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

Направление подготовки: «Фундаментальная информатика и информационные технологии»

Отчет

по лабораторной работе

«Линейная фильтрация изображений (вертикальное разбиение). Ядро Гаусса 3×3.»

Выполнила:
студентка группы 381606-1
ТужилкинаА.А.
Проверил:
Доцент кафедры МОиСТ, к.т.н.
Сысоев А.В.

Нижний Новгород
2019

Содержание

Введение

Одна из самых важных задач при работе с изображениями связана с их предварительной обработкой - выделением и фильтрацией шума. 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 обусловлена интенсивностями (цветами) пикселей, расположенных в некоторой его окрестности в исходном изображении. При этой обработке необходимо обеспечить максимальное сохранение деталей изображения. Фильтрация искаженных пикселей относится к группе низкоуровневых операций обработки изображения. При последовательной обработке каждого пикселя время обработки изображений достаточно велико. Это неприемлемо в реальном времени для решения различных прикладных задач. Поэтому эту проблему можно решить, если использовать высокопроизводительные параллельные вычислительные машины.

В данной лабораторной работе ставится задача реализовать программу, использующую средства параллельного программирования OpenMP и TBB, которая фильтрует изображение, используя ядро Гаусса 3×3. Размытие по Гауссу — это характерный фильтр размытия изображения, который использует нормальное распределение (также называемое Гауссовым распределением, отсюда название) для вычисления преобразования, применяемого к каждому пикселю изображения.

Постановка задачи

Необходимо составить программу, которая выполняет фильтрацию изображения с помощью фильтра Гаусса с ядром 3*3 и использует вертикальное разбиение, задействовав основные технологии параллельного программирования для систем с общей памятью (OpenMP и TBB).

Цель:

- Реализация последовательной версии программы.
- Реализация параллельной версии программы с помощью технологии OpenMP.
- Реализация параллельной версии программы с помощью технологии TBB.
- Оценка эффективности программ.
- Проанализировать полученные результаты и сделать выводы.

Описание алгоритма

Итак, рассмотрим алгоритм на примере изображения 5×5 пикселей.

1. Сначала у нас создается матрица изображения с помощью функции InitMatr(rows, cols, picture), представленной на Рисунке 1.

```
void InitMatr(int rows, int cols, int** m) {
  for (int i = 0; i < rows; i++)
    for (int j = 0; j < cols; j++)
      m[i][j] = static_cast<int>(std::rand() % 255);
}
```

Рисунок 1. Реализация функции InitMatr.

2. Затем создаём главную часть матричного фильтра – ядро – это матрица коэффициентов, которая покомпонентно умножается на значение пикселей изображения для получения требуемого результата (не то же самое, что матричное умножение, коэффициенты матрицы являются весовыми коэффициентами для выбранного подмассива изображения) с помощью функции InitKern(kernel, 1, 1.0), которую можем увидеть на Рисунке 2.

Для размытия изображений используют фильтр Гаусса, коэффициенты для которого рассчитываются по формуле Гаусса:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

```
void InitKern(double kernel[3][3], int radius, double sigma) {
  double norm = 0;

  for (int i = -radius; i <= radius; i++)
    for (int j = -radius; j <= radius; j++) {
      kernel[i + radius][j + radius] = (exp(-(i * i + j * j) / (2 * sigma * sigma)));
      norm += kernel[i + radius][j + radius];
    }

  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
      kernel[i][j] /= norm;
}
```

Рисунок 2. Реализация функции InitKern.

3. После этого происходит сама фильтрация изображения с помощью функции Gauss_seq(rows, cols, kernel, picture, res_seq), которая представлена на Рисунке 3. Обработка краёв происходит дважды, для примера рассмотрим Рисунок 4 и Рисунок 5, на котором представлена реализация функции Clamp().

```
void Gauss_seq(int rows, int cols, double kernel[3][3], int **picture1, int **picture2) {
  double temp;
  for (int j = 0; j < cols; j++)
    for (int i = 0; i < rows; i++) {
      temp = 0.0;
      for (int q = -1; q <= 1; q++)
        for (int l = -1; l <= 1; l++) {
          int idX = Clamp(i + q, 0, rows - 1);
          int idY = Clamp(j + l, 0, cols - 1);
          temp += picture1[idX][idY] * kernel[q + 1][l + 1];
        }
      picture2[i][j] = static_cast<int>(temp);
    }
}
```

Рисунок 3. Реализация функции Gauss_seq.

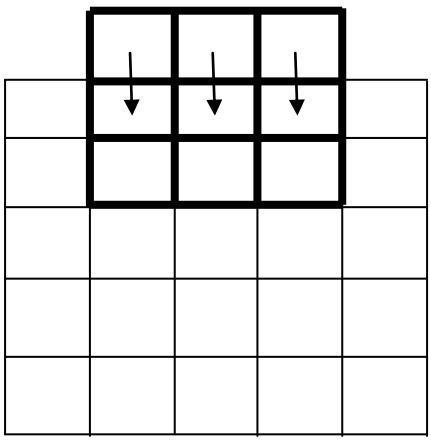

Рисунок 4. Обработка краёв.

```
int Clamp(int value, int min, int max) {
    if (value < min)
        return min;

    if (value > max)
        return max;

    return value;
}
```

Рисунок 5. Реализация функции Clamp.

4. Пример результата работы фильтра с ядром Гаусса 3 × 3 представлен на Рисунке 6.

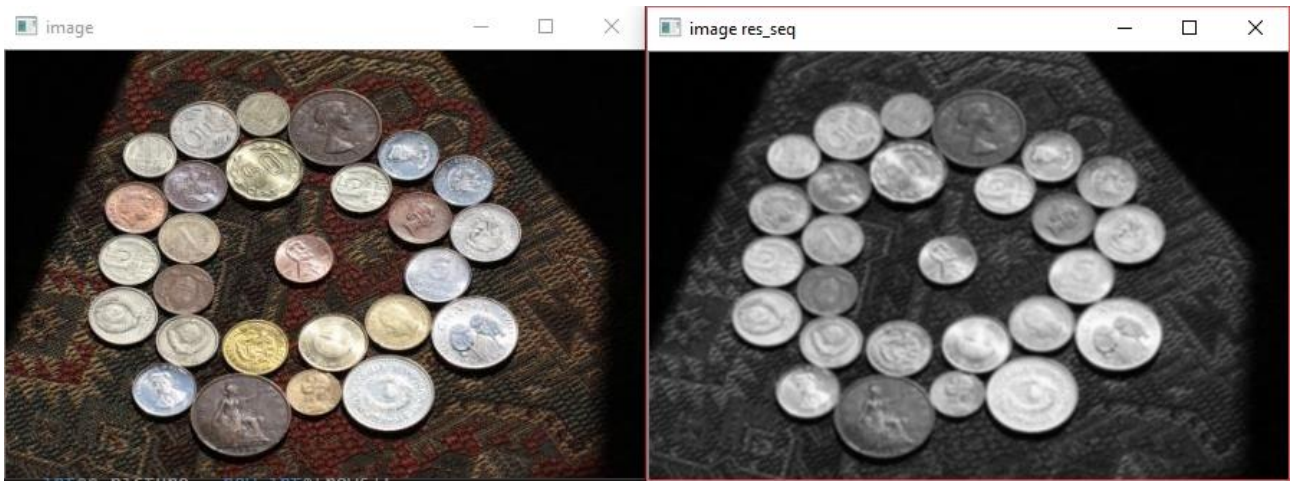

Рисунок 6. Результат работы.

Описание схемы распараллеливания

Схема распараллеливания заключается в том, чтобы распараллелить внешний цикл в обеих версиях программы. В OpenMP с помощью #pragma omp parallel for, а в TBB - tbb::parallel for. Так как алгоритм обработки практически всегда линейный.

Описание OpenMP-версии

OpenMP (Open Multi-Processing) — открытый стандарт для распараллеливания программ на языках Си, Си++ и Фортран. Дает описание совокупности директив компилятора, библиотечных функций и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью. OpenMP реализует параллельные вычисления с помощью многопоточности, в которой «главный» (master) поток создает набор подчиненных (slave) потоков и задача распределяется между ними. Предполагается, что потоки выполняются параллельно на машине с несколькими процессорами (количество процессоров не обязательно должно быть больше или равно количеству потоков). Задачи, выполняемые потоками параллельно, так же, как и данные, требуемые для выполнения этих задач, описываются с помощью специальных директив препроцессора соответствующего языка — прагм. Количество создаваемых потоков может регулироваться как самой программой при помощи вызова библиотечных процедур, так и извне, при помощи переменных окружения.

Так как алгоритмы обработки изображений почти всегда линейны, каких-либо проблем с разделением задачи между потоками не возникло, в задании указано вертикальное разбиение, поэтому перед внешним циклом (по столбцам) была указана программная директива #pragma omp parallel for firstprivate(temp). Реализация представлена на Рисунке 7.

Был выбран параметр firstprivate, т.к. он позволяет создать локальные переменные потоков, которые перед использованием инициализируются значениями исходных переменных.

```
void Gauss_par(int rows, int cols, double kernel[3][3], int **picture1, int **picture2) {
    double temp = 0.0;
    omp_set_num_threads(2);

#pragma omp parallel for firstprivate(temp)
    for (int j = 0; j < cols; j++)
        for (int i = 0; i < rows; i++) {
            temp = 0.0;
            for (int q = -1; q <= 1; q++)
                for (int l = -1; l <= 1; l++) {
                    int idX = Clamp(i + q, 0, rows - 1);
                    int idY = Clamp(j + l, 0, cols - 1);
                    temp += picture1[idX][idY] * kernel[q + 1][l + 1];
                }
            picture2[i][j] = static_cast<int>(temp);
        }
}
```

Рисунок 7. Реализация параллельной версии программы с помощью технологии OpenMP.

Описание TBB-версии

Intel Threading Building Blocks — кроссплатформенная библиотека шаблонов C++, разработанная компанией Intel для параллельного программирования. Библиотека содержит алгоритмы и структуры данных, позволяющие программисту избежать многих сложностей, возникающих при использовании традиционных реализаций потоков, таких как POSIX Threads, Windows threads или Boost Threads, в которых создаются отдельные потоки исполнения, синхронизируемые и останавливаемые вручную. Библиотека TBB абстрагирует доступ к отдельным потокам. Все операции трактуются как «задачи», которые динамически распределяются между ядрами процессора. Кроме того, достигается эффективное использование кэша. Программа, написанная с использованием TBB, создаёт, синхронизирует и разрешает графы зависимостей задач в соответствии с алгоритмом. Затем задачи исполняются в соответствии с зависимостями. Этот подход позволяет программировать параллельные алгоритмы на высоком уровне, абстрагируясь от деталей архитектуры конкретной машины.

Параллельный код, с использование библиотеки Intel TBB, не сильно отличается от предыдущей версии, но было решено использовать лямбда-выражение. Это позволяет значительно сократить объем программного кода, не объявляя класс функтора.

```
void par_mat(double kernel[3][3], int **picture1, int **picture2, int j, size_t size_rows, size_t size_cols) {
    for (size_t i = 0; i < size_rows; i++) {
        double temp = 0;
        for (int q = -1; q <= 1; q++)
            for (int l = -1; l <= 1; l++) {
                int idX = Clamp(i + q, 0, size_rows - 1);
                int idY = Clamp(j + l, 0, size_cols - 1);

                temp += picture1[idX][idY] * kernel[q + 1][l + 1];
            }
        picture2[i][j] = static_cast<int>(temp);
    }
}

void parallel_matrix_multiply(double kernel[3][3], int **picture1, int **picture2, size_t size_rows, size_t size_cols) {
    tbb::parallel_for(tbb::blocked_range<size_t>(0, size_cols), [=](const tbb::blocked_range<size_t>& r) {
        for (size_t j = r.begin(); j != r.end();j++) {
            par_mat(kernel, picture1, picture2, j, size_rows, size_cols);
        }
    });
}
```

Рисунок 8. Реализация параллельной версии программы с помощью технологии TBB.

Результаты экспериментов

Эксперименты проводились на ПК со следующими характеристиками:

- Процессор: Intel(R) Core(TM) i3-7100 CPU @ 3.90GHz 3.90GHz.
- Оперативная память: 8,00ГБ.
- ОС: Windows 10 для образовательных учреждений.

Рассмотрим приведённую ниже таблицу 1 для оценки результатов, время дано в секундах:

Таблица 1.

Размер изображения	Последовательный алгоритм	OpenMP		TBB		Количество потоков
		Время	Ускорение	Время	Ускорение	
1000×1000	1.82397	1.84547	0.98835	1.86518	0.97791	1
		1.10452	1.65137	1.05778	1.72434	2
		0.73566	2.47937	0.69437	2.62679	4
		0.84586	2.15635	0.66519	2.74203	8
3000×5000	28.1427	28.3104	0.99407	28.6192	0.98335	1
		17.1025	1.64553	16.3455	1.72173	2
		10.0935	2.7882	9.8852	2.84695	4
		11.4166	2.46506	9.8255	2.86425	8
5000×5000	46.5935	46.7709	0.9962	46.7814	0.99598	1
		27.3857	1.70138	27.7194	1.68089	2
		16.7565	2.78062	16.5165	2.82102	4
		17.8471	2.6107	16.3552	2.84884	8

Выводы из результатов

По данным таблицы 1 были составлены следующие диаграммы:

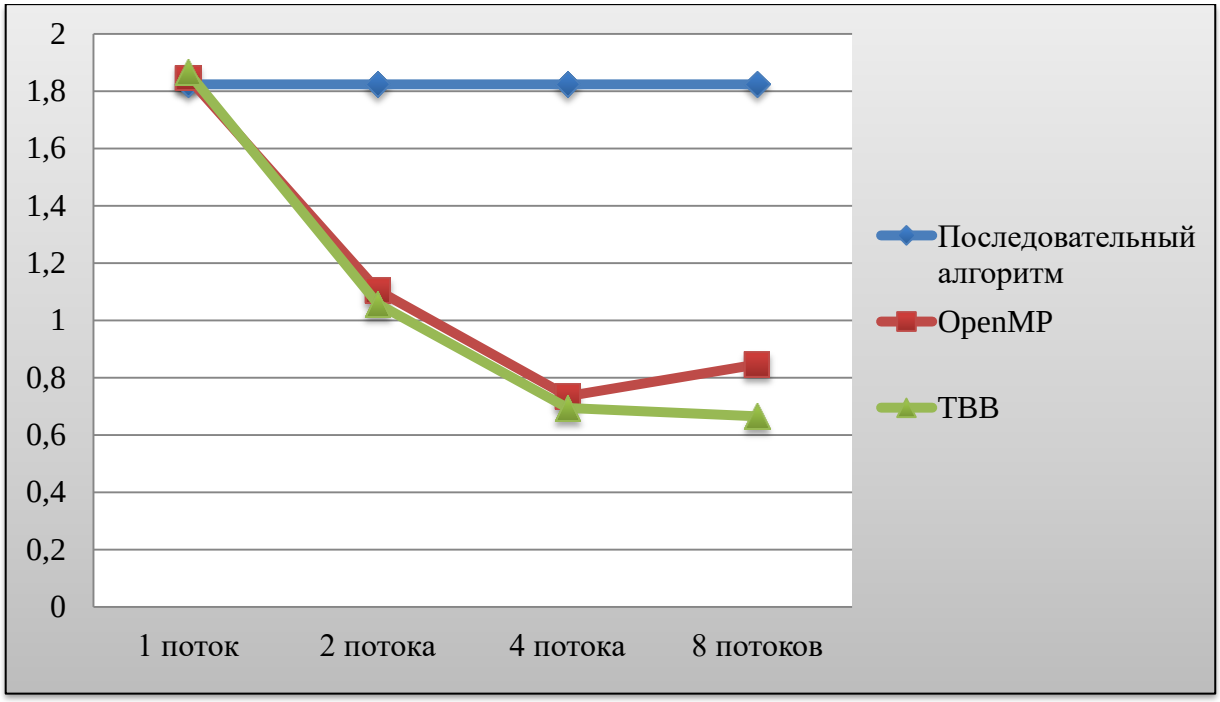

Диаграмма 1. Сравнение времени выполнения программ при различном количестве потоков для изображения 1000×1000.

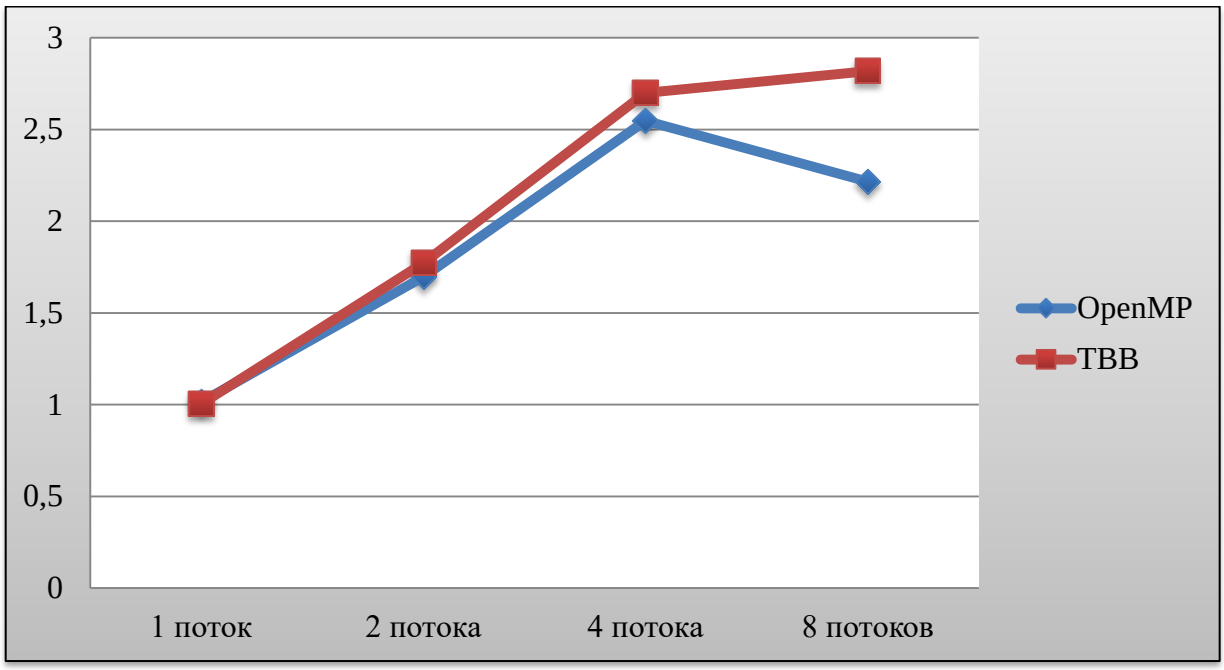

Диаграмма 2. Сравнение ускорений OpenMP и TBB для изображения 1000×1000.

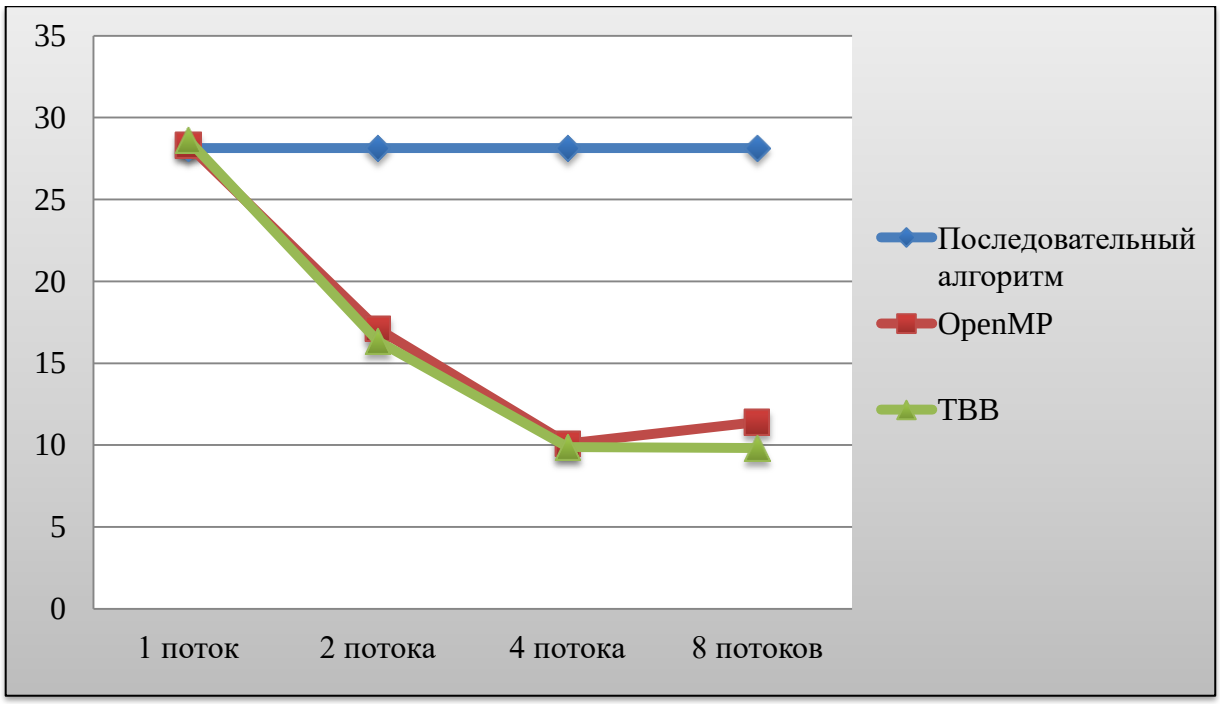

Диаграмма 3. Сравнение времени выполнения программ при различном количестве потоков для изображения 3000×5000.

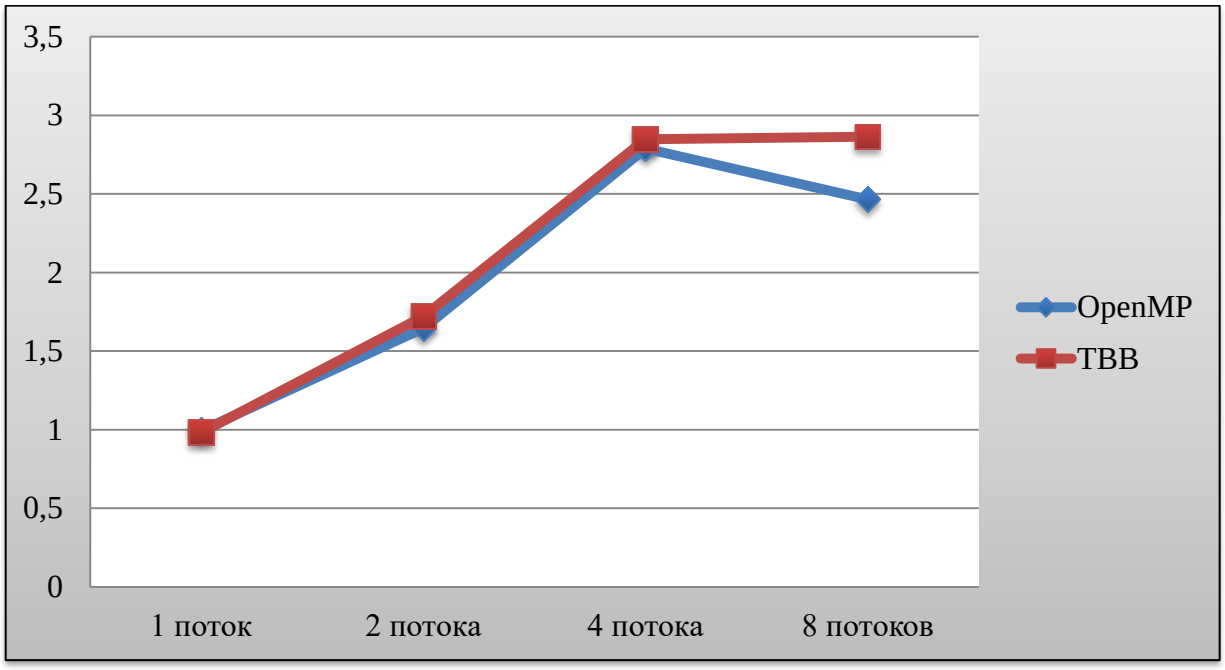

Диаграмма 4. Сравнение ускорений OpenMP и TBB для изображения 3000×5000.

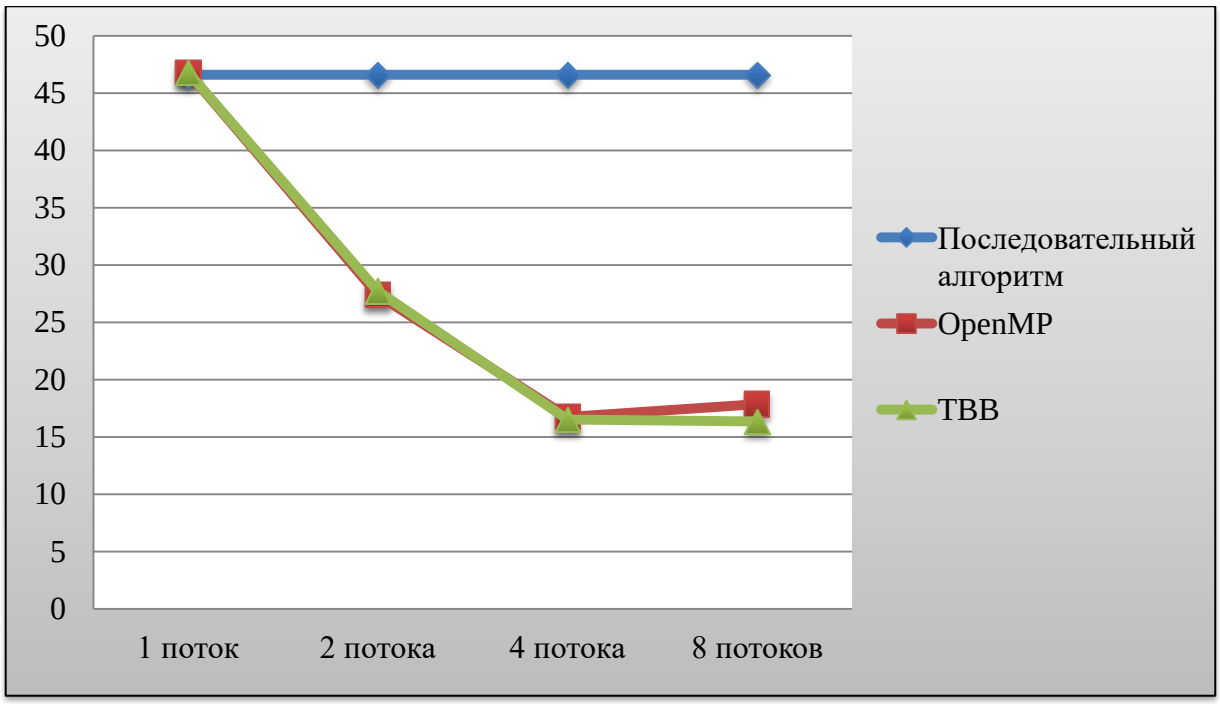

Диаграмма 1. Сравнение времени выполнения программ при различном количестве потоков для изображения 5000×5000.

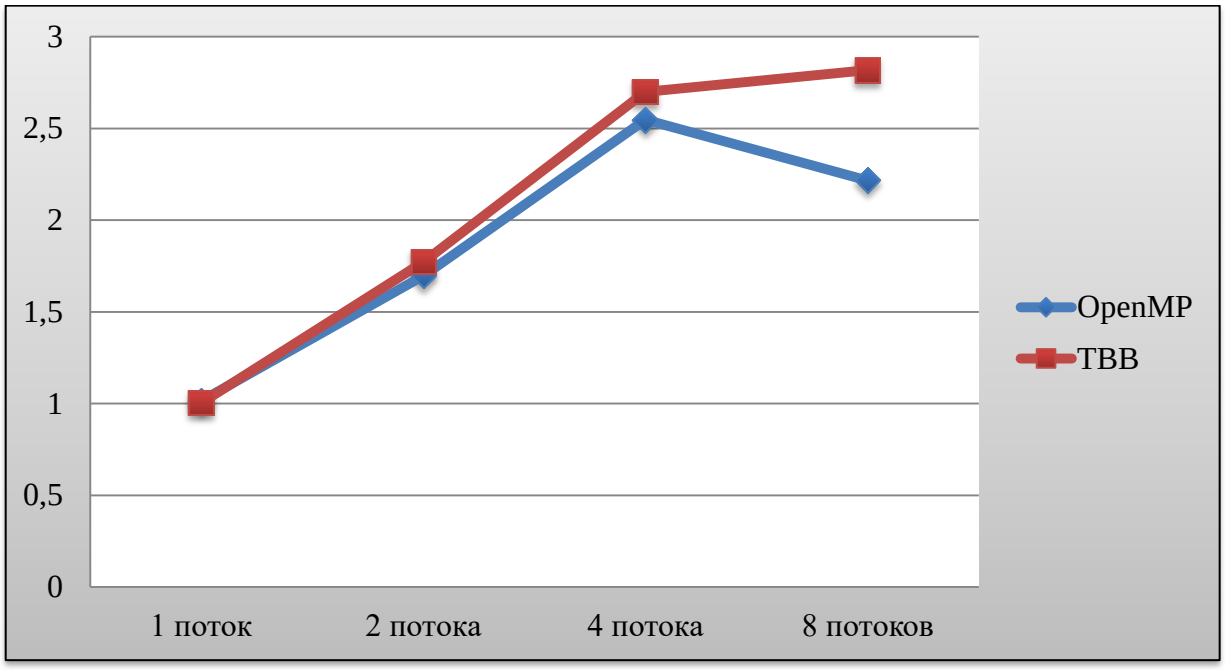

Диаграмма 2. Сравнение ускорений OpenMP и TBB для изображения 5000×5000.

Нетрудно заметить, что последовательная версия при 1 потоке работает быстрее, чем версии OpenMP и TBB, т.к. они затрачивают время на накладные расходы. С увеличением количества потоков параллельные версии начинают работать быстрее последовательной за счет распараллеливания вычислений. Но при количестве потоков больше 4, ускорение не

увеличивается, т.к. тестируем программу на 4-х ядерном компьютере и увеличение количества потоков ни к чему не приведет. Из-за увеличения количества потоков программа может работать медленнее за счет увеличения накладных расходов на распараллеливание. Ускорение для версий OpenMP и TBB на различных проверенных размерах изображений достигает ≈ 1.7 при использовании двух потоков и достигает ≈ 2.7 при использовании четырех потоков.

Заключение

В данной лабораторной работе были разработаны параллельные алгоритмы фильтрации изображений, используя ядро Гаусса 3×3. Исходя из результатов экспериментов, представленных выше, можно сделать следующий вывод. TBB и OpenMP версии показали примерно одинаковые результаты. Чем больше размер изображения, тем наиболее эффективно распараллеливание, т.к. использование высокоуровневых библиотек значительно упрощает задачу параллелизма и позволяет без особых временных затрат получать солидный прирост в производительности. Но не стоит забывать о том, что параллелизм, при не большом объеме работ, может привести к замедлению алгоритма.

Литература

1. Справочник по TBB. [Электронный ресурс] https://pt.slideshare.net/michaelkarpov/tbb
2. Справочник по OpenMP, часть 1. [Электронный ресурс] https://ppt-online.org/570815
3. Справочник по OpenMP, часть 2. [Электронный ресурс] https://ppt-online.org/570816
4. Матричные фильтры обработки изображений. [Электронный ресурс] https://habr.com/post/142818