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 ядром Гаусса при помощи технологий OpenMP и TBB»**

**Выполнил:**

студент группы 0836-2

Волков П.С.

**Проверил:**

Козинов Е.А.

Нижний Новгород

2019

**Содержание**

[Постановка задачи 3](#_Toc8717770)

[Метод решения 4](#_Toc8717771)

[OS и окружение 4](#_Toc8717772)

[Компоновка файлов 4](#_Toc8717773)

[Фильтрация Гаусса 4](#_Toc8717774)

[Схема распараллеливания 6](#_Toc8717775)

[Описание программной реализации 7](#_Toc8717776)

[Подтверждение корректности 13](#_Toc8717777)

[Результаты экспериментов 14](#_Toc8717778)

[Заключение 15](#_Toc8717779)

[Приложение 16](#_Toc8717780)

# Постановка задачи

Основная задача состоит в реализации методов линейной фильтрации изображения, используя фильтрацию Гаусса. Данные методы должны быть представлены в трех реализациях:

1. Последовательная версия
2. Параллельная версия с использование библиотеки OpenMP
3. Параллельная версия с использование библиотеки Intel Threading Building Blocks

Данные блоки должны основываться на одном и том же внутреннем процессе обработки, т.к. после их реализации мы попробуем сравнить производительность полученных решений. Разбиение изображения на части в случае непоследовательной обработки – горизонтальное.

# Метод решения

## OS и окружение

1. OS - Kubuntu 18.04
2. g++ (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0
3. OpenCV 3.3.1
4. OpenMP – libtbb-dev версии (2017~U7-8)
5. Аппаратное обеспечение
   1. Процессор - Intel Core i7 4700MQ 2400 МГц (4 ядра, в два потока)
   2. Ram - 8 ГБ DDR3 1333 МГц
   3. SSD Kingston A400 [SA400S37/120G]

## Компоновка файлов

Для удобства пользователя в каждой из трех версий программы решено было использовать один и тот же класс, предоставляющий набор одинаковых методов для фильтрации изображения вне зависимости от их внутренней реализации.

Конечный продукт может использоваться с помощью двух файлов с именами GaussImageProcessor.h и GaussImageProcessor.cpp, и соответствующих библиотек для компиляции. Также было реализовано тестовое приложение, находящееся в файле main.cpp, которым можно быстро обработать изображение и получить результат.

## Фильтрация Гаусса

Фильтрация изображения с использованием «ядра Гаусса» подразумевает некоторое усреднение значений интенсивности пикселей за счет его соседей и коэффициентов самого ядра гаусса.

Что же такое ядро гаусса? Ядро Гаусса представляет из себя симметричную квадратную матрицу размером N \* N элементов, где N – целое нечетное число. Элементы матрицы – это числа, вычисляемые по следующим формулам.

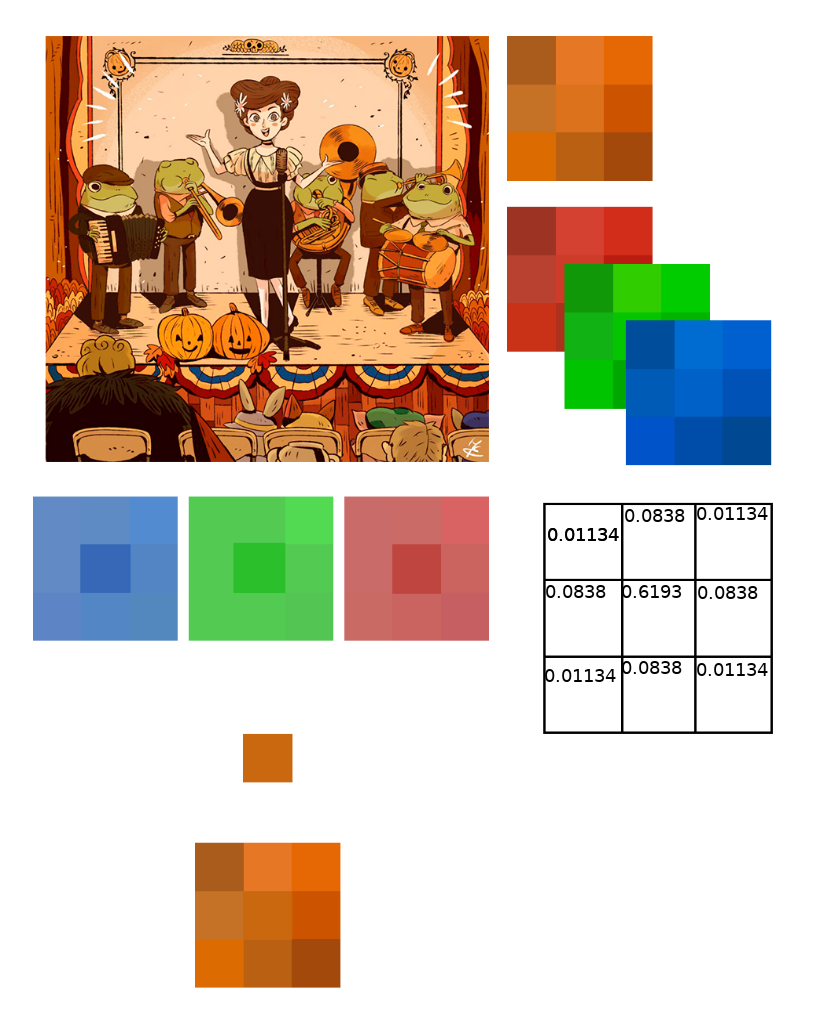
1. Для начала вычислим матрицу Гаусса G’ в ненормированном виде. Пусть размер матрицы NxN. Тогда коэффициенты будут вычислены следующим образом:

Число – это степень размытия изображения. Чем меньше , тем большее влияние окажут соседние пиксели на интенсивность искомого.

1. После чего, мы сможем вычислить конечную матрицу Гаусса просто нормировав все коэффициенты их общей суммой по всем :

Теперь рассмотрим применение ядра Гаусса к фильтрации изображения. Рассмотрим фильтрацию конкретного пикселя на небольшом примере.

Пусть у нас есть некоторое изображение, где мы будем рассматривать пиксель [x, y] для его фильтрации матрицей размером 3x3 и



Итак, что же будет происходить при фильтрации? Рассмотрим алгоритм по шагам:

1. Из изображения извлекается подматрица, где элемент [x, y] находится в центре.
2. Подматрица раскладывается на цветовые составляющие.
3. Каждая интенсивность пикселя цветовой составляющей умножается на коэффициент матрицы Гаусса.
4. Затем происходит суммирование всех интенсивностей в каждой цветовой составляющей. Получается три новых значения R, G и B.
5. И наконец пикселю [x, y] присваивается новый цвет, полученный путем усреднения соседних с помощью матрицы гаусса.

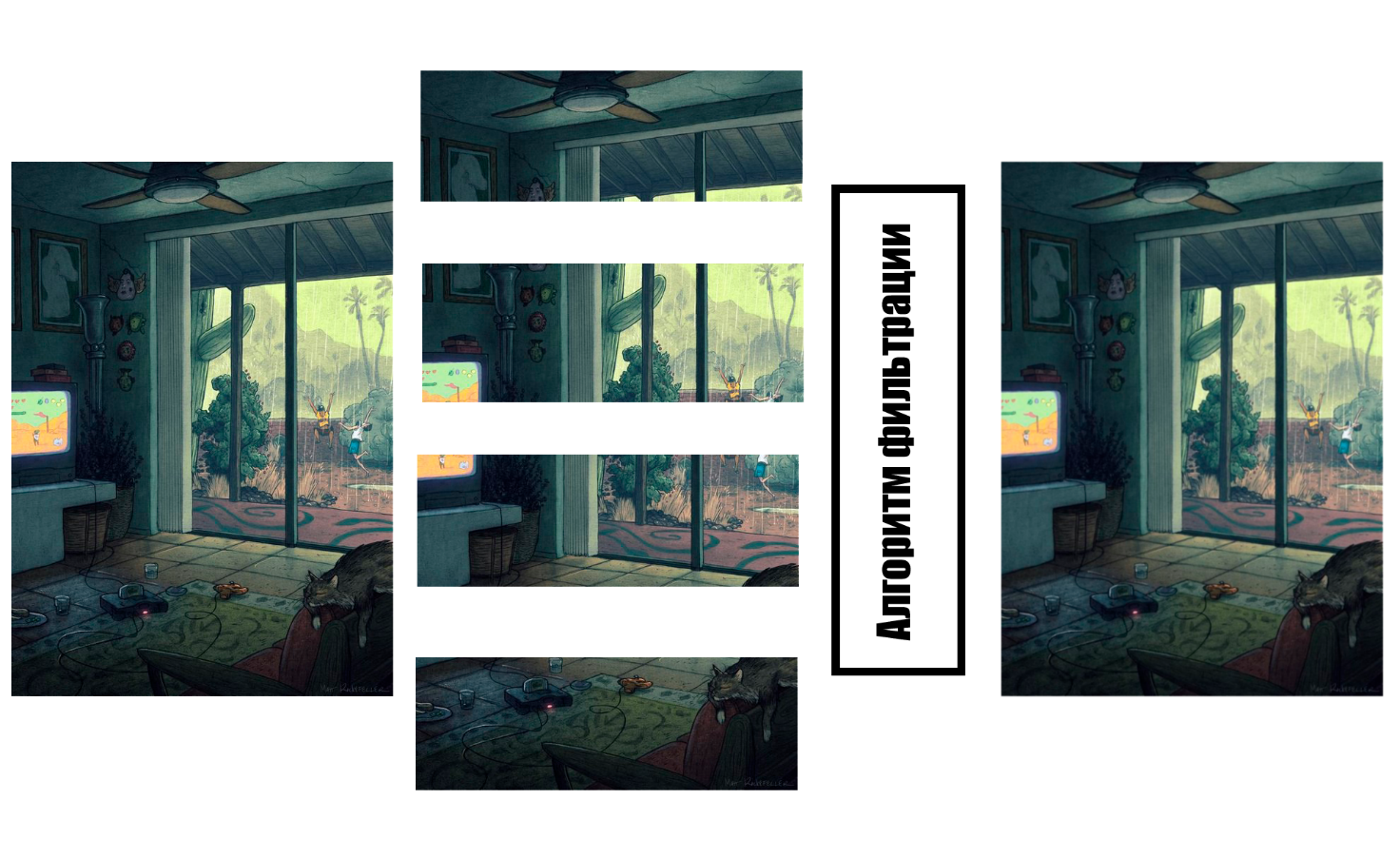
Слева представлена схема происходящего.

Разумеется, подобные вычисления связаны с огромной вычислительной нагрузкой. Особенно, когда матрица имеет большой размер. Количество вычислений возрастает в квадратичном порядке. Но вычисления могу производиться параллельно, т.к. значение пикселя в новой картинке зависят лишь от интенсивности пикселей оригинала и никак от своих соседей.

# Схема распараллеливания

Так как значения интенсивностей новых пикселей изображения зависят от значений этих пикселей и их соседей лишь на старой картинке, то каждый пиксель новой картинки можно обрабатывать в любом порядке и независимо. Средствами параллельного программирования мы можем добиться существенного прироста производительности при решении данной задачи. Осталось лишь распределить работу.

Схема распределения работ для параллельных версий одна. Мы делим исходную картинку на несколько горизонтальных полос, которые затем обрабатываются одновременно. Визуально мы будем иметь следующую картину:

Приведенная схема распределения работы верна для приведенной картинки и при условии, что пользователь имеет в распоряжении (инициализировал) для работы четыре потока. А также работы будут поделены иначе при использовании картинки, в высоту имеющей не более пикселей, чем потоков в распоряжении. В таком случае, всю работу придется выполнять меньшему количеству потоков. Но задание четко оговаривает вариант разбиения работ.

Для получения корректных результатов сравнения времени работы программы необходимо реализовать модуль фильтрации максимально одинаковой с точки зрения кода. А также, будет предоставлен идентичный интерфейса пользователя во всех трех версиях программы.

# Описание программной реализации

**Руководство пользователя**

Рассмотрим простые примеры использования трех созданных классов. Начнем с общего для всех классов. Реализация представленных классов требует наличия библиотеки OpenCV. Версия, используемая при разработке – **VERSION.**

Для использования методов фильтрации необходимо породить один объект, в котором будет храниться картинка, которую необходимо отфильтровать. Это объект библиотеки OpenCV – Mat.

Mat scrImg;

После чего порождается класс ***GaussImageProcessor.*** В качестве параметра конструктора класс принимает исходное изображение

GaussImageProcessor g(srcImg);

Каждый класс ***GaussImageProcessor*** имеет публичный метод - set\_core(int size, double sigma). Метод принимает два аргумента. Размер матрицы гаусса и параметр сигма. Данный метод позволяет переопределить ядро Гаусса, используемое конкретным экземпляром класса. Пример использования:

g.set\_core(3, 0.5);

Также каждый класс ***GaussImageProcessor*** класс имеет еще один публичный метод – выполнение фильтрации. Метод возвращает все тот же объект класс Mat. Пример использования:

Mat filtred\_img = g.execute\_gauss();

Теперь рассмотрим отличия реализаций со стороны пользователя.

1. Последовательная версия.

Последовательная версия полностью описана и может быть использована при данном выше наборе команд.

1. Версия OpenMP.

Данная версия программы также может использоваться с тем же набором команд. Это связано с тем, что библиотека OpenMP не требует какой-либо явной инициализации. Программы будет выполняться на оптимально-доступном количестве потоков. Но при желании пользователь может явно задать количество потоков в программе стандартным методом OpenMP - omp\_set\_num\_threads(<int>);

1. Версия с использованием библиотеки Intel threading building blocks.

Данная версия требует инициализации самой библиотеки. Это делает порождением единственного класса, который будет заниматься распределением очередей заданий.

tbb::task\_scheduler\_init in(<int>);

После объявления данного класса, который принимает на вход конструктора целое число – число потоков, которое будет использоваться в дальнейшем при исполнении параллельной части программы.

Благодаря подобной реализации всех представленных версий программы, пользователь может не сильно вдаваться в техническую реализацию продукта, а использовать его по своим нуждам быстро и удобно. Ниже представлены набор команд, который был использован для компиляции простейших приложений с использованием технологий TBB и OpenMP. Последовательная версия алгоритма использует для компиляции лишь библиотеки OpenCV для работы с картинками.

КОМПИЛЯЦИОНЫЕ КОММАНДЫ

Для последовательной версии подойдет почти любой компилятор C++ и Библиотека OpenCV. Для компиляции вместе с фалом, где расположена точка входа main использована следующая команда:

g++ \*.cpp -o test `pkg-config --cflags --libs opencv`

Версия с использованием OpenMP:

g++ \*.cpp -o test -fopenmp `pkg-config --cflags --libs opencv`

TBB версия программы. Оставим чуть более подробный список компилируемых файлов для примера:

G++ main.cpp GaussImageProcessor.cpp -o test -ltbb `pkg-config –cflags –libs opencv`

Для всех трех лабораторных справедлив следующий набор файлов в папке проекта, в которой и проходила сборка исполняемого файла:

GaussImageProcessor.cpp

GaussImageProcessor.h

main.cpp

readme.txt

standartArgumentParser.h

Для запуска тестового приложения используется следующий набор аргументов:

use argument: (string)<image path> (U\_int)<gauss\_core\_size> (U\_int)<sigma\_in\_core> (int)[thread\_num]

**Руководство программиста**

В этом разделе рассмотрим лишь основные методы и нюансы их реализации для каждой из версий программы. Начнем с самой простой части – обработки последовательным способом. Приведем наиболее важную часть кода – место обработки картинки:

void gauss\_processing(Mat& originalImage, Mat& outImage, vector<vector<double>> \_core, int begin, int end)

{

vector<vector<double>> core;

core = \_core;

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

for (int j = 0; j < originalImage.cols; j++) // For all pixel

{

… //Обработка пикселей

}

…

}

Данный метод присутствует во всех трех реализациях. Во всех трех реализациях он предоставляет один и тот же интерфейс для его использования. Параметры в порядке их объявления – Оригинал картинки, ссылка на выходную картинку, которую хранит в себе экземпляр класса. Матрица, представленная двойным векторов double – ядро гаусса, begin и end – начальная строка, которую следует обработать включительно и конечная строка, не включительно, до которой следует вести обработку. Все полученные результаты записываются в картинку, которая предназначена на выход. Для последовательной версии программы вызывается метод с начальными параметрами begin и end, равными нулю и высоте картинке соответственно.

Теперь рассмотрим программу, реализованную с помощью библиотеки OpenMP.

Основным отличием от последовательной версии будет фрагмент кода, который будет выполняться параллельно. А значит, перед тем как приступать к его выполнению – нужно поделить задания для каждого из потоков. Это мы сделаем исходя из количества и номеров потоков в параллельной части программы. Далее приведен код, в котором происходит этот расчет и выполняется преобразования гаусса над каждой частью изображения.

void gauss\_processing(Mat& originalImage, Mat& outImage, vector<vector<double>> \_core, int begin, int end)

{

vector<vector<double>> core;

core = \_core;

#pragma omp parallel shared(originalImage, outImage)

{

int thread\_nums = omp\_get\_num\_threads();

int start\_pos;

int end\_pos;

int my\_id = omp\_get\_thread\_num();

int standart\_block;

standart\_block = originalImage.rows / thread\_nums;

// Find the barrier of img parts

if (my\_id < thread\_nums - 1)

{

start\_pos = standart\_block \* my\_id;

end\_pos = start\_pos + standart\_block;

}

else

{

start\_pos = standart\_block \* (thread\_nums - 1);

end\_pos = originalImage.rows;

}

for (int i = start\_pos; i < end\_pos; i++)

for (int j = 0; j < originalImage.cols; j++) // For all pixel

{

… //Обработка части [start\_pos, end\_pos)

}

} // Конец параллельной секции

…

}

Как видно, не так сильно отличается реализация версии OpenMP. Теперь каждый поток рассчитывает свои **start\_pos** и **end\_pos** исходя из своего номера и общего количества потоков. После чего обрабатывает лишь строки, которые сам получил. Все это достигается посредством одной директивы.

Более сложной реализацией обладает последняя версия программы. TBB версия использует так называемы «Таски» и их списки. Начальной целью будет генерация этих самых заданий и заполнения единственного списка, после чего, список с заданиями будет отправлен на выполнение.

Рассмотрим метод, который вызывается пользователем:

Mat GaussImageProcessor::execute\_gauss()

{

outImage = originalImage.clone();

partition();

tbb::task::spawn\_root\_and\_wait(tasks);

return outImage;

}

Метод ***partition()*** заполняет список тасков ***tasks***, объявленный как экземпляр класса. После чего происходит запуск их исполнения. Рассмотрим, как порождаются таски и что они из себя представляют.

class ImageProcessorPart : public tbb::task

{

private:

Mat originalImage;

Mat outImage;

int begin, end; // rows

vector<vector<double>> core;

public:

ImageProcessorPart(Mat& \_orIm, Mat& \_outIm, int \_begin, int \_end, vector<vector<double>>& \_core);

tbb::task\* execute();

};

Класс, представленный выше представляет из себя тот самый единичный таск, множество которых мы будем порождать в методе ***partition()***. Класс содержит в себе две ссылки. На оригинал и на изображение, в которое будет записываться результат. А также переменные, которые означают начало и конец работ, которые необходимо выполнить.

…

{

int standart\_pack = rows / thread\_num;

for (int i = 0; i < thread\_num - 1; i++)

{

ImageProcessorPart& ip = \*new (tbb::task::allocate\_root()) ImageProcessorPart(originalImage, outImage, i \* standart\_pack, i \* standart\_pack + standart\_pack, core);

tasks.push\_back(ip);

}

ImageProcessorPart& ip = \*new (tbb::task::allocate\_root()) ImageProcessorPart(originalImage, outImage, standart\_pack \* (thread\_num - 1), rows, core);

tasks.push\_back(ip);

}

…

Как видно, в этом фрагменте кода и происходит генерация тасков. Таск инициализируется двумя ссылками на изображение, началом и концом каждого таска. После чего отправляется в список, о котором говорилось ранее. Осталось лишь описать, каждый таск с моделью его исполнения. Это делается при перегрузки следующего метода:

tbb::task\* ImageProcessorPart::execute()

{

gauss\_processing(originalImage, outImage, core, begin, end);

return NULL;

}

На этом руководство над программную часть можно считать завершенным. Остальные методы, такие как генерация ядра Гаусса или обработка краев картинки довольно тривиальны. И не требуют разбора или пояснений. Они вполне тривиальны.

Ко всему этому можно добавить, что, если программисту захочется использовать какой-либо другой матричный фильтр – он должен лишь изменить его генерацию.

Таким образом, были реализованы не только классы, которые могут производить фильтрацию Гаусса, но и шаблон для использования матричных фильтров в совокупности с преимуществами параллельных версий программ.

# Подтверждение корректности

Для подтверждения корректности версий с параллельными участками кода и последовательными версиями создана одна небольшая функция.

bool check(Mat im1, Mat im2)

{

for (int i = 0; i < im1.rows; i++)

for (int j = 0; j < im1.cols; j++)

{

if (im1.at<Vec3b>(i, j)[2] != im2.at<Vec3b>(i, j)[2])

return false;

if (im1.at<Vec3b>(i, j)[1] != im2.at<Vec3b>(i, j)[1])

return false;

if (im1.at<Vec3b>(i, j)[0] != im2.at<Vec3b>(i, j)[0])

return false;

}

return true;

}

Эта функция просто проверяет попиксельно два изображения. Таким образом можно сравнить несколько реализаций преобразования картинок.

# Результаты экспериментов

Для экспериментов возьмем одну картинку и проведем несколько тестов с помощью различных версий программ.

Размеры картинки: 1600x926 px.

Результаты экспериментов представим в графическом виде. Эксперименты будем производить на размерах матриц 3x3 и 6x6. Теоретически, реализуемые методы должны позволить добиться линейного ускорения при условии, что все потоки работают независимо и не перекрывают время работы друг друга. Но т.к. их явно недостаточно, то ускорение будет линейным лишь при начально приращении количества потоков. При увеличении матрицы Гаусса до размера 6x6 предполагается квадратичный рост времени работы программы. Аппаратная и Программная часть среды исполнения программы описана в начале.

Результаты:

# Заключение

Как можно заметить из результатов экспериментов – наши предположения во многом были верны. Для начала отметим, что использование параллельных версий программ действительно дает большое преимущество в скорости работы. Время результатов экспериментов – усредненное время из десяти попыток на каждое количество потоков.

Мы действительно видим первоначальный линейный прирост производительности. В особенности при переходе от одного потока к двум и трем. Но по мере увеличения их количества время перестает так стремительно убывать. Потоки не успевают работать все вместе на данной конфигурации оборудования. При увеличении размера матрицы в два раза, скорость работы программ выросла, как и предполагалось квадратично начальным результатом, исключая те моменты, когда очень много потоков пересекаются между собой. При их большом количестве время все же имеет некоторую склонность к более медленному убыванию.

Версии OpenMP и TBB практически ничем не уступают друг другу. Их графики держаться почти на одинаковом уровне.

Таким образом были реализованы и протестированы три версии программ, которые осуществляют фильтрацию Гаусса. Освоены технологии использования OpenMP и TBB для достижения большей производительности программ.

# Приложение

Линейная версия приложения:

Версия с использованием OpenMP:



Версия с использованием TBB:



Текстовые версии программ можно просмотреть в следующем репозитории:

<https://github.com/Snewar79/2018_par_prog_381606-2>

Или:

<https://github.com/kozinove/2018_par_prog_381606-2> (Возможно в открытых реквестах)

~/MPI/Volkov Pavel Sergeevich/