский контроль: Разработчики могут использовать пользовательские функции и объекты для более тонкой настройки и контроля распределения работы. Это позволяет реализовать специфическую логику распределения или выполнения итераций цикла.

7. Охарактеризуйте oneTBB parallel\_for\_each.

oneTBB parallel\_for\_each - это модель диапазона, предоставляемая библиотекой Intel oneTBB, которая позволяет выполнять операции над элементами контейнера параллельно. Она автоматически разделяет элементы контейнера между доступными потоками выполнения, что позволяет эффективно использовать многопоточность и ускорить выполнение операций.

Основные характеристики и возможности parallel\_for\_each включают:

Параллельное выполнение операций: parallel\_for\_each автоматически разделяет элементы контейнера между потоками выполнения, что позволяет выполнять операции над ними параллельно. Это особенно полезно для операций, которые могут быть выполнены независимо для каждого элемента.

Динамическое распределение работы: parallel\_for\_each динамически распределяет элементы контейнера между потоками выполнения во время выполнения. Это позволяет более равномерно распределить нагрузку и обеспечить эффективное использование ресурсов процессора.

Удобный интерфейс: parallel\_for\_each предоставляет удобный интерфейс для определения операции, выполняемой над каждым элементом контейнера. Разработчики могут использовать функциональные объекты, лямбда-выражения или функции для определения операции.

Поддержка различных типов контейнеров: parallel\_for\_each поддерживает различные типы контейнеров, включая стандартные контейнеры STL, пользовательские контейнеры и диапазоны oneTBB. Это позволяет применять parallel\_for\_each к различным видам данных.

Возможность управления выполнением: parallel\_for\_each предоставляет возможность управлять выполнением операций, включая возможность ограничения числа одновременно выполняемых операций (task throttling), установки приоритетов и ожидания завершения выполнения.

8. В чем заключается особенность использования oneTBB

parallel\_invoke?

Особенность использования parallel\_invoke в Intel oneTBB заключается в возможности параллельного выполнения нескольких функций или операций одновременно. parallel\_invoke позволяет эффективно использовать многопоточность и ускорить выполнение программы.

Основные особенности и преимущества parallel\_invoke включают:

Параллельное выполнение: parallel\_invoke позволяет запускать несколько функций или операций параллельно, что позволяет эффективно использовать доступные ресурсы процессора и ускорить выполнение программы.

Автоматическое распределение работы: parallel\_invoke автоматически распределяет функции или операции между доступными потоками выполнения. Это позволяет более равномерно распределить нагрузку и обеспечить эффективное использование ресурсов.

Удобный интерфейс: parallel\_invoke предоставляет удобный интерфейс для определения функций или операций, которые нужно выполнить параллельно. Разработчики могут передавать функции или операции в parallel\_invoke в виде аргументов.

Блокирующее ожидание завершения: parallel\_invoke блокирует выполнение текущего потока до завершения всех переданных функций или операций. Это позволяет синхронизировать выполнение программы и продолжить работу только после завершения всех параллельных операций.

Поддержка ошибок: parallel\_invoke обрабатывает исключения, возникающие во время выполнения переданных функций или операций. Если одна из функций или операций выбрасывает исключение, parallel\_invoke прекращает выполнение остальных функций и передает исключение наверх для обработки.

9. Что из себя представляет oneTBB task\_group?

oneTBB task\_group представляет собой механизм для управления группой задач в библиотеке Intel oneTBB. Он позволяет разработчикам создавать и управлять набором задач, которые могут быть выполнены параллельно и синхронно.

Основные особенности и функциональность task\_group включают:

Создание и добавление задач: С помощью task\_group можно создавать и добавлять задачи в группу. Задачи могут быть представлены в виде функциональных объектов, лямбда-выражений или функций.

Параллельное выполнение задач: task\_group позволяет параллельно выполнять задачи внутри группы. Это позволяет эффективно использовать многопоточность и ускорить выполнение программы.

Синхронизация выполнения: task\_group предоставляет методы для синхронизации выполнения задач в группе. Например, метод wait() блокирует текущий поток до завершения всех задач в группе.

Обработка исключений: task\_group обрабатывает исключения, возникающие во время выполнения задач. Если одна из задач выбрасывает исключение, task\_group прекращает выполнение остальных задач и передает исключение наверх для обработки.

Управление завершением: task\_group предоставляет методы для проверки состояния выполнения задач в группе, такие как is\_empty() и is\_canceling(). Это позволяет контролировать и управлять процессом выполнения задач.

10. Что необходимо сделать ожидание завершения задач при работе

с oneTBB task\_group?

Для ожидания завершения задач при работе с task\_group в Intel oneTBB необходимо вызвать метод wait() на объекте task\_group. Этот метод блокирует текущий поток выполнения до тех пор, пока все задачи в группе не завершатся.

11. Приведите пример использования task\_group и parallel\_invoke.

#include <tbb/task\_group.h>

#include <iostream>

void Task1() {

std::cout << "Task 1 executed" << std::endl;

}

void Task2() {

std::cout << "Task 2 executed" << std::endl;

}

void Task3() {

std::cout << "Task 3 executed" << std::endl;

}

int main() {

tbb::task\_group taskGroup;

taskGroup.run(Task1);

taskGroup.run(Task2);

tbb::parallel\_invoke(

[&]() { taskGroup.run(Task3); },

[&]() { std::cout << "Parallel task executed" << std::endl; }

);

taskGroup.wait();

std::cout << "All tasks completed" << std::endl;

return 0;

}

В этом примере мы создаем task\_group и добавляем в него три задачи: Task1, Task2 и Task3. Затем мы используем parallel\_invoke для параллельного выполнения двух задач: Task3 и анонимной лямбда-функции, которая выводит сообщение "Parallel task executed". После этого мы вызываем wait() для ожидания завершения всех задач в task\_group. Наконец, мы выводим сообщение "All tasks completed" для подтверждения завершения всех задач.

12. Охарактеризуйте oneTBB parallel\_reduce.

oneTBB parallel\_reduce - это модель диапазона, предоставляемая библиотекой Intel oneTBB, которая позволяет выполнять операции сокращения (reduction) над элементами результат сокращения.

Основные характеристики и возможности parallel\_reduce включают:

Параллельное выполнение операций сокращения: parallel\_reduce автоматически разделяет элементы диапазона между операций сокращения.

Поддержка пользовательских операций сокращения: parallel\_reduce позволяет разработчикам определить свою собственную операцию сокращения, которая будет применяться к элементам диапазона. Сокращения, такие как сумма, произведение, минимум, максимум и другие.

Гибкость и эффективность: parallel\_reduce обеспечивает гибкость в выборе типа диапазона и операции сокращения, а также эффективное использование ресурсов процессора и parallel\_reduce в Intel oneTBB позволяет разработчикам легко и эффективно выполнять операции сокращения над элементами диапазона, ускоряя выполнение программы и повышая производительность.

13. Дайте описание класса для реализации редукционной операции в

Intel oneTBB.

В Intel oneTBB для реализации редукционной операции можно использовать класс tbb::combinable.

Класс tbb::combinable представляет собой контейнер, который позволяет хранить и обрабатывать значения, связанные с редукционной операцией. Он обеспечивает механизм автоматического разделения данных и выполнения операций сокращения между потоками выполнения.

Основные методы класса tbb::combinable включают:

local(): Возвращает ссылку на локальное хранилище значения для текущего потока выполнения. Если значение еще не было установлено для данного потока, будет создано новое значение.

combine(): Выполняет операцию сокращения для всех локальных значений, хранящихся в разных потоках выполнения. Результат сокращения будет сохранен в глобальном значении.

clear(): Очищает все локальные значения, хранящиеся в разных потоках выполнения.

14. Каким образом возможно реализовать выполнение разнородных

редукционных операций одновременно?

Для выполнения разнородных редукционных операций одновременно в Intel oneTBB можно использовать класс tbb::parallel\_deterministic\_reduce. Этот класс позволяет выполнять несколько редукционных операций одновременно и гарантирует, что результаты будут воспроизводимы и детерминированы.

Для реализации выполнения разнородных редукционных операций одновременно с помощью tbb::parallel\_deterministic\_reduce необходимо выполнить следующие шаги:

Определить структуру данных, которая будет хранить результаты разнородных редукционных операций. Например, можно использовать структуру std::tuple для хранения результатов.

Создать класс-функтор, который будет выполнять редукционную операцию для каждого элемента структуры данных. Класс-функтор должен быть перегружен для каждой редукционной операции.

Создать экземпляр класса tbb::parallel\_deterministic\_reduce и передать ему структуру данных и класс-функторы для каждой редукционной операции.

Вызвать метод run() для выполнения редукционных операций одновременно.

struct Sum {

int operator()(int a, int b) const {

return a + b;

}

};

struct Product {

int operator()(int a, int b) const {

return a \* b;

}

};

int main() {

std::tuple<int, int> result;

tbb::parallel\_deterministic\_reduce(

tbb::blocked\_range<int>(0, 10),

result,

[](const tbb::blocked\_range<int>& range, std::tuple<int, int>& local\_result) {

int& sum = std::get<0>(local\_result);

int& product = std::get<1>(local\_result);

for (int i = range.begin(); i != range.end(); ++i) {

sum += i;

product \*= i;

}

},

Sum(),

Product()

);

int finalSum = std::get<0>(result);

int finalProduct = std::get<1>(result);

std::cout << "Final sum: " << finalSum << std::endl;

std::cout << "Final product: " << finalProduct << std::endl;

return 0;

}