МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского» (ННГУ)

Институт информационных технологий, математики и механики

Кафедра программной инженерии

Направление подготовки: «Программная Инженерия»

ОТЧЕТ

по курсу «Параллельное программирование» на тему

«Линейная фильтрация изображений (горизонтальное разбиение). Ядро Гаусса 3*3.»

Выполнил: Студент группы 381608,
Черненко Валерий Николаевич

Научный руководитель: Доцент кафедры МОиСТ, кандидат технических наук,
Сысоев Александр Владимирович

Нижний Новгород
2019

Оглавление:

Введение:

Обработка изображения – это любая форма обработки информации, для которой входные данные предоставлены изображением. Обработка изображения может осуществляться как для получения статичного изображения, так и для изображения, измеряющегося со временем. С обработкой изображения связано множество математический и прикладных задач. Среди них геометрические преобразования, такие как вращение, масштабирование, цветовая коррекция, применение фильтров и др.

Под фильтрацией изображений понимают операцию, имеющую своим результатом изображение того же размера, полученное из исходного по некоторым правилам. Обычно интенсивность (цвет) каждого пикселя результирующего изображения обусловлен интенсивностями (цветами) пикселей, расположенных в некоторой его окрестности в исходном изображении.

Линейная фильтрация характерна тем, что значение выходного пикселя определяется линейной комбинацией соответствующих входных пикселей. Линейная фильтрация изображения выполняется с помощью операции свертки. При выполнении этой операции, значение выходного пикселя вычисляется как взвешенная сумма соседних пикселей. Матрица весов называется ядром свертки (эта матрица также называется фильтром).

Постановка задачи:

Задачей практической работы по курсу «Параллельное программирование» является разработка программы, реализующей последовательный и параллельный алгоритм применения к изображению фильтра Гаусса с матрицей 3*3 и линейный разбиением (по строкам) изображения в параллельной версии.

Цель данной работы:

1. Реализовать последовательную версию алгоритма.
2. Реализовать параллельную версию алгоритма с помощью технологии OpenMP, используя линейное разбиение изображения.
3. Реализовать параллельную версию алгоритма с помощью технологии TBB, использую линейное разбиение изображения.
4. Замерить время работы последовательной и параллельной версии при запуске на многопроцессорном персональном компьютере (или кластере), оценить ускорение, эффективность и масштабируемость каждой из версии. Сделать выводы.

Методы решения:

Последовательная версия:

Последовательная версия алгоритма является основной для разработки программы и делиться на 3 этапа:

1. Получение изображения (с диска или случайная генерация).
2. Обработка изображения.
3. Сохранение изображения.

Работа с изображениями будет осуществляться с помощью библиотеки OpenCV версии 346. Для этого используется объект *cv::Mat,* находящийся в подключаемом файле <opencv2/core/core.hpp>. Список основных методов, которые могут понадобиться:

- *cv::Mat::Mat(int _rows, int _cols, int _type)* – конструктор класса *cv::Mat*, создающий изображение размером *_rows*_cols* и типом *_type*. Для получения трехцветного изображения в формате RGB, параметр *_type* необходимо задать как **CV_8UC3.**
- *int cv::Mat::rows* – поле класса, содержащее информацию о высоте изображения.
- *int cv::Mat::cols* – поле класса, содержащее информацию о ширине изображения.
- *unsigned char* cv::Mat::data* – поле класса, являющееся одномерным массивом, содержащим информацию о значениях пикселей изображения. Так для изображения типа **CV_8UC3,** поле *cv::Mat::data* имеет размер, равный утроенному произведению высоты изображения на его ширину, и хранит информацию в следующем виде: R-компонента 1 пикселя, G-компонента 1 пикселя, B-компонента 1 пикселя, R-компонента 2 пикселя, G-компонента 2 пикселя, B-компонента 2 пикселя и т.д. Так для получения R-компоненты пикселя по координатам (x, y), где x – номер строки, начиная сверху, а y – номер столбца, начиная

слева, вычисляется по следующей формуле: *image.data[3*(x*image.cols + y)]*.

- *cv::Mat cv::imread(const cv::String &filename, int flag = 1)* – Функция, содержащаяся в подключаемом файле <opencv2/highgui/highgui.hpp>, которая записывает изображение, находящийся по пути **filename,** в возвращаемый объект типа *cv::Mat.* В случае неудачи возвращает пустой объект типа *cv::Mat*. Вторым параметром можно пренебречь.
- *Bool cv::imwrite(const cv::String &filename, cv::InputArray img, const std::vector<int> &params = std::vector<int> ())* - Функция, содержащаяся в подключаемом файле <opencv2/highgui/highgui.hpp>, которая записывает изображение, передаваемый параметром **img,** в файловую систему по пути **filename**. Третьим параметром можно пренебречь.

Таким образом для получения изображения с диска необходимо воспользоваться функцией *cv::imread*. В случае, если изображение отсутствует или его необходимо задать случайным образом с конкретными размерами, реализована функция *cv::Mat getRandImage(int width, int height)*. Реализация функции находится в *приложении 1*.

После считывания информации о изображении задать фильтр Гаусса. Размытие по Гауссу - это характерный фильтр размытия изображения, который использует нормальное распределение (также называемое Гауссовым распределение) для вычисления преобразования, применяемого к каждому пикселю изображения. Фильтр Гаусса представляет из себя квадратную матрицу n*n, где каждое значение вычисляется по следующей формуле:

$$G(x + r, y + r) = \frac{1}{2\pi r^2} e^{\frac{-(x^2+y^2)}{2r^2}}, \forall x, y \in [-r, r]$$

Где r – радиус распределения равный целой части от деления n / 2.

Так как сумма всех элементов в фильтре Гаусса должна быть равна 1, необходимо выполнить нормировку – разделить каждый элемент на сумму всех элементов в матрице. Данная задача реализована в функции *double** *getGaussesFilter(int n)* (см. Прил. 1).

Вычисление нового значения пикселя в конкретной точке изображения вычисляется по формуле:

$M_{\text{н}}(x, y) = \sum_{u=-r}^{r} \sum_{v=-r}^{r} (G(u + r, v + r) * M(x + u, y + v))$,

т.е. для каждого пикселя изображения (обход изображения происходит двойным циклом: сначала цикл по строчкам, затем в каждой итерации цикл по элементам строки) получаем окрестность размером фильтра с центром в заданной точке. К примеру, если у нас есть точка с координатами (x, y), то получаем квадрат (x-r, y-r), (x+r, y+r), где r – радиус распределения, равный целой части деления n/2. Затем происходит «наложение» фильтра на эту область. Результирующее значение получаем как результат суммирования произведений соответствующего пикселя на значение в матрице. Для наглядного примера предоставлен иллюстрация применения матрицы на такую область (рис. 1) (div – коэффициент нормирования, равный сумме элементов матрицы)

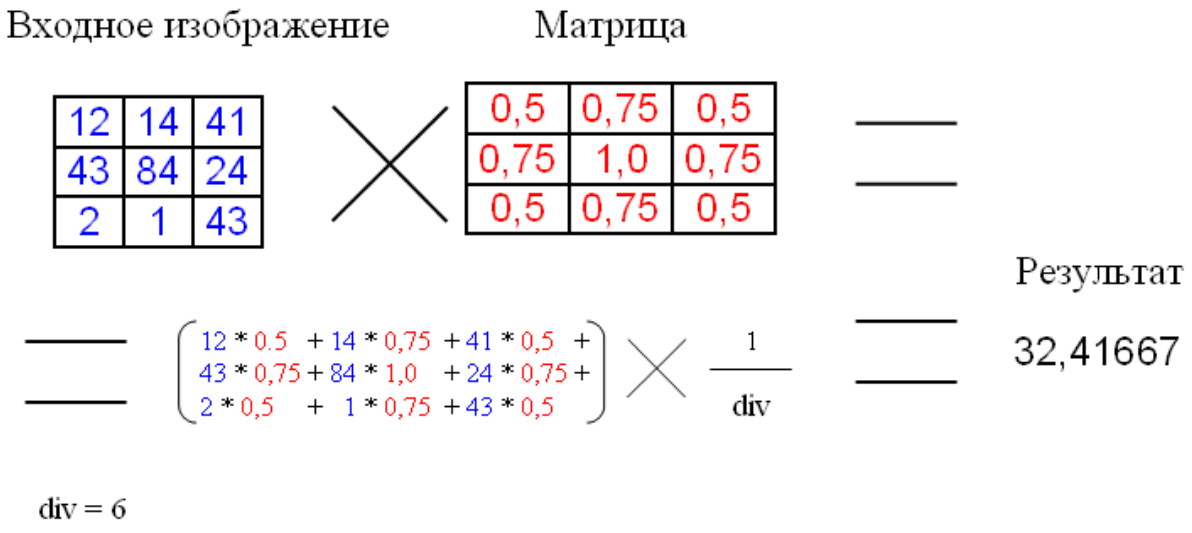

*Рисунок 1. Вычисление нового значения для матрицы 3*3*

Реализация последовательного алгоритма расположена в функции *cv::Mat filterImage(cv::Mat image, double** filter, int n)*, находящейся в прил. 1.

Общий код программы, выполняющий последовательный алгоритм считывания изображения, его обработки и записи находится в прил. 1.

Для проверки корректности применяемого фильтра программу запустили по отношению к изображению черно-белых полос размером 256*256 (рис. 2). Затем к данному изображению был применен фильтр Гаусса размером 3*3 (рис. 3), 7*7 (рис. 4), 15*15 (рис. 5).

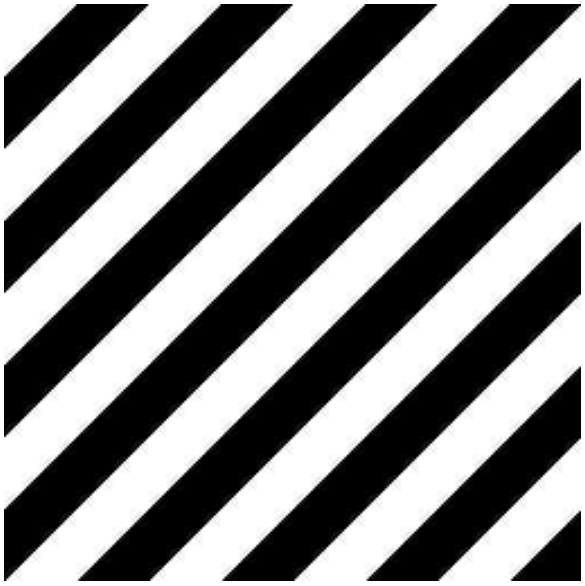

Рисунок 2. Исходное изображение

*Рисунок 3. Примененный фильтр 3*3*

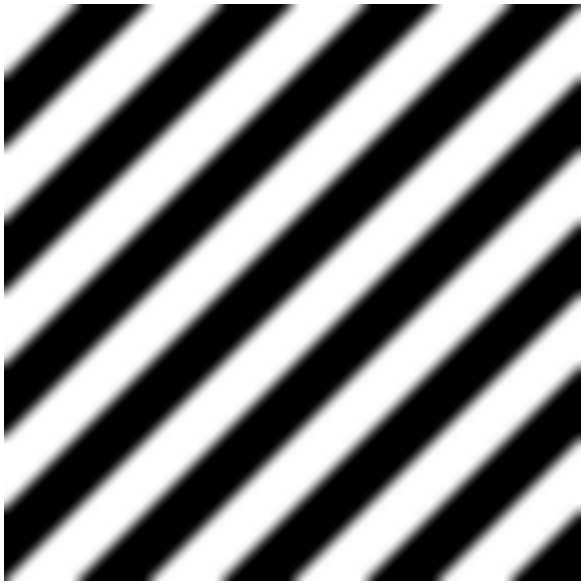

*Рисунок 4. Примененный фильтр 7*7*

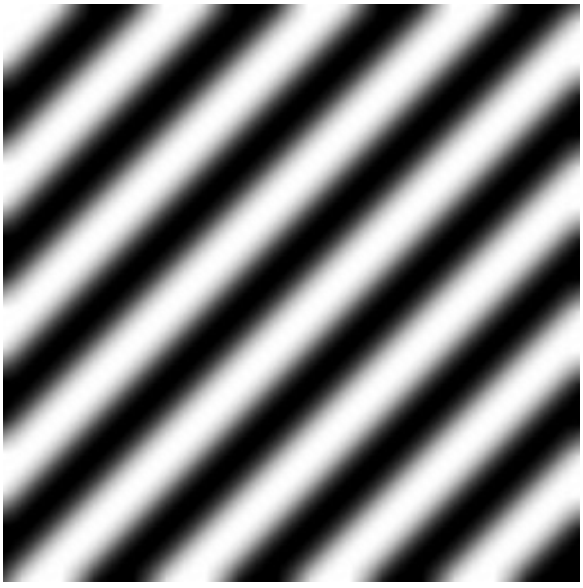

*Рисунок 5. Примененный фильтр 15*15*

Исходя из полученный изображений можно сделать вывод, что примененный фильтр работает корректно и изображение размывается.

Параллельная версия с использованием технологии OpenMP:

Параллельная версия программы делиться на те же три этапа, что и разработка последовательной версии, причем получение изображения и запись его на диск остается аналогичной.

Для распараллеливания алгоритма следует изменить код функции *cv::Mat filterImage(cv::Mat image, double** filter, int n).* Так как по заданию требуется горизонтальное разбиение, т.е. разбиение по строчкам, то необходимо распараллелить цикл прохода по строчкам изображения, который является циклом с известным числом повторений. Для этого используем директиву for с параметром *schedule(static).* Реализация находится в функции *cv::Mat filterImageOmpPar(cv::Mat image, double** filter, int n) (см. Прил. 2).* Общий код программы, реализующий параллельную версию алгоритма с использованием технологии OpenMP, находится в приложении 2.

Для проверки корректности применяемого фильтра программу запустили по отношению к изображению черно-белых полос размером 256*256 (рис. 6). Затем к данному изображению был применен фильтр Гаусса размером 3*3 (рис. 7), 7*7 (рис. 8), 15*15 (рис. 9).

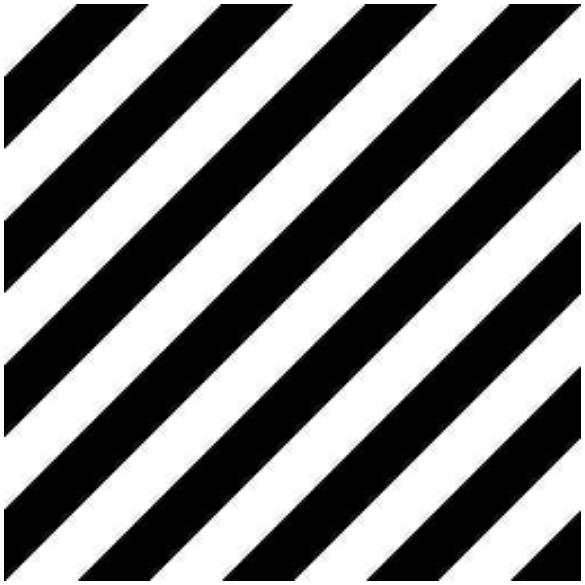

Рисунок 6. Исходное изображение

*Рисунок 7. Примененный фильтр 3*3*

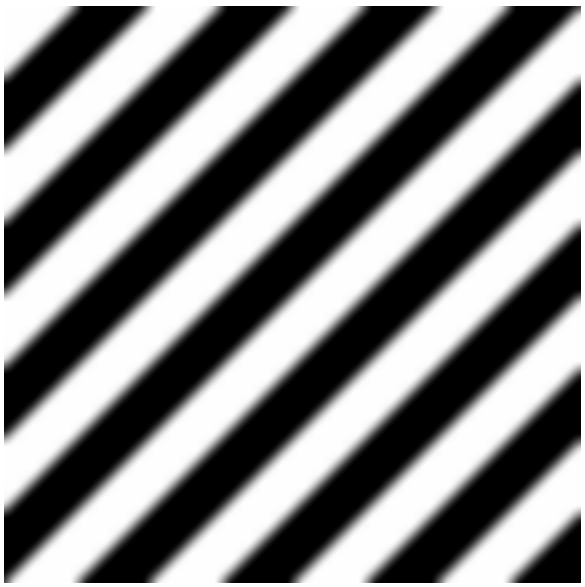

*Рисунок 8. Примененный фильтр 7*7*

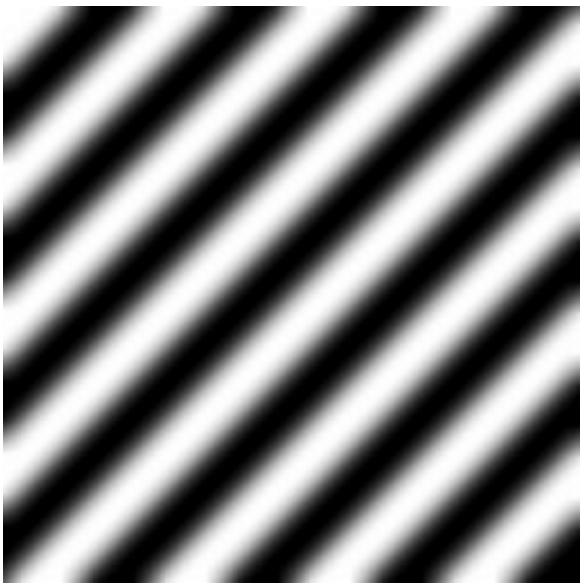

*Рисунок 9. Примененный фильтр 15*15*

Как видно, полученные изображения никак не отличаются от изображений, полученных с помощью последовательной версии, что означает корректную реализацию.

Параллельная версия с использованием технологии TBB:

Параллельная версия программы делиться на те же три этапа, что и разработка последовательной версии, причем получение изображения и запись его на диск остается аналогичной.

Для распараллеливания алгоритма следует изменить код функции *cv::Mat filterImage(cv::Mat image, double** filter, int n).* Так как по заданию требуется горизонтальное разбиение, т.е. разбиение по строчкам, то необходимо распараллелить цикл прохода по строчкам изображения, который является циклом с известным числом повторений. Для этого используем функцию *tbb::parallel_for,* расположенную в подключаемом файле <tbb/tbb.h>. Первым параметром функции передаем стандартное одномерное итерационное пространство tbb::*blocked_range<int>(0, image.rows, image.rows/ tbb::task_scheduler_init::default_num_threads()),* где в качестве параметра *grandsize* используется число строк поделенное на число потоков, создаваемое по умолчанию. Это гарантирует эффективную работу каждого потока и уменьшает накладные расходы на разбиение итерационного пространства. Вторым параметром передается функтор. Для этого написан функтор *FilteringImage*, хранящий в себе информацию о изображении, фильтре и перегружающий метод *operator(),* принимающий итерационное пространство и выполняющий непосредственную обработку изображения. (см. Прил. 3). Полная обработка изображения с созданием объекта функтора *FilteringImage* и вызовом функции *tbb::parallel_for* реализована в функции *cv::Mat filterImageTbb(cv::Mat image, double** filter, int n).* Общий код программы, реализующий параллельную версию алгоритма с использованием технологии TBB, находится в приложении 3.

Для проверки корректности применяемого фильтра программу запустили по отношению к изображению черно-белых полос размером 256*256 (рис. 10). Затем к данному изображению был применен фильтр Гаусса размером 3*3 (рис. 11), 7*7 (рис. 12), 15*15 (рис. 13).

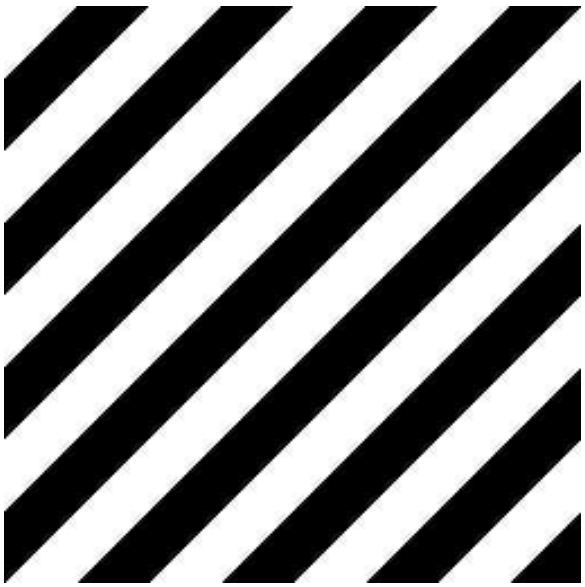

Рисунок 10. Исходное изображение

*Рисунок 11. Примененный фильтр 3*3*

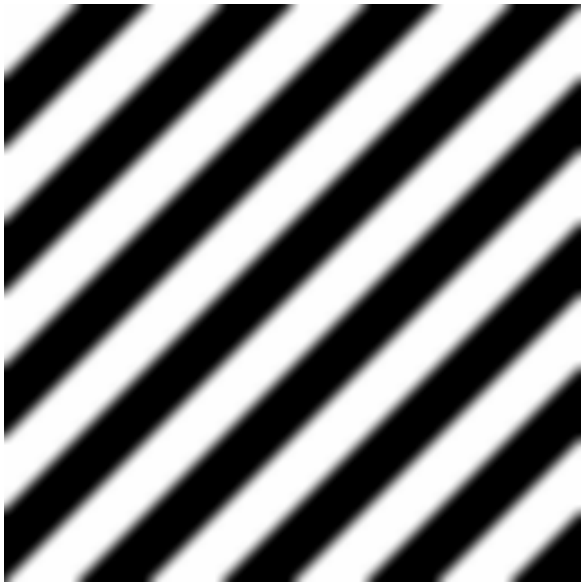

*Рисунок 12. Примененный фильтр 7*7*

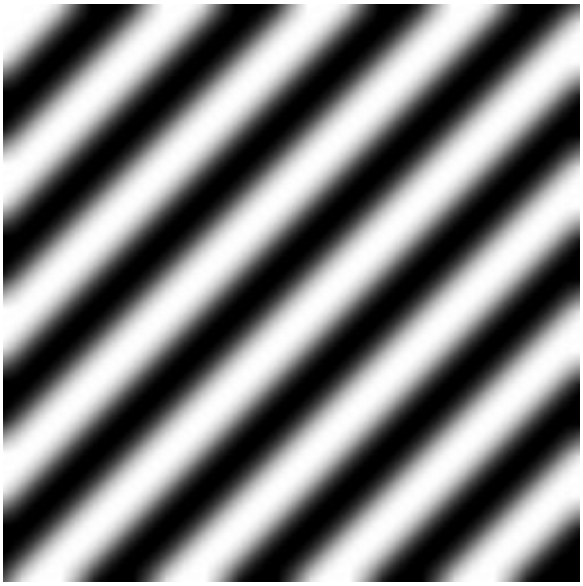

*Рисунок 13. Примененный фильтр 15*15*

Как видно, полученные изображения никак не отличаются от изображений, полученных с помощью последовательной версии, что означает корректную реализацию.

Результаты экспериментов:

Для определения эффективности параллельных версий по сравнению с последовательной были проведены несколько пробных запусков программы на персональном компьютере со следующими характеристиками ЦП:

Наименование: Intel® Core™ i5-7300HQ

Количество ядер: 4

Количество потоков: 4

Полученные результаты были занесены в таблицу (табл. 1) (Все значения указаны в секундах).

Количество потоков:		1 поток			2 потока		4 потока		8 потоков	
Размеры из-ния	Размеры фильтра	Linear	OpenMP	TBB	OpenMP	TBB	OpenMP	Tbb	OpenMP	TBB
128	3	0,0006	0,00069	0,00081	0,000464	0,00058	0,00036	0,00049	0,0003	0,00067
	7	0,0031	0,00302	0,00324	0,0016	0,0017	0,0011	0,0014	0,009	0,0014
	13	0,0095	0,0097	0,0095	0,0052	0,0051	0,00295	0,00298	0,003	0,0032
	17	0,0157	0,017	0,0176	0,008	0,0079	0,005	0,0051	0,0053	0,0052
	21	0,0237	0,0235	0,0234	0,014	0,015	0,0088	0,0081	0,0079	0,0084
256	3	0,0025	0,00278	0,00272	0,001582	0,001589	0,0011	0,0014	0,0008	0,0013
	7	0,0133	0,0129	0,013	0,00648	0,00639	0,00369	0,00371	0,0037	0,0039
	13	0,0377	0,0377	0,0377	0,019	0,0187	0,0107	0,0106	0,0113	0,116
	17	0,0648	0,065	0,0649	0,0318	0,031	0,0186	0,0176	0,0189	0,0185
	21	0,097	0,096	0,094	0,0468	0,046	0,028	0,026	0,0265	0,0261
512	3	0,0106	0,0113	0,0105	0,0053	0,0053	0,003	0,0037	0,0031	0,0035
	7	0,051	0,0519	0,0506	0,025	0,0247	0,0158	0,0144	0,0145	0,0148
	13	0,154	0,154	0,149	0,0755	0,075	0,041	0,039	0,042	0,044
	17	0,25	0,253	0,25	0,124	0,121	0,07	0,067	0,069	0,064
	21	0,383	0,384	0,379	0,191	0,182	0,104	0,103	0,106	0,113
1024	3	0,0417	0,043	0,0415	0,022	0,021	0,0154	0,014	0,011	0,011
	7	0,1999	0,1997	0,199	0,098	0,095	0,058	0,0529	0,525	0,509
	13	0,606	0,609	0,594	0,292	0,288	0,16	0,162	0,163	0,163
	17	1,044	1,033	1,034	0,505	0,488	0,257	0,255	0,276	0,316
	21	1,506	1,509	1,488	0,748	0,734	0,39	0,383	0,395	0,399
2048	3	0,1712	0,176	0,171	0,084	0,08	0,0505	0,044	0,47	0,48
	7	0,774	0,777	0,755	0,39	0,37	0,204	0,202	0,204	0,207
	13	2,4	2,38	2,42	1,203	1,188	0,62	0,619	0,626	0,625
	17	4,107	4,12	4,106	2,067	2,012	1,065	1,048	1,072	1,068
	21	6,127	6,128	6,034	3,039	3,021	1,57	1,56	1,58	1,61
4096	3	0,669	0,681	0,679	0,342	0,325	0,176	0,169	0,177	0,18
	7	3,164	3,158	3,085	1,579	1,53	0,8	0,79	0,81	0,82
	13	9,92	9,9	9,87	4,958	4,876	2,58	2,51	2,54	2,51
	17	16,36	16,4	16,2	8,12	7,97	4,25	4,17	4,218	4,223
	21	24,467	24,4966	24,1112	12,11	11,92	6,27	6,21	6,26	6,2
8192	3	2,7	2,74	2,65	1,355	1,293	0,697	0,665	0,697	0,672
	7	12,945	12,8	12,36	6,358	6,227	3,245	3,162	3,238	3,187
	13	39,527	39,6698	39,1783	19,72	19,4062	10,207	10,01	10,19	9,997
	17	65,672	65,1865	64,6963	32,658	32,448	16,9377	16,5556	16,9097	16,6006
	21	97,832	98,1974	97,6805	49,0582	48,3536	25,2152	24,776	25,1522	25,2294

Таблица 1. Результаты запусков программы (сек)

Для большей точности каждое значение было высчитано как среднее арифметическое среди 10 запусков программы, чтобы избежать возможных ситуаций, когда время увеличивается из-за работы посторонних программ.

По данной таблице для более детальной оценки построили несколько графиков. Первый график отображает зависимость времени выполнения работы программы на разных алгоритмах для изображения 2048*2048 и фильтре 3*3 от количества потоков (граф. 1), так же уменьшенную копию таблицы с вычисленным ускорением (табл. 2)

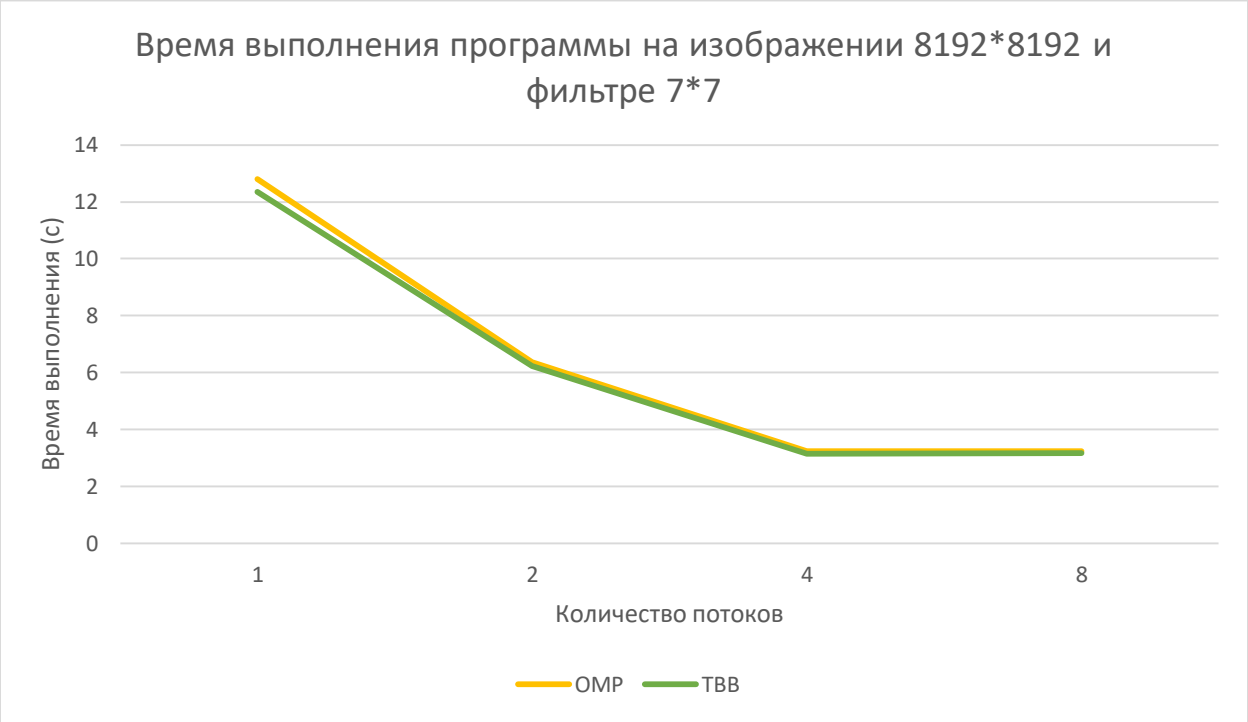

График 1. Зависимость времени выполнения программы от количества потоков.

	1	2	4	8
Linear	12,945	12,945	12,945	12,945
OMP	12,8	6,358	3,245	3,238
TBB	12,36	6,227	3,162	3,187
Boost OMP	1,011328	2,036018	3,989214	3,997838
Boost TBB	1,04733	2,07885	4,093928	4,061814

Таблица 2. Зависимость времени выполнения программы от количества потоков (сек). *Ускорение OMP и TBB.*

По этому графику можно заметить, что на 2 потоках и TBB, и OpenMP имеют ускорение близкое к 2-м, а на 4 потоках близкое к 3,5 (3,39 и 3,89 соответственно), следовательно, можно сделать вывод о эффективном распараллеливании. Небольшие отклонения можно объяснить кэшированием данных, а также загруженностью процессора другими процессами. На 8 потоках ускорение также близко к 3,5, так как персональный компьютер, на котором тестировалась программа, ограничен 4 ядрами процессора. Так же по

этому графику можно сделать вывод, что для изображений 2000 на 2000 более эффективно использовать TBB версию, так как она дает большее ускорение.

Второй график показывает зависимость времени выполнении программы от размера изображений на 4 потоках и фильтре 3*3 (граф. 2). Также приведена соответствующая таблица (Табл. 3)

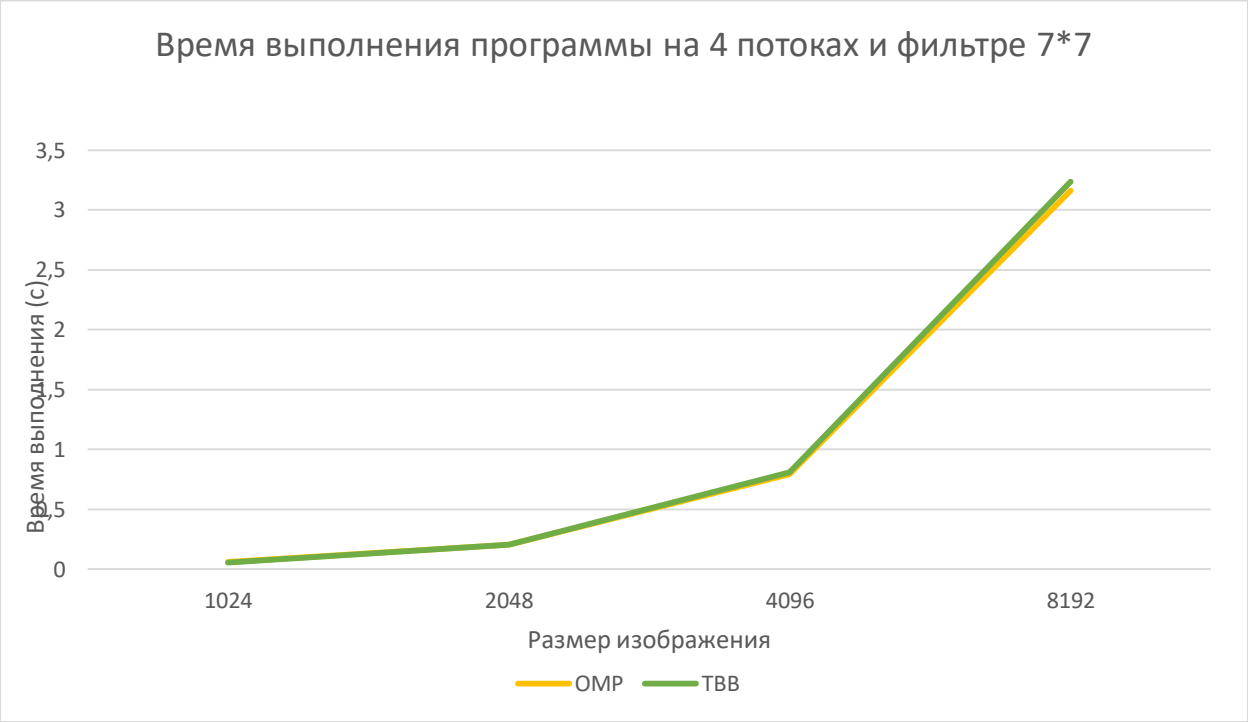

График 2. Зависимость времени выполнения программы от размера изображения

	128	256	512	1024	2048	4096	8192
Linear	0,0031	0,0133	0,051	0,1999	0,774	3,164	12,945
OMP	0,0011	0,037	0,0158	0,058	0,204	0,79	3,162
TBB	0,0014	0,037	0,0144	0,0529	0,202	0,81	3,238
Boost OMP	2,818182	0,359459	3,227848	3,446552	3,794118	4,005063	4,093928
Boost TBB	2,214286	0,359459	3,541667	3,778828	3,831683	3,906173	3,997838

Таблица 3. Зависимость времени выполнения программы от размера изображения (сек). Ускорение OpenMP и TBB.

По этому графику и таблице можно сделать вывод о том, что для обработки квадратных изображений размерами менее 512 и более 2048, целесообразно использовать OpenMP версию, а для обработки изображений от 512 до 2048 – TBB версию.

Третий график показывает зависимость времени выполнения программы на 4 потоках и для изображения 2048*2048 в зависимости от размера фильтра (граф. 3). Также приведена соответствующая таблица (Табл. 4).

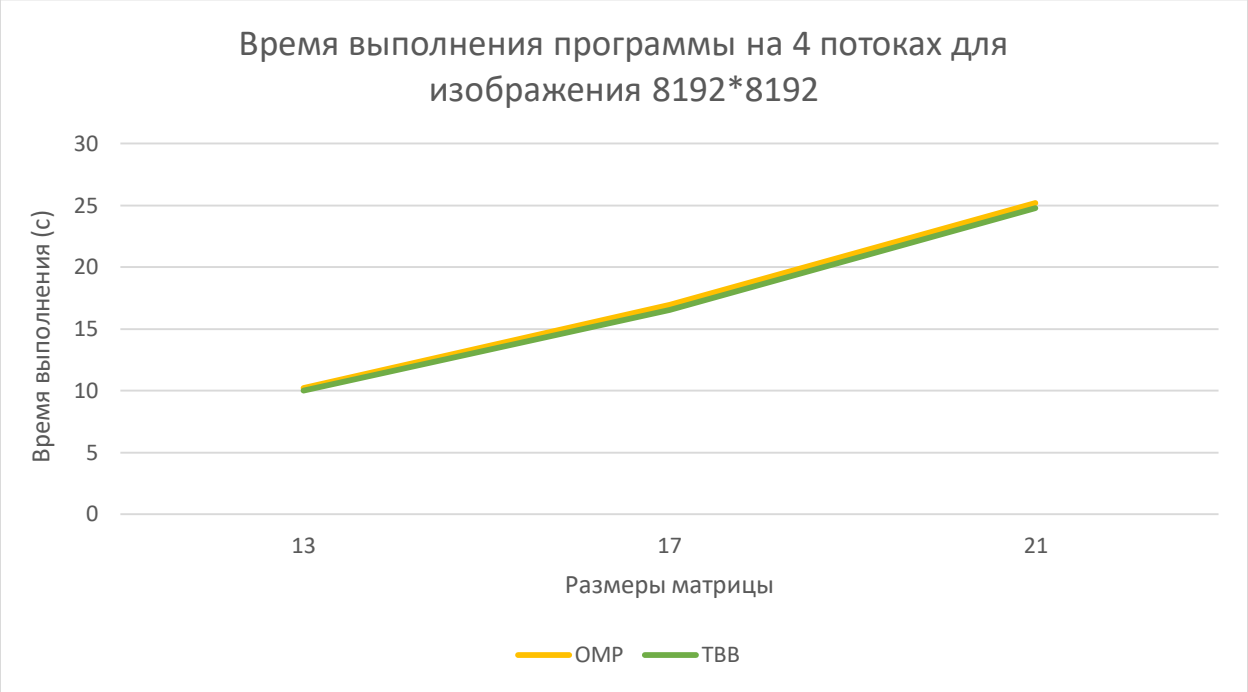

График 3. Зависимость времени выполнения программы от размера матрицы.

	3	7	13	17	21
Linear	2,7	12,945	39,527	65,672	97,832
OMP	0,697	3,245	10,207	16,9377	25,2152
TBB	0,665	3,162	10,01	16,5556	24,776
Boost OMP	3,873745	3,989214	3,872538	3,877268	3,879882
Boost TBB	4,06015	4,093928	3,948751	3,966754	3,94866

Таблица 4. Зависимость времени выполнения программы от размера матрицы (сек). Ускорение OMP и TBB

По данному графику видно, что с увеличением размера фильтра, ускорение OpenMP увеличивается, а ускорение TBB почти не меняется, т.е. зависимость линейная.

Вывод:

Ускорение и для OpenMP, и для TBB версии на большинстве проверенных размерах изображения и фильтров достигается ≈1,9 на 2-х потоках и ≈3,5 на 4-х потоках. Это большое ускорение можно объяснить тем, что задача применение линейного фильтра имеет сложность $O(x * y * m * n)$, где x, y – размеры изображения, а m, n – размеры фильтра, поэтому при достаточно больших изображениях и фильтрах затраты на создание и переключения между потоками занимают гораздо меньше времени, чем непосредственное применение фильтра.

Заключение:

В данной работе были реализованы последовательная и параллельная версия алгоритма линейной фильтрации изображения с линейным разбиением с использованием средств OpenMP и TBB, а также библиотекой OpenCV для работы с изображениями. Были проведены эксперименты и оценено ускорение параллельных версий. На основании результатов, можно сделать следующий вывод: Для изображений менее 512*512 и более 2048*2048 целесообразно использовать OpenMP версию программы, а для изображений между этими параметрами – TBB версию.

Литература и Источники:

- *Потапов А. А., Пахомов А. А., Никитин С. А., Гуляев Ю. В.*, Новейшие методы обработки изображений. — М.: Физматлит, 2008. — 496 с.
- Справочник по OpenCV. [Электронный ресурс] // https://opencv-tutorial.ru/
- *А. Ю. Дёмин, А. В. Кудинов*. Компьютерная графика. 2.3.4. Цифровые фильтры изображений. [Электронный ресурс] // http://compgraph.tpu.ru/filtrs.html

Код программы, реализующий последовательную обработку изображения фильтром Гаусса 3*3

```
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <iostream>
#include <fstream>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

#define NUMTHREAD 4

using namespace cv;
using namespace std;

//Функция, возвращающая само число, если оно находится в диапазоне (min, max), и
соответвующие края интервала в противном случае.
int clamp(int x, int min, int max) {
    if (x < min) return min;
    if (x > max) return max;
    return x;
}

//Функция получения случайного изображения с указанными размерами.
Mat getRandImage(int width, int height) {
    Mat resultImage(height, width, CV_8UC3);
    for (int i = 0; i < 3 * width * height; i++) {
        resultImage.data[i] = static_cast<unsigned char>(rand() % 256);
    }
    return resultImage;
}

//Функция получающая матрицу фильтра Гаусса.
double** getGausseFilter(int n) {
    int rad = n / 2;
    double** result = new double*[n];
    for (int i = 0; i < n; i++)
        result[i] = new double[n];
    double coeff1 = 1 / (2 * 3.14159265 * rad * rad);
    for (int i = -rad; i <= rad; i++) {
        for (int j = -rad; j <= rad; j++) {
        result[i + rad][j + rad] = coeff1 * exp((-(i*i + j * j)) / (2 * rad * rad));
        }
    }
    //Необходимая нормировка фильтра, во избежание затухания цвета на изображении
    double coeff2 = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            coeff2 += result[i][j];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            result[i][j] /= coeff2;
    return result;
}
```

```
//Функция обработки изображения
Mat filterImage(Mat image, double** filter, int n) {
    int width = image.cols;
    int height = image.rows;
    int rad = n / 2;
    Mat resultImage(height, width, CV_8UC3);
    for(int i = 0; i < height; i++)
        for (int j = 0; j < width; j++) {
            double resultR = 0;
            double resultG = 0;
            double resultB = 0;
            for(int fi = -rad; fi <= rad; fi++)
                for (int fj = -rad; fj <= rad; fj++) {
                    //Получение действительных пикселей, во избежание выхода за
                    //границу изображения. Необходимо для обработки крайних элементов
                    int curI = clamp(i + fi, 0, height - 1);
                    int curJ = clamp(j + fj, 0, width - 1);
                    resultR+=image.data[3*(curI*width+curJ)]*filter[fi+rad][fj+rad];
                    resultG+=image.data[3*(curI*width+curJ)+1]*filter[fi+rad][fj+rad];
                    resultB+=image.data[3*(curI*width+curJ)+2]*filter[fi+rad][fj+rad];
                }
            resultImage.data[3*(i*width+j)] = clamp((int)(resultR), 0, 255);
            resultImage.data[3*(i*width+j)+1] = clamp((int)(resultG), 0, 255);
            resultImage.data[3*(i*width+j)+2] = clamp((int)(resultB), 0, 255);
        }
    return resultImage;
}

int main(int argv, char** argc) {
    srand((unsigned int)time(0));
    setlocale(LC_ALL, "Russian");

    int n = 3;
    if (n % 2 == 0) n++;
    double** filter = getGausseFilter(n);

    char* path = "S:\\Projects\\C++\\Files\\image.jpg";
    Mat image = imread(path);

    if (image.rows == 0 || image.cols == 0) {
        cout << "File not found! Generate rand image" << endl;
        int width = 2048;
        int height = 2048;
        image = getRandImage(width, height);
        cout << "Image generated" << endl;
    }

    clock_t startTime = clock();
    Mat resultImage = filterImage(image, filter, n);
    clock_t finishTime = clock();
    cout << "Consistent work completed. Elapsed time = " << ((double)(finishTime -
startTime) / CLOCKS_PER_SEC) << endl;
    time += ((double)(finishTime - startTime) / CLOCKS_PER_SEC);
    imwrite("S:\\Projects\\C++\\Files\\resultImage.bmp", resultImage);
}
```

Код программы, реализующий параллельную обработку изображения фильтром Гаусса 3*3, используя средства OpenMP

```
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <omp.h>
#include <iostream>
#include <fstream>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

#define NUMTHREAD 4

using namespace cv;
using namespace std;

//Функция, возвращающая само число, если оно находится в диапазоне (min, max), и
соответвующие края интервала в противном случае.
int clamp(int x, int min, int max) {
    if (x < min) return min;
    if (x > max) return max;
    return x;
}

//Функция получения случайного изображения с указанными размерами.
Mat getRandImage(int width, int height) {
    Mat resultImage(height, width, CV_8UC3);
    for (int i = 0; i < 3 * width * height; i++) {
        resultImage.data[i] = static_cast<unsigned char>(rand() % 256);
    }
    return resultImage;
}

//Функция получающая матрицу фильтра Гаусса.
double** getGausseFilter(int n) {
    int rad = n / 2;
    double** result = new double*[n];
    for (int i = 0; i < n; i++)
        result[i] = new double[n];
    double coeff1 = 1 / (2 * 3.14159265 * rad * rad);
    for (int i = -rad; i <= rad; i++) {
        for (int j = -rad; j <= rad; j++) {
        result[i + rad][j + rad] = coeff1 * exp((-(i*i + j * j)) / (2 * rad * rad));
        }
    }
    //Необходимая нормировка фильтра, во избежание затухания цвета на изображении
    double coeff2 = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            coeff2 += result[i][j];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            result[i][j] /= coeff2;
    return result;
}
```

```
//Функция обработки изображения, использующая средства OpenMP
Mat filterImageOmpPar(Mat image, double** filter, int n) {
    int width = image.cols;
    int height = image.rows;
    int rad = n / 2;
    Mat resultImage(height, width, CV_8UC3);
#pragma omp parallel num_threads(NUMTHREAD)
#pragma omp for schedule(static)
    for (int i = 0; i < height; i++)
        for (int j = 0; j < width; j++) {
            double resultR = 0;
            double resultG = 0;
            double resultB = 0;
            for (int fi = -rad; fi <= rad; fi++)
                for (int fj = -rad; fj <= rad; fj++) {
                    int curI = clamp(i + fi, 0, height - 1);
                    int curJ = clamp(j + fj, 0, width - 1);
                    resultR+=image.data[3*(curI*width+curJ)]*filter[fi+rad][fj+rad];
                    resultG+=image.data[3*(curI*width+curJ)+1]*filter[fi+rad][fj+rad];
                    resultB+=image.data[3*(curI*width+curJ)+2]*filter[fi+rad][fj+rad];
                }
            resultImage.data[3 * (i*width + j)] = clamp((int)(resultR), 0, 255);
            resultImage.data[3 * (i*width + j) + 1] = clamp((int)(resultG), 0, 255);
            resultImage.data[3 * (i*width + j) + 2] = clamp((int)(resultB), 0, 255);
        }
    return resultImage;
}

int main(int argv, char** argc) {
    srand((unsigned int)time(0));
    setlocale(LC_ALL, "Russian");

    int n = 3;
    if (n % 2 == 0) n++;
    double** filter = getGausseFilter(n);

    char* path = "S:\\Projects\\C++\\Files\\image.jpg";
    Mat imageOmp = imread(path);

    if (image.rows == 0 || image.cols == 0) {
        cout << "File not found! Generate rand image" << endl;
        int width = 2048;
        int height = 2048;
        imageOmp = getRandImage(width, height);
        cout << "Image generated" << endl;
    }

    double startTimeOmpPar = omp_get_wtime();
    Mat resultImageOmpPar = filterImageOmpPar(imageOmp, filter, n);
    double finishTimeOmpPar = omp_get_wtime();
    cout << "Parallel Omp work completed. Elapsed time = " << finishTimeOmpPar -
startTimeOmpPar << endl;
    omptime += finishTimeOmpPar - startTimeOmpPar;
    imwrite("S:\\Projects\\C++\\Files\\resultImageOmpPar.bmp", resultImageOmpPar);
}
```

Код программы, реализующий параллельную обработку изображения фильтром Гаусса 3*3, используя средства OpenMP

```
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <tbb/tbb.h>
#include <iostream>
#include <fstream>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

#define NUMTHREAD 4

using namespace cv;
using namespace std;
using namespace tbb;

//Функция, возвращающая само число, если оно находится в диапазоне (min, max), и соответвующие края интервала в противном случае.
int clamp(int x, int min, int max) {
    if (x < min) return min;
    if (x > max) return max;
    return x;
}

//Функция получения случайного изображения с указанными размерами.
Mat getRandImage(int width, int height) {
    Mat resultImage(height, width, CV_8UC3);
    for (int i = 0; i < 3 * width * height; i++) {
        resultImage.data[i] = static_cast<unsigned char>(rand() % 256);
    }
    return resultImage;
}

//Функция получающая матрицу фильтра Гаусса.
double** getGausseFilter(int n) {
    int rad = n / 2;
    double** result = new double*[n];
    for (int i = 0; i < n; i++)
        result[i] = new double[n];
    double coeff1 = 1 / (2 * 3.14159265 * rad * rad);
    for (int i = -rad; i <= rad; i++) {
        for (int j = -rad; j <= rad; j++) {
        result[i + rad][j + rad] = coeff1 * exp((-(i*i + j * j)) / (2 * rad * rad));
        }
    }
    //Необходимая нормировка фильтра, во избежание затухания цвета на изображении
    double coeff2 = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            coeff2 += result[i][j];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            result[i][j] /= coeff2;
    return result;
}
```

```
//Класс-функтор обрабатывающий изображение
class FilteringImage {

private:

    Mat image;
    int n;
    double** filter;
    Mat* resultImage;

public:

    FilteringImage(Mat _image, double** _filter, int _n) {
        image = _image;
        resultImage = new Mat(image.rows, image.cols, CV_8UC3);
        n = _n;
        filter = _filter;
    }

    void operator() (const blocked_range<int>& r) const {
        int width = image.cols;
        int height = image.rows;
        int rad = n / 2;
        for (int i = r.begin(); i < r.end(); i++)
            for (int j = 0; j < width; j++) {
                double resultR = 0;
                double resultG = 0;
                double resultB = 0;
                for (int fi = -rad; fi <= rad; fi++)
                    for (int fj = -rad; fj <= rad; fj++) {
                        int curI = clamp(i + fi, 0, height - 1);
                        int curJ = clamp(j + fj, 0, width - 1);
                        resultR+=image.data[3*(curI*width+curJ)]*filter[fi+rad][fj+
rad];
                        resultG+=image.data[3*(curI*width+curJ)+1]*filter[fi+rad][fj+
rad];
                        resultB+=image.data[3*(curI*width+curJ)+2]*filter[fi+rad][fj+
rad];
                    }
                resultImage->data[3*(i*width + j)] = clamp((int)(resultR), 0, 255);
                resultImage->data[3*(i*width + j)+1] = clamp((int)(resultG), 0, 255);
                resultImage->data[3*(i*width + j)+2] = clamp((int)(resultB), 0, 255);
            }
    }

    Mat getResutlImage() {
        return *resultImage;
    }
};


//Функция обработки изображения, использующая средства TBB
Mat filterImageTbbPar(Mat image, double** filter, int n) {
    task_scheduler_init init(NUMTHREAD);
    Mat resultImage(image.rows, image.cols, CV_8UC3);
    FilteringImage filtImg(image, filter, n);
    parallel_for(blocked_range<int>(0, image.rows, image.rows/NUMTHREAD), filtImg);
    return filtImg.getResutlImage();
    init.terminate();
}
```

```
int main(int argv, char** argc) {
    srand((unsigned int)time(0));
    setlocale(LC_ALL, "Russian");

    int n = 3;
    if (n % 2 == 0) n++;
    double** filter = getGausseFilter(n);

    char* path = "SS:\\Projects\\C++\\Files\\image.jpg";
    Mat imageTbb = imread(path);

    if (image.rows == 0 || image.cols == 0) {
        cout << "File not found! Generate rand image" << endl;
        int width = 2048;
        int height = 2048;
        imageTbb = getRandImage(width, height);
        cout << "Image generated" << endl;
    }

    tick_count startTimeTbbPar = tick_count::now();
    Mat resultImageTbbPar = filterImageTbbPar(imageTbb, filter, n);
    tick_count finishTimeTbbPar = tick_count::now();
    cout << "Parallel Tbb work completed. Elapsed time = " << (finishTimeTbbPar -
startTimeTbbPar).seconds() << endl;
    tbbtime += (finishTimeTbbPar - startTimeTbbPar).seconds();
    imwrite("S:\\Projects\\C++\\Files\\resultImageTbbPar.bmp", resultImageTbbPar);
}
```