Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчёт по лабораторной работе №2 по дисциплине "Анализ алгоритмов"

Тема Умножение матриц

Студент Варламова Е.А.

Группа ИУ7-51Б

Оценка (баллы) _____

Преподаватели Волкова Л.Л.

Москва — 2021 г.

Оглавление

Введение

Умножение матриц — это один из базовых алгоритмов, который широко применяется в численных методах, в частности, в машинном обучении, в компьютерной графике. Именно поэтому важно, чтобы алгоритм был максимально эффективен по затрачиваемым ресурсам. Так, **целью** данной работы является изучение алгоритмов умножения матриц, в частности: обычный алгоритм, алгоритм Винограда и оптимизированный алгоритм Винограда.

Для достижения поставленной цели необходимо решить следующие задачи.

1. Изучение двух алгоритмов умножения матриц: обычного и алгоритма Винограда.
2. Разработка оптимизированного алгоритма умножения матриц.
3. Реализация трёх алгоритмов умножения матриц: обычного, алгоритма Винограда и оптимизированного алгоритма Винограда.
4. Сравнительный анализ трудоёмкости алгоритмов на основе расчетов в выбранной модели вычислений.
5. Сравнительный анализ алгоритмов на основе экспериментальных данных.

1 Аналитическая часть

В данном разделе рассматриваются два алгоритма умножения матриц - стандартный и Винограда. При этом предполагается, что количество столбцов первой умножаемой матрицы совпадает с количеством строк второй матрицы.

1.1 Стандартный алгоритм

Пусть даны две прямоугольные матрицы

$$A_{rs} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1s} \\ a_{21} & a_{22} & \dots & a_{2s} \\ \vdots & \vdots & \ddots & \vdots \\ a_{r1} & a_{r2} & \dots & a_{rs} \end{pmatrix}, \quad B_{sc} = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1c} \\ b_{21} & b_{22} & \dots & b_{2c} \\ \vdots & \vdots & \ddots & \vdots \\ b_{s1} & b_{s2} & \dots & b_{sc} \end{pmatrix}, \tag{1.1}$$

тогда матрица C

$$C_{rc} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1c} \\ c_{21} & c_{22} & \dots & c_{2c} \\ \vdots & \vdots & \ddots & \vdots \\ c_{r1} & c_{r2} & \dots & c_{rc} \end{pmatrix}, \tag{1.2}$$

где

$$c_{ij} = \sum_{t=1}^{s} a_{it} b_{tj} \quad (i = \overline{1, r}; j = \overline{1, c}) \tag{1.3}$$

будет называться произведением матриц A и B. Стандартный алгоритм реализует данную формулу.

1.2 Алгоритм Винограда

Если посмотреть на результат умножения двух матриц, то видно, что каждый элемент в нем представляет собой скалярное произведение соответствующих строки и столбца исходных матриц. Можно заметить также, что такое умножение допускает предварительную обработку, позволяющую часть работы выполнить заранее.

Рассмотрим два вектора $V = (v_1, v_2, v_3, v_4)$ и $W = (w_1, w_2, w_3, w_4)$. Их скалярное произведение равно: $V \cdot W = v_1w_1 + v_2w_2 + v_3w_3 + v_4w_4$, что эквивалентно (1.4):

$$V \cdot W = (v_1 + w_2)(v_2 + w_1) + (v_3 + w_4)(v_4 + w_3) - v_1v_2 - v_3v_4 - w_1w_2 - w_3w_4. \tag{1.4}$$

Несмотря на то, что второе выражение требует вычисления большего количества операций, чем стандартный алгоритм: вместо четырёх умножений - шесть, а вместо трёх сложений - десять, выражение в правой части последнего равенства допускает предварительную обработку: его части можно вычислить заранее и запомнить для каждой строки первой матрицы и для каждого столбца второй, что позволит для каждого элемента выполнять лишь два умножения и пять сложений, складывая затем только лишь с 2 предварительно посчитанными суммами соседних элементов текущих строк и столбцов. Из-за того, что операция сложения быстрее операции умножения в ЭВМ, на практике алгоритм должен работать быстрее стандартного.

Формула 1.4 подходит лишь для того случая, когда длина векторов чётная. Соответственно, если количество столбцов первой матрицы (или количество строк второй) нечётное, то требуется сделать дополнительный проход по результирующей матрице и добавить в её элементы недостающие произведения.

1.3 Вывод

В данном разделе были рассмотрены алгоритмы классического умножения матриц и алгоритм Винограда. Было выявлено, что алгоритм Винограда отличается от стандартного наличием предварительной обработки строк и столбцов, что позволяет сократить количество умножений, а значит ускорить алгоритм.

2 Конструкторская часть

В данном разделе разрабатываются схемы алгоритмов, описанных в аналитическом разделе, а также даётся оценка трудоёмкости этих алгоритмов.

2.1 Схемы алгоритмов

На рисунках 2.1, 2.2 и 2.3 приведены схемы стандартного алгоритма умножения матриц и алгоритма Винограда (обычного и оптимизированного) соответственно. Предполагается, что размеры первой матрицы (n1, m1), а второй - (n2, m2).

Алгоритм Винограда оптимизируется следующим образом.

1. Сокращается количество умножений в основном цикле и циклах предварительной обработки строк и столбцов.

2. Дополнительный цикл, работающий при нечётном количестве столбцов первой матрицы (строк второй), заменяется на условный оператор, помещённый в основной цикл.

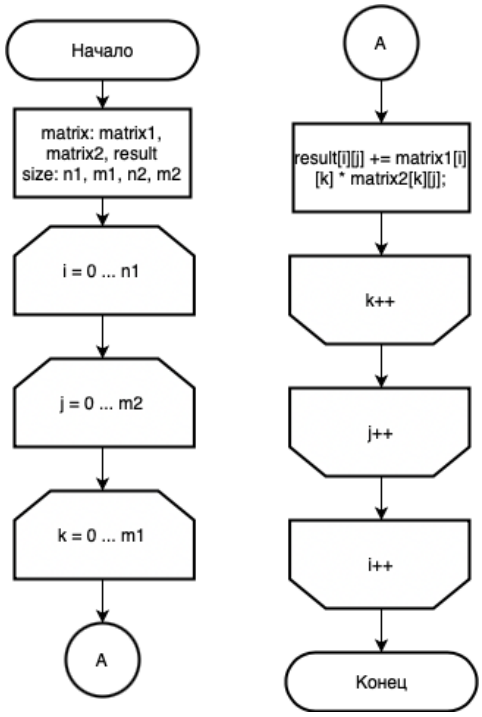

Рис. 2.1: Схема стандартного алгоритма умножения матриц

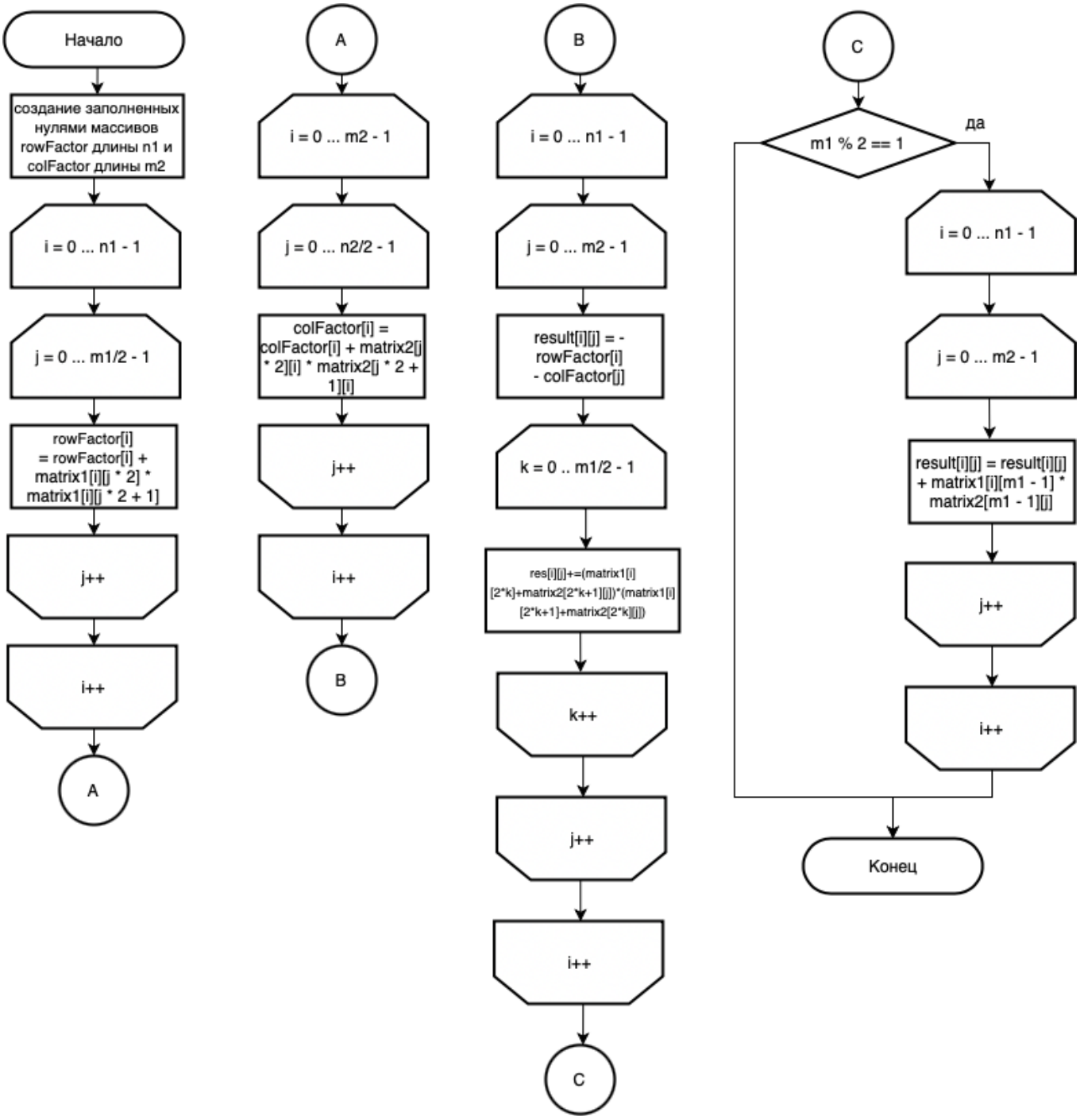

Рис. 2.2: Схема алгоритма Винограда

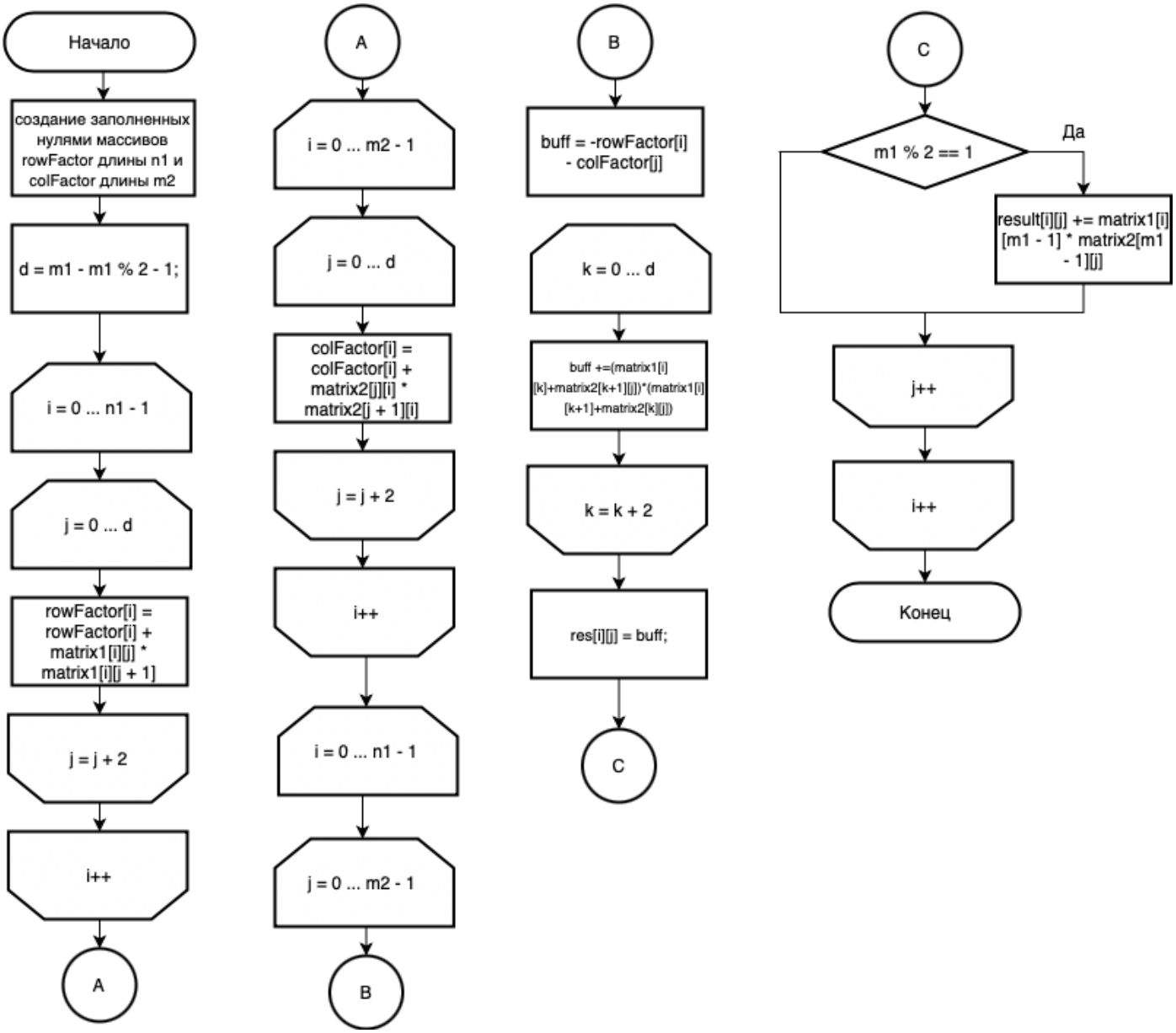

Рис. 2.3: Схема оптимизированного алгоритма Винограда

2.2 Модель вычислений

Для последующего вычисления трудоемкости введём модель вычислений.

1. Операции из списка (2.1) имеют трудоемкость 1.

$$+, -, *, /, \%, ==, != , <, >, <=, >=, [], ++, --\tag{2.1}$$

2. Трудоемкость оператора выбора if условие then A else B рассчитывается, как (2.2).

$$f_{if} = f_{\text{условия}} + \begin{cases} f_A, & \text{если условие выполняется,} \\ f_B, & \text{иначе.} \end{cases}\tag{2.2}$$

3. Трудоемкость цикла рассчитывается, как (2.3).

$$f_{for} = f_{\text{инициализации}} + f_{\text{сравнения}} + N(f_{\text{тела}} + f_{\text{инкремента}} + f_{\text{сравнения}})\tag{2.3}$$

   где N - количество итераций цикла.

4. Трудоемкость вызова функции равна 0.

2.3 Трудоёмкость алгоритмов

Примем, что размеры первой матрицы (r, s), второй - (s, c).

2.3.1 Стандартный алгоритм умножения матриц

Трудоёмкость стандартного алгоритма в выбранной модели вычислений в худшем и лучшем случаях рассчитывается следующим образом:

$$f_{base} = 2 + r(2 + 2 + c(2 + 2 + s(2 + 11))) = 13scr + 4cr + 4r + 2.\tag{2.4}$$

2.3.2 Алгоритм Винограда

Для алгоритма Винограда худшим случаем являются матрицы с нечётным s, а лучшим - с чётным, из-за того что отпадает необходимость в последнем цикле.

Трудоёмкость алгоритма Винограда является суммой трудоёмкостей следующих последовательно выполненных действий.

1. Заполнения вектора rowFactor:

$$f_{rowFactor} = 3 + r(2 + 2 + \frac{s}{2}(2 + 11)) = 6.5sr + 4r + 3.\tag{2.5}$$

2. Заполнения вектора colFactor:

$$f_{colFactor} = 2 + c(2 + 2 + \frac{s}{2}(2 + 11)) = 6.5sc + 4r + 2.\tag{2.6}$$

3. Основного цикла заполнения матрицы:

$$f_{cycle} = 2 + r(2 + 2 + c(2 + 2 + 7 + \frac{s}{2}(2 + 23))) = 12.5scr + 11cr + 4r + 2. \quad (2.7)$$

4. Цикла для дополнения умножения, если s нечётный:

$$f_{last} = \begin{cases} 2, & \text{s чётный,} \\ 2 + 2 + r(2 + 2 + c(2 + 13)) = 15cr + 4r + 4, & \text{иначе.} \end{cases} \quad (2.8)$$

Итак, для лучшего случая (s чётный):

$$f_{vin\_b} = 6.5sr+4r+3+6.5sc+4r+2+12.5scr+11cr+4r+2+2 = 12.5scr+6.5sr+6.5sc+11cr+12r+9. \quad (2.9)$$

Для худшего случая (s нечётный):

$$f_{vin\_w} = 6.5sr + 4r + 3 + 6.5sc + 4r + 2 + 12.5scr + 11cr + 4r + 2 + 15cr + 4r + 4 = \quad (2.10)$$

$$= 12.5scr + 6.5sr + 6.5sc + 26cr + 16r + 11. \quad (2.11)$$

2.3.3 Оптимизированный алгоритм Винограда

Трудоёмкость оптимизированного алгоритма Винограда является суммой трудоёмкостей следующих последовательно выполненных действий.

1. Заполнения вектора rowFactor:

$$f_{rowFactor} = 5 + r(3 + 2 + \frac{s}{2}(3 + 10)) = 6.5sr + 5r + 5. \quad (2.12)$$

2. Заполнения вектора colFactor:

$$f_{colFactor} = 2 + c(2 + 2 + \frac{s}{2}(2 + 11)) = 6.5s + 5r + 2. \quad (2.13)$$

3. Основного цикла заполнения матрицы:

$$f_{cycle} = 3+2+r(2+2+c(2+2+5+\frac{s}{2}(3+15)+f_{last}+3)) = 9scr+12cr+f_{last}cr+4r+5. \quad (2.14)$$

$$f_{last} = \begin{cases} 0, & \text{s чётный,} \\ 9, & \text{иначе.} \end{cases} \quad (2.15)$$

Итак, для лучшего случая (s чётный):

$$f_{vinOpt\_b} = 6.5sr+5r+5+6.5s+5r+2+9scr+12cr+4r+5 = 9scr+6.5sr+6.5sc+12cr+14r+12. \quad (2.16)$$

Для худшего случая (s нечётный):

$$f_{vinOpt\_w} = 6.5sr+5r+5+6.5s+5r+2+9scr+12cr+9cr+4r+5 = 9scr+6.5sr+6.5sc+21cr+14r+12. \quad (2.17)$$

2.4 Вывод

На основе теоретических данных, полученных из аналитического раздела, были построены схемы алгоритмов умножения матриц, оценены их трудоёмкости в лучшем и худшем случаях.

3 | Технологическая часть

В данном разделе приводится реализация алгоритмов, схемы которых были разработаны в конструкторской части. Кроме того, обосновывается выбор технологического стека и проводится тестирование реализованных алгоритмов.

3.1 Средства реализации

В качестве языка программирования был выбран C++ из-за его быстродействия, а среды разработки – CLion.

3.2 Реализация алгоритмов

В листингах 3.1, 3.2 и 3.3 приведены реализации алгоритмов умножения матриц.

Листинг 3.1: Функция обычного алгоритма умножения матриц

```
1  Matrix Matrix::convMul(Matrix &matr)
2  {
3      Matrix result(rows, matr.cols);
4      for (size_t i = 0; i < rows; i++) {
5          for (size_t k = 0; k < matr.cols; k++) {
6              result.matrix_ptr[i][k] = 0;
7              for (size_t j = 0; j < cols; j++)
8                  result.matrix_ptr[i][k] += matrix_ptr[i][j] * matr.
                       matrix_ptr[j][k];
9          }
10     }
11     return result;
12 }
```

Листинг 3.2: Функция алгоритма умножения матриц Винограда

```
Matrix Matrix::vinogradMul(Matrix &matr)
{
    Matrix res(rows, matr.cols);
    double *rowFactor = (double *) malloc (rows * sizeof(double));
    double *colFactor = (double *) malloc (matr.cols * sizeof(double ));

    int d = this->cols / 2;
    for (int i = 0; i < this->rows; i++) {
        rowFactor[i] = matrix_ptr[i][0] * matrix_ptr[i][1];
        for (int j = 1; j < d; j++)
        {
            rowFactor[i] = rowFactor[i] + matrix_ptr[i][2 * j] * matrix_ptr[
               i][2 * j + 1];

        }
    }

    for (int i = 0; i < matr.cols; i++) {
        colFactor[i] = matr.matrix_ptr[0][i] * matr.matrix_ptr[1][i];
        for (int j = 1; j < d; j++)
        {
            colFactor[i] = colFactor[i] + matr.matrix_ptr[2 * j][i] * matr.
               matrix_ptr[2 * j + 1][i];
        }
    }

    for (int i = 0; i < this->rows; i++) {
        for (int j = 0; j < matr.cols; j++) {
            res.matrix_ptr[i][j] = -rowFactor[i] - colFactor[j];
            for (int k = 0; k < d; k++)
                res.matrix_ptr[i][j] = res.matrix_ptr[i][j] + (matrix_ptr[i
                   ][2 * k] + matr.matrix_ptr[2 * k + 1][j]) * (matrix_ptr[i
                   ][2 * k + 1] + matr.matrix_ptr[2 * k][j]);
        }
    }

    if (this->cols % 2 == 1)
    {
        for (int i = 0; i < rows; i++)
            for (int j = 0; j < matr.cols; j++)
                res.matrix_ptr[i][j] = res.matrix_ptr[i][j] + matrix_ptr[i][
                   this->cols - 1] * matr.matrix_ptr[this->cols - 1][j];
    }
    free(rowFactor);
    free(colFactor);
    return res;
}
```

Листинг 3.3: Функция оптимизированного алгоритма умножения матриц Винограда

```
Matrix Matrix::optimizedMul(Matrix &matr)
{
    Matrix res(rows, matr.cols);
    double *rowFactor = (double *) malloc (rows * sizeof(double));
    double *colFactor = (double *) malloc (matr.cols * sizeof(double ));

    int d = cols - cols % 2;
    for (int i = 0; i < this->rows; ++i, ++i) {
        rowFactor[i] = 0;
        for (int j = 0; j < d; ++j, ++j)
            rowFactor[i] = rowFactor[i] + matrix_ptr[i][j] * matrix_ptr[i][j
                + 1];
    }

    for (int i = 0; i < matr.cols; i++) {
        colFactor[i] = 0;
        for (int j = 0; j < d; ++j, ++j)
        {
            colFactor[i] = colFactor[i] + matr.matrix_ptr[j][i] * matr.
               matrix_ptr[j + 1][i];
        }
    }

    bool flag = this->cols % 2 == 1;
    double buf;
    for (int i = 0; i < this->rows; ++i) {
        for (int j = 0; j < matr.cols; ++j) {
            buf = -rowFactor[i] - colFactor[j];
            for (int k = 0; k < d; ++k, ++k)
                buf = buf + (matrix_ptr[i][k] + matr.matrix_ptr[k + 1][j]) *
                                                                (matrix_ptr[i
                                                                   ][k + 1] +
                                                                   matr.
                                                                   matrix_ptr[
                                                                   k][j]);
            if (flag)
                buf = buf + matrix_ptr[i][this->cols - 1] * matr.matrix_ptr[
                   this->cols - 1][j];
            res.matrix_ptr[i][j] = buf;
        }
    }
    free(rowFactor);
    free(colFactor);
    return res;
}
```

3.3 Тестирование

В таблице 3.1 приведены модульные тесты для функций, реализующих стандартный алгоритм умножения матриц, алгоритм Винограда и оптимизированный алгоритм Винограда. Методолгия тестирования - чёрный ящик.

Таблица 3.1: Тестирование функций

Первая матрица	Вторая матрица	Ожидаемый результат
$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \end{pmatrix}$	$\begin{pmatrix} 2 & 3 & 7 \\ 5 & 1 & 10 \\ 6 & -1 & 4 \end{pmatrix}$	$\begin{pmatrix} 12 & 5 & 27 \\ 26 & 13 & 61 \end{pmatrix}$
$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$	$\begin{pmatrix} 2 & 3 & 7 \\ 5 & 1 & 10 \end{pmatrix}$	$\begin{pmatrix} 12 & 5 & 27 \\ 26 & 13 & 61 \end{pmatrix}$
$\begin{pmatrix} 2 \end{pmatrix}$	$\begin{pmatrix} 2 \end{pmatrix}$	$\begin{pmatrix} 4 \end{pmatrix}$
$\begin{pmatrix} 1 & -2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} -1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 0 & 4 & 6 \\ 4 & 12 & 18 \\ 4 & 12 & 18 \end{pmatrix}$

Все тесты пройдены успешно.

3.4 Вывод

В данном разделе были реализованы алгоритмы умножения матриц: обычный алгоритм, алгоритм Винограда и оптимизированный алгоритм Винограда. Кроме того, реализации были успешно протестированы.

4 | Исследовательская часть

В эданном разделе проводится сравнительный анализ реализованных алгоритмов по процессорному времени.

4.1 Технические характеристики

Все нижепреведенные замеры времени проведены на процессоре: Intel Core i5, 1,4 GHz, 4ядерный. Время работы алгоритмов было замерено с помощью time.h, функции clock, которая измеряет процессорное время [1].

4.2 Время выполнения реализаций алгоритмов

Для сравнительного анализа времени выполнения реализаций алгоритмов был проведен эксперимент. Для замеров были сформированы следующие пары матриц.

1. Внутренний размер чётный и варьируется в пределах от 1000 до 2100 с шагом 100, а внешний постоянен и равен 100.

2. Внутренний размер нечётный и варьируется в пределах от 1001 до 2101 с шагом 100, а внешний постоянен и равен 100.

В таблицах 4.1 и 4.2 представлены результаты замеров. Время измерялось 10 раз для каждой пары матриц, после усреднялось.

Таблица 4.1: Таблица времени выполнения (в мс) алгоритмов при чётных размерах

Размер матрицы	Стандартный	Виноград	Оптимизированный
1000	62.86	56.38	41.96
1100	63.60	48.91	47.35
1200	67.08	52.26	49.35
1300	76.38	60.68	58.20
1400	83.19	65.09	62.93
1500	92.65	71.02	70.90
1600	100.19	76.75	77.12
1700	111.02	94.34	82.80
1800	115.22	97.15	87.39
1900	119.39	101.51	92.23
2000	134.29	102.36	99.96

Таблица 4.2: Таблица времени выполнения (в мс) алгоритмов при нечётных размерах

Размер матрицы	Стандартный	Виноград	Оптимизированный
1001	56.73	60.47	42.62
1101	61.71	48.64	46.49
1201	68.57	53.81	51.87
1301	76.09	59.92	57.12
1401	84.76	64.74	63.88
1501	90.83	70.58	68.64
1601	98.00	75.73	76.02
1701	105.49	90.68	81.28
1801	113.42	96.89	87.83
1901	121.53	103.43	93.42
2001	134.73	103.37	100.57

Время на графиках 4.1 и 4.2 представлено в милисекундах.

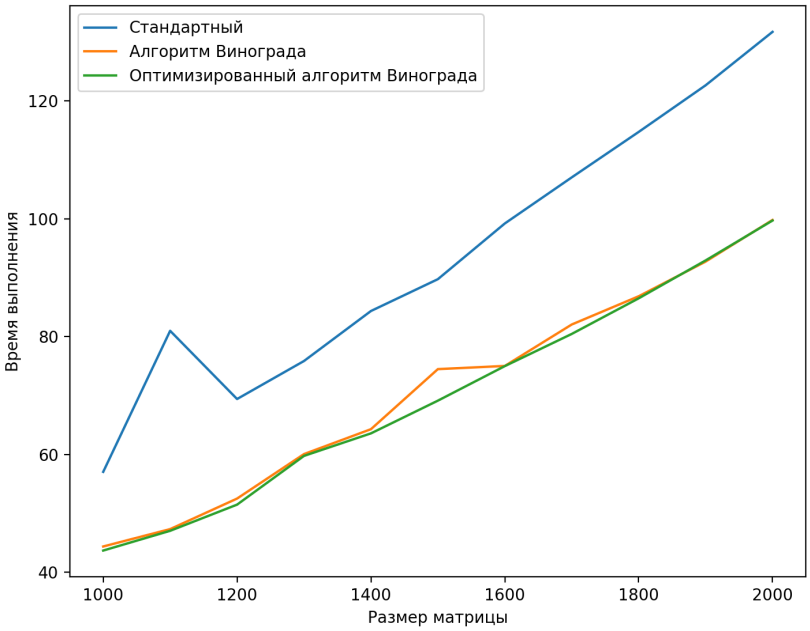

Рис. 4.1: Зависимость времени выполнения от размера матриц (чётный)

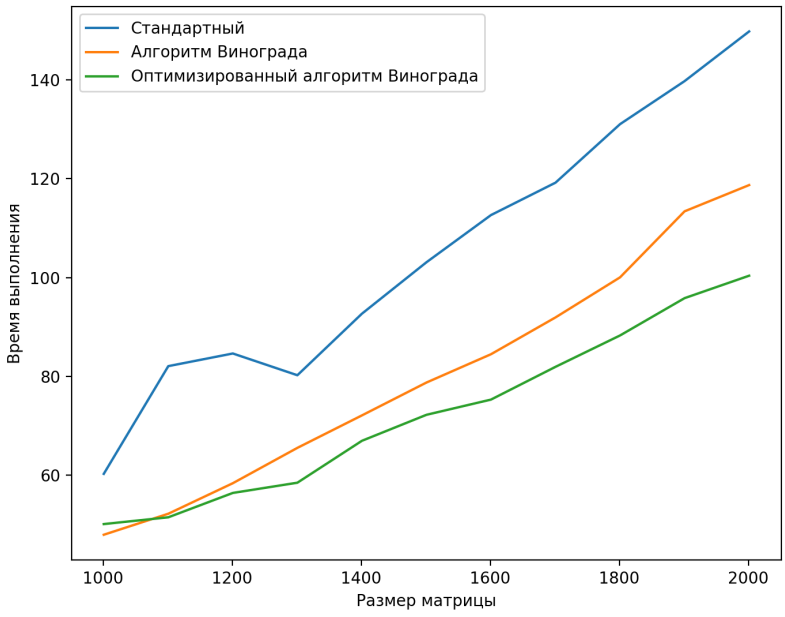

Рис. 4.2: Зависимость времени выполнения от размера матриц (чётный)

4.3 Вывод

На основе замеров процессорного времени было получено, что алгоритм Винограда в среднем в 1.2-1.4 раза быстрее, чем обычный алгоритм умножения матриц и немного хуже оптимизированного.

Заключение

В рамках данной лабораторной работы были решены следующие задачи.

1. Были изучены два алгоритма умножения матриц: обычный и алгоритм Винограда.
2. Разработан оптимизированный алгоритм Винограда.
3. Реализованы три алгоритма умножения матриц: обычный, алгоритм Винограда и оптимизированный алгоритм Винограда.
4. Был проведён сравнительный анализ трудоёмкости алгоритмов на основе расчетов в выбранной модели вычислений.
5. Был проведён равнительный анализ алгоритмов на основе экспериментальных данных.

На основании анализа трудоёмкости алгоритмов в выбранной модели вычислений было показано, что улучшенный алгоритм Винограда имеет меньшую сложность, нежели простой алгоритм умножения матриц. На основании замеров времени исполнения алгоритмов, был сделан вывод о том, что алгоритм Винограда в среднем в 1.2-1.4 раза быстрее, чем обычный алгоритм умножения матриц и немного хуже оптимизированного.

Поставленная цель была достигнута.

Литература

1. Стандартная функция clock(), измеряющая процессорное время [Электронный ресурс]. Режим доступа: https://en.cppreference.com/w/cpp/chrono/c/clock. Дата обращения: 03.10.2021