Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

«Нижегородский государственный университет им. Н.И. Лобачевского»

**Институт информационных технологий, математики и механики**

**Отчет**

по лабораторной работе №3

**«Линейная фильтрация изображений**

**(блочное разбиение). Ядро Гаусса 3x3»**

**Выполнил:**

студент группы 381608

Блинцов С.Д.

**Проверила:**

Кустикова В.Д.

Нижний Новгород

2018

**Содержание**

[Постановка задачи 2](#_Toc531367741)

[Метод решения 3](#_Toc531367742)

[Схема распараллеливания 4](#_Toc531367743)

[Описание программной реализации 5](#_Toc531367744)

[Подтверждение корректности 6](#_Toc531367745)

[Результаты экспериментов 7](#_Toc531367746)

[Заключение 8](#_Toc531367747)

[Приложение 9](#_Toc531367748)

# Постановка задачи

Цель – получение навыков распараллеливания задач компьютерной графики средствами библиотеки MPI.

Критерии эффективности разработки:

* результаты тестирования корректности работы алгоритма (сравнение результатов работы линейной и параллельной версии программы)
* производительность параллельной версии программы выше, чем линейной

Требуется:

* Исходное изображение должно быть свёрнуто с применением ядра Гаусса 3х3
* Должна использоваться библиотека MPI
* Схема распараллеливания не зависит от количества предоставленных ресурсов

# Метод решения

Концепция: Задача была решена методом блочного разбиения исходного изображения.

1. Формируется ядро Гаусса 3х3;
2. Находится оптимальное количество блоков, на которое разбивается исходное изображение (находятся простые делители числа процессов, затем на их основе производится оценка оптимальности разбиения);
3. Изображение разбивается на блоки и отсылается от 0 процесса всем остальным;
4. Ядро Гаусса применяется к блокам изображения;
5. Обработанные блоки отсылаются на 0 процесс;
6. 0 процесс производит компоновку блоков для получения результирующего изображения.

# Схема распараллеливания

Нулевой процесс отсылает блоки исходного изображения всем остальным процесса.

Обработка изображения происходит на всех процессах.

Затем обработанные блоки отсылаются обратно на нулевой процесс и он же осуществляет финальную компоновку результирующего изображения.

# Описание программной реализации

**Руководство пользователя**

Программа реализует линейную и параллельную версии алгоритма свёртки с ядром Гаусса.

Для начала работы с программой необходимо установить библиотеки MPI и OpenCV 2.4.6.13

Запуск программы возможен только из интерфейса командной строки, посредством введения в неё команды: <pathToMPI>/mpiexec [–n <нужное кол-во процессов>] ./<Путь к скомпилированному файлу> <путь к файлу изображения> .

**Руководство программиста**

Для изменения размера и характеристики σ необходимо соответственно изменить переменные kernel\_size (должна быть положительной и нечётной) и sigma.

Код программы можно просмотреть в разделе «[Приложение](#_Приложение)».

# Подтверждение корректности

Для подтверждения корректности в программе предусмотрена автоматическая проверка полного соответствия результата работы линейной версии алгоритма результату работы параллельной версии алгоритма.

Также предусмотрена возможность визуально сравнить исходной изображение, изображение полученное линейной версией алгоритма программы и изображение полученное в результате работы параллельной версии алгоритма.

# Результаты экспериментов

По данным экспериментов видно, что меняя количество процессов на которых исполняется программа – меняется время работы алгоритма. При определённом количестве процессов накладные расходы на пересылку данных нивелируются сложностью вычислений и виден прирост производительности параллельной версии программы по сравнению с линейной. Также можно заметить, что от изменения числа процессов не меняется результат работы программы.

# Заключение

Выполнив данную лабораторную работу мы получили навыки работы с библиотекой MPI: подключение библиотеки к проекту, использование средств библиотеки, запуск приложений использующих данную библиотеку. Был получен опыт оптимального разбиения двухмерных данных на блоки.

Также нам удалось убедиться в эффективности работы с библиотекой MPI в том случае когда пересылка большого объема данных нивелируется сложностью вычислений.

# Приложение

#define \_USE\_MATH\_DEFINES

#include <opencv2/core/core.hpp>

#include <opencv2/highgui/highgui.hpp>

#include <mpi.h>

#include <math.h>

#include <stdlib.h>

#include <iostream>

#include <vector>

using namespace cv;

unsigned char clamp(double val) {

if (val < 0)

val = 0;

else if(val > (double)UCHAR\_MAX)

val = (double)UCHAR\_MAX;

return (unsigned char)val;

}

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

{

if (argc != 2) {

std::cout << " Usage: display\_image ImageToLoadAndDisplay" << std::endl;

return -1;

}

Mat \*image = nullptr;

Mat \*lin\_image = nullptr;

Mat \*res\_image = nullptr;

int img\_width = 0;

int img\_height = 0;

int img\_size = 0;

int kernel\_size = 3;

int center = kernel\_size / 2;

double sigma = 0.5f;

unsigned char \*buf = nullptr;

int size = 0;

double start\_time = 0;

double end\_time = 0;

MPI\_Init(&argc, &argv);

int proc\_num = 0;

int proc\_rank = 0;

int height\_tiles = 0;

int width\_tiles = 0;

double \*data = nullptr;

MPI\_Comm\_size(MPI\_COMM\_WORLD, &proc\_num);

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

if (proc\_rank == 0) {

image = new Mat();

\*image = imread(argv[1], IMREAD\_GRAYSCALE);

if (!image->data) {

std::cout << "Could not open or find the image" << std::endl;

return -1;

}

img\_width = image->cols;

img\_height = image->rows;

img\_size = img\_height \* img\_width;

lin\_image = new Mat(img\_height, img\_width, CV\_8UC1);

res\_image = new Mat(img\_height, img\_width, CV\_8UC1);

// считаем кернел ядро фильтра гаусса

data = new double[kernel\_size \* kernel\_size];

for (int i = -center; i <= center; i++) {

for (int j = -center; j <= center; j++) {

data[(i + center) + kernel\_size \* (j + center)] = pow(sqrt(2 \* M\_PI) \* sigma, -1) \* exp(-(pow(i, 2) + pow(j, 2)) / (2 \* pow(sigma, 2)));

}

}

start\_time = MPI\_Wtime();

// делаем свёртку с этим ядром

for (int i = 0; i < img\_width; i++) {

for (int j = 0; j < img\_height; j++) {

lin\_image->data[i + j \* img\_width] = 0;

for (int k\_i = -center; k\_i <= center; k\_i++) {

for (int k\_j = -center; k\_j <= center; k\_j++) {

int new\_i = i - k\_i;

new\_i = new\_i > 0 ? new\_i : 0;

new\_i = new\_i < img\_width ? new\_i : (img\_width - 1);

int new\_j = j - k\_j;

new\_j = new\_j > 0 ? new\_j : 0;

new\_j = new\_j < img\_height ? new\_j : (img\_height - 1);

double add = image->data[new\_i + new\_j \* img\_width] \* data[(k\_i + center) + (k\_j + center) \* kernel\_size];

lin\_image->data[i + j \* img\_width] = clamp(lin\_image->data[i + j \* img\_width] + add);

}

}

}

}

end\_time = MPI\_Wtime();

std::cout << "Line time: " << end\_time - start\_time << std::endl;

// считаем оптимальные блоки на которые нужно разбить изображение

std::vector<int> mults;

int tmp = proc\_num;

int mul = 1;

for (int i = 2; i < tmp;) {

if ((tmp / mul) % i == 0) {

mults.push\_back(i);

mul \*= i;

}

else {

i++;

}

}

if (!mults.empty()) {

int sum = 0;

for (auto &el : mults) {

sum += el;

}

float sumDiv2 = (float)sum / 2;

int sum1 = 0;

int i = 0;

for (; i < mults.size(); i++) {

if (sum1 + mults[i] < sumDiv2) {

sum1 += mults[i];

}

else {

i--;

break;

}

}

if (fabs((float)sum1 - sumDiv2) >= fabs((float)sum1 + (float)mults[i + 1] - sumDiv2)) {

sum1 += mults[++i];

}

height\_tiles = 1;

width\_tiles = 1;

if (mults.size() != 2) {

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

height\_tiles \*= mults[j];

for (int j = i + 1; j < mults.size(); j++)

width\_tiles \*= mults[j];

}

else {

height\_tiles = mults[0];

width\_tiles = mults[1];

}

}

else {

height\_tiles = proc\_num;

width\_tiles = 1;

}

start\_time = MPI\_Wtime();

// вычисляем какого размера блоки мы должны отослать и отсылаем их, также отсылаем размеры этих блоков и ядро фильтра

for (int i = 0; i < width\_tiles; i++) {

for (int j = 0; j < height\_tiles; j++) {

if (i + j \* width\_tiles == 0)

continue;

else {

int tile\_width\_begin = i \* img\_width / width\_tiles;

int tile\_width\_end = (i + 1) \* img\_width / width\_tiles;

int tile\_height\_begin = j \* img\_height / height\_tiles;

int tile\_height\_end = (j + 1) \* img\_height / height\_tiles;

int width\_size = tile\_width\_end - tile\_width\_begin + 2 \* center;

int height\_size = tile\_height\_end - tile\_height\_begin + 2 \* center;

int size = width\_size \* height\_size;

MPI\_Send(&width\_size, 1, MPI\_INT, i + j \* width\_tiles, 1, MPI\_COMM\_WORLD);

MPI\_Send(&height\_size, 1, MPI\_INT, i + j \* width\_tiles, 2, MPI\_COMM\_WORLD);

buf = new unsigned char[size];

for(int w = 0; w < width\_size; w++)

for (int h = 0; h < height\_size; h++) {

int w\_idx = tile\_width\_begin + w - center;

w\_idx = w\_idx > 0 ? (w\_idx < img\_width ? w\_idx : (img\_width - 1)) : 0;

int h\_idx = tile\_height\_begin + h - center;

h\_idx = h\_idx > 0 ? (h\_idx < img\_height ? h\_idx : (img\_height - 1)) : 0;

buf[w + h \* width\_size] = image->data[w\_idx + h\_idx \* img\_width];

}

MPI\_Send(buf, size, MPI\_CHAR, i + j \* width\_tiles, 0, MPI\_COMM\_WORLD);

MPI\_Send(data, kernel\_size \* kernel\_size, MPI\_DOUBLE, i + j \* width\_tiles, 3, MPI\_COMM\_WORLD);

delete buf;

}

}

}

}

if (proc\_rank > 0){

MPI\_Status stat;

int width\_size;

int height\_size;

// принимаем всё что отослали и применяем ядро к блоку

MPI\_Recv(&width\_size, 1, MPI\_INT, 0, 1, MPI\_COMM\_WORLD, &stat);

MPI\_Recv(&height\_size, 1, MPI\_INT, 0, 2, MPI\_COMM\_WORLD, &stat);

MPI\_Probe(0, 0, MPI\_COMM\_WORLD, &stat);

buf = new unsigned char[width\_size \* height\_size];

MPI\_Recv(buf, width\_size \* height\_size, MPI\_CHAR, 0, 0, MPI\_COMM\_WORLD, &stat);

double \*data = new double[kernel\_size \* kernel\_size];

unsigned char \*res\_buf = new unsigned char[width\_size \* height\_size];

MPI\_Recv(data, kernel\_size \* kernel\_size, MPI\_DOUBLE, 0, 3, MPI\_COMM\_WORLD, &stat);

for (int i = 0; i < width\_size; i++) {

for (int j = 0; j < height\_size; j++) {

res\_buf[i + j \* width\_size] = 0;

for (int k\_i = -center; k\_i <= center; k\_i++) {

for (int k\_j = -center; k\_j <= center; k\_j++) {

int new\_i = i - k\_i;

new\_i = new\_i > 0 ? new\_i : 0;

new\_i = new\_i < width\_size ? new\_i : (width\_size - 1);

int new\_j = j - k\_j;

new\_j = new\_j > 0 ? new\_j : 0;

new\_j = new\_j < height\_size ? new\_j : (height\_size - 1);

double add = buf[new\_i + new\_j \* width\_size] \* data[(k\_i + center) + (k\_j + center) \* kernel\_size];

res\_buf[i + j \* width\_size] = clamp(res\_buf[i + j \* width\_size] + add);

}

}

}

}

// отсылаем обработанный блок обратно

MPI\_Send(res\_buf, width\_size \* height\_size, MPI\_CHAR, 0, 0, MPI\_COMM\_WORLD);

}

else {

//обрабатываем блок нулевого процесса

int tile\_width\_begin = 0;

int tile\_width\_end = img\_width / width\_tiles;

int tile\_height\_begin = 0;

int tile\_height\_end = img\_height / height\_tiles;

for (int i = 0; i < tile\_width\_end; i++) {

for (int j = 0; j < tile\_height\_end; j++) {

res\_image->data[i + j \* img\_width] = 0;

for (int k\_i = -center; k\_i <= center; k\_i++) {

for (int k\_j = -center; k\_j <= center; k\_j++) {

int new\_i = i - k\_i;

new\_i = new\_i > 0 ? new\_i : 0;

new\_i = new\_i < img\_width ? new\_i : (img\_width - 1);

int new\_j = j - k\_j;

new\_j = new\_j > 0 ? new\_j : 0;

new\_j = new\_j < img\_height ? new\_j : (img\_height - 1);

double add = image->data[new\_i + new\_j \* img\_width] \* data[(k\_i + center) + (k\_j + center) \* kernel\_size];

res\_image->data[i + j \* img\_width] = clamp(res\_image->data[i + j \* img\_width] + add);

}

}

}

}

}

if (proc\_rank == 0) {

for (int i = 1; i < proc\_num; i++) {

//принимаем обработанные блоки, пересчитываем их размер(мы его уже забыли к этому времени) и засовываем блоки в результирующую картинку

MPI\_Status stat;

MPI\_Probe(MPI\_ANY\_SOURCE, 0, MPI\_COMM\_WORLD, &stat);

int width\_idx = stat.MPI\_SOURCE % width\_tiles;

int height\_idx = stat.MPI\_SOURCE / width\_tiles;

int tile\_width\_begin = width\_idx \* img\_width / width\_tiles;

int tile\_width\_end = (width\_idx + 1) \* img\_width / width\_tiles;

int tile\_height\_begin = height\_idx \* img\_height / height\_tiles;

int tile\_height\_end = (height\_idx + 1) \* img\_height / height\_tiles;

int width\_size = tile\_width\_end - tile\_width\_begin + 2 \* center;

int height\_size = tile\_height\_end - tile\_height\_begin + 2 \* center;

unsigned char \*res\_buf = new unsigned char[width\_size \* height\_size];

MPI\_Recv(res\_buf, width\_size \* height\_size, MPI\_CHAR, stat.MPI\_SOURCE, 0, MPI\_COMM\_WORLD, &stat);

for (int w = tile\_width\_begin; w < tile\_width\_end; w++)

for (int h = tile\_height\_begin; h < tile\_height\_end; h++)

res\_image->data[w + h \* img\_width] = res\_buf[(w - tile\_width\_begin + center) + (h - tile\_height\_begin + center) \* width\_size];

delete res\_buf;

}

end\_time = MPI\_Wtime();

std::cout << "Parallel time: " << end\_time - start\_time << std::endl;

// проверяем сходится ли результат линейной и параллельной версии

bool ok = true;

for (int i = 0; i < img\_size; i++) {

if (res\_image->data[i] != lin\_image->data[i]) {

ok = false;

}

}

if (ok) {

std::cout << "TEST PASSED!" << std::endl;

}

else {

std::cout << "TEST FAILED!" << std::endl;

}

//выводим картинки

namedWindow("RAW IMAGE", WINDOW\_KEEPRATIO);

imshow("RAW IMAGE", \*image);

namedWindow("LINE CALC", WINDOW\_KEEPRATIO);

imshow("LINE CALC", \*lin\_image);

namedWindow("PARALLEL CALC", WINDOW\_KEEPRATIO);

imshow("PARALLEL CALC", \*res\_image);

waitKey(0);

}

MPI\_Finalize();

return 0;

}