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»

Отчет по лабораторной работе №3 по дисциплине "Анализ алгоритмов"

Тема Умножение матриц

Студент Костев Д.И.

Группа ИУ7-51Б

Оценка (баллы) _____

Преподаватели Волкова Л.Л.

Москва — 2021 г.

Оглавление

Введение

Алгоритм Копперсмита — Винограда — алгоритм умножения квадратных матриц, предложенный в 1987 году Д. Копперсмитом и Ш. Виноградом. В исходной версии асимптотическая сложность алгоритма составляла $O(n^{2,3755})$, где n — размер стороны матрицы.

Алгоритм Копперсмита — Винограда, с учётом серии улучшений и доработок в последующие годы, обладает лучшей асимптотикой среди известных алгоритмов умножения матриц.

На практике алгоритм Копперсмита — Винограда не используется, так как он имеет очень большую константу пропорциональности и начинает выигрывать в быстродействии у других известных алгоритмов только для матриц, размер которых превышает память современных компьютеров. Поэтому пользуются алгоритмом Штрассена по причинам простоты реализации и меньшей константе в оценке трудоемкости.

Ниже представлены задачи лабораторной работы.

1. Изучение и реализация трёх алгоритмов умножения матриц: обычный, Копперсмита-Винограда, оптимизированный Копперсмита-Винограда.

2. Сравнительный анализ трудоёмкости алгоритмов на основе теоретических расчетов и выбранной модели вычислений.

3. Сравнительный анализ алгоритмов на основе экспериментальных данных.

1 Аналитическая часть

1.1 Стандартный алгоритм

Пусть даны две прямоугольные матрицы

$$A_{lm} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{l1} & a_{l2} & \dots & a_{lm} \end{pmatrix}, \quad B_{mn} = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{pmatrix}, \tag{1.1}$$

тогда матрица C

$$C_{ln} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{l1} & c_{l2} & \dots & c_{ln} \end{pmatrix}, \tag{1.2}$$

где

$$c_{ij} = \sum_{r=1}^{m} a_{ir} b_{rj} \quad (i = \overline{1,l}; j = \overline{1,n}) \tag{1.3}$$

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

1.2 Алгоритм Копперсмита – 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.

1.3 Оптимизированный алгоритм Копперсмита – Винограда

Оптимизированный алгоритм Винограда представляет собой обычный алгоритм Винограда, за исключением следующих оптимизаций:

- вычисление происходит заранее;
- используется битовый сдвиг, вместо деления на 2;
- последний цикл для нечётных элементов включён в основной цикл, используя дополнительные операции в случае нечётности.

1.4 Вывод из аналитической части

В данном разделе были рассмотрены алгоритмы классического умножения матриц и алгоритм Винограда, основное отличие которого от классического алгоритма — наличие предварительной обработки, а также количество операций умножения.

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.

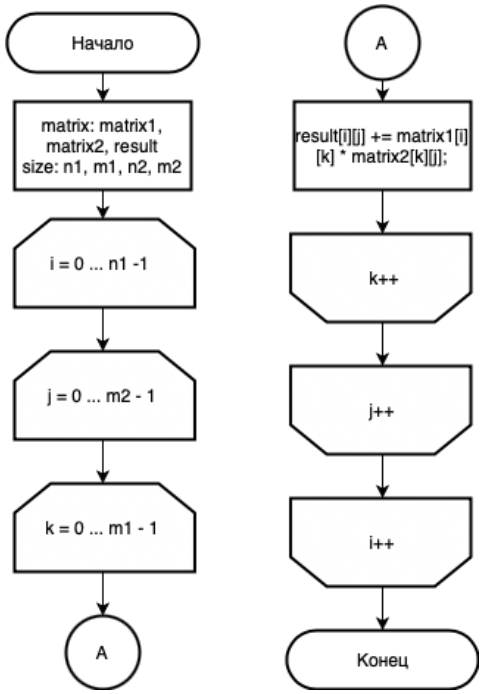

Рис. 2.1: Схема стандартного алгоритма умножения матриц

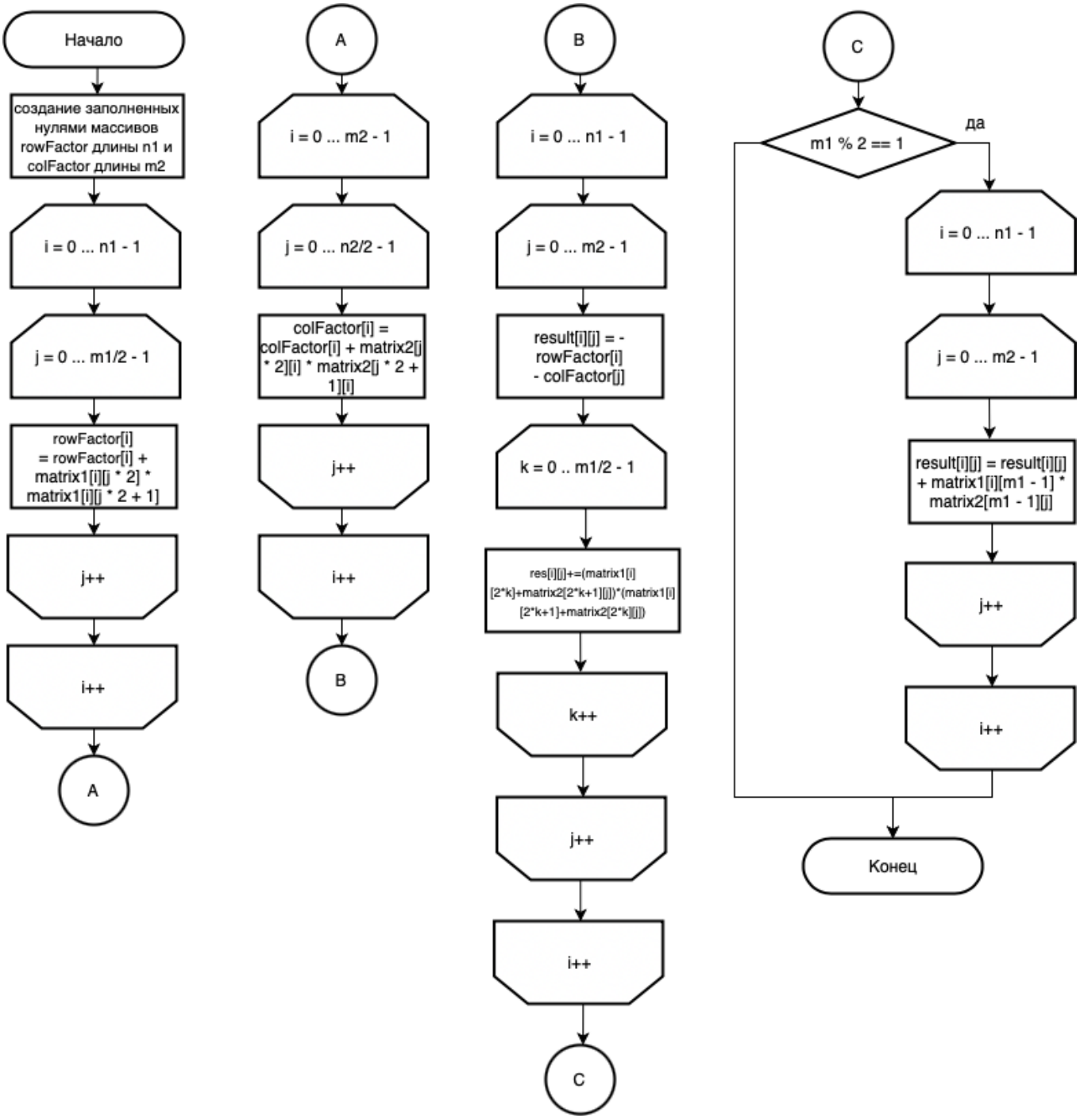

Рис. 2.2: Схема алгоритма Винограда

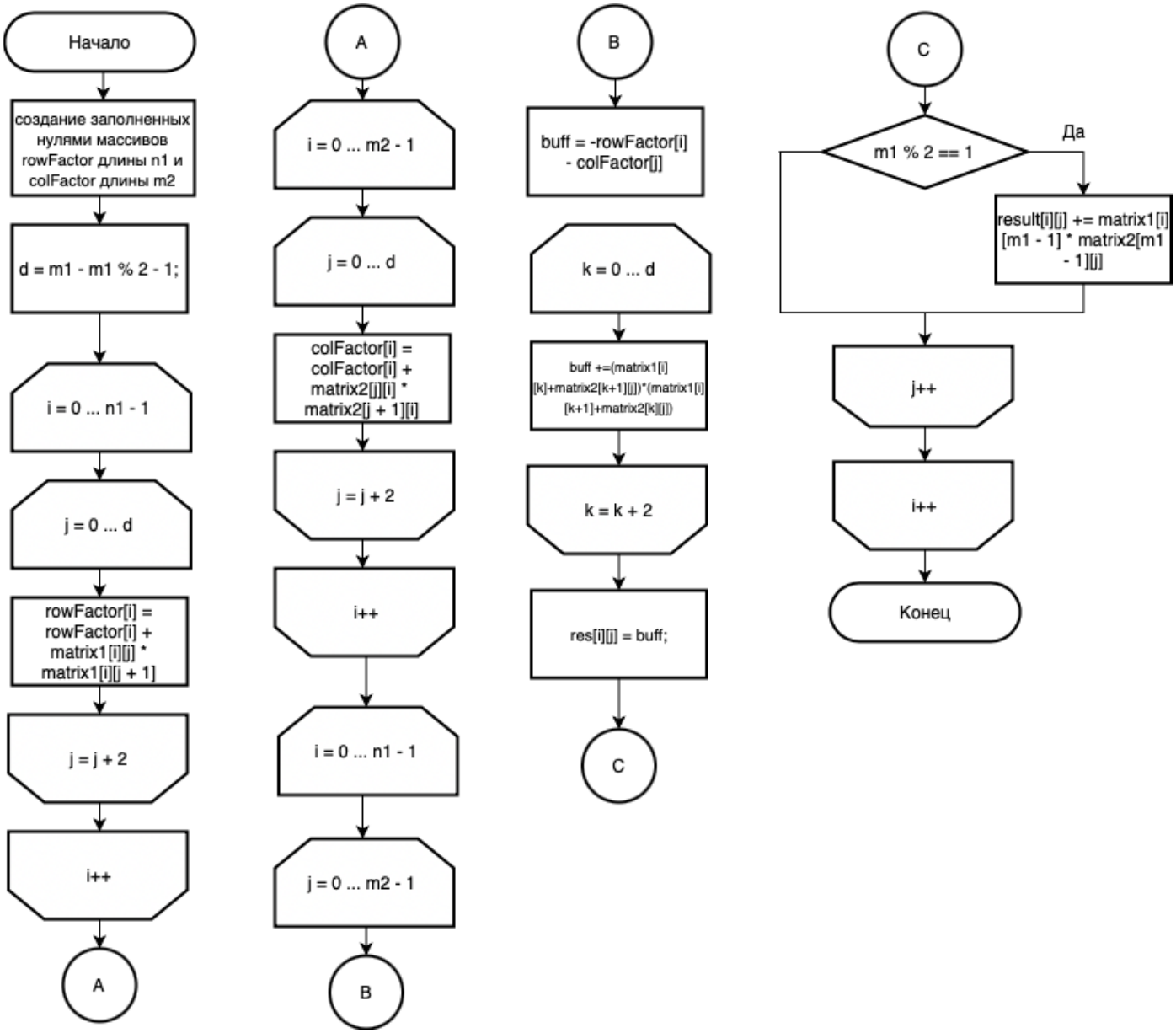

Рис. 2.3: Схема оптимизированного алгоритма Винограда

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

3.1 Требование к ПО

К программе предъявляется ряд требований.

1. На вход ПО получает размеры 2 матриц, а также их элементы.
2. На выходе — ПО печатает матрицу, которая является результатом умножения входных матриц.

3.2 Средства реализации

Для реализации ПО я выбрал язык программирования C++.

3.3 Реализация алгоритмов

В листингах 3.1 - 3.4 приведена реализация алгоритмов перемножения матриц.

Листинг 3.1: Функция умножения матриц обычным способом

```
Matrix Matrix::convMul(Matrix &matr, clock_t&time)
{
    Matrix result(rows, matr.cols);
    clock_t sTime = clock();
    for (size_t i = 0; i < rows; i++) {
        for (size_t k = 0; k < matr.cols; k++) {
            result.matrix_ptr[i][k] = 0;
            for (size_t j = 0; j < cols; j++)
                result.matrix_ptr[i][k] = result.matrix_ptr[i][k] +
                    matrix_ptr[i][j] * matr.matrix_ptr[j][k];
        }
    }
    clock_t eTime = clock();
    time = eTime - sTime;
    return result;
}
```

Листинг 3.2: Функция умножения матриц по Винограду

```
Matrix Matrix::vinogradMul(Matrix &matr, clock_t&time)
```

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

    clock_t sTime = clock();
    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];
41      }
42      clock_t eTime = clock();
43      time = eTime - sTime;
44      free(rowFactor);
45      free(colFactor);
46      return res;
47  }
```

Листинг 3.3: Функция умножения матриц по Винограду с оптимизацией

```
1   Matrix Matrix::optimizedMul(Matrix &matr, clock_t&time)
2   {
3       Matrix res(rows, matr.cols);
4       double *rowFactor = (double *) malloc (rows * sizeof(double));
5       double *colFactor = (double *) malloc (matr.cols * sizeof(double ));
6
7       clock_t sTime = clock();
8       int d = cols - cols % 2;
9       for (int i = 0; i < this->rows; ++i, ++i) {
10          rowFactor[i] = 0;
11          for (int j = 0; j < d; ++j, ++j)
12              rowFactor[i] = rowFactor[i] + matrix_ptr[i][j] * matrix_ptr[i][j
                    + 1];
13      }
14
15      for (int i = 0; i < matr.cols; i++) {
16          colFactor[i] = 0;
17          for (int j = 0; j < d; ++j, ++j)
18          {
19              colFactor[i] = colFactor[i] + matr.matrix_ptr[j][i] * matr.
                    matrix_ptr[j + 1][i];
20          }
21      }
22
23      bool flag = this->cols % 2 == 1;
24      double buf;
25      for (int i = 0; i < this->rows; ++i) {
26          for (int j = 0; j < matr.cols; ++j) {
27              buf = -rowFactor[i] - colFactor[j];
28              for (int k = 0; k < d; ++k, ++k)
29                  buf = buf + (matrix_ptr[i][k] + matr.matrix_ptr[k + 1][j]) *
30                                                                  (matrix_ptr[i
                                                                        ][k + 1] +
                                                                        matr.
                                                                        matrix_ptr[
                                                                        k][j]);
31              if (flag)
32                  buf = buf + matrix_ptr[i][this->cols - 1] * matr.matrix_ptr[
                        this->cols - 1][j];
```

```
33                 res.matrix_ptr[i][j] = buf;
34             }
35         }
36         clock_t eTime = clock();
37         time = eTime - sTime;
38         free(rowFactor);
39         free(colFactor);
40         return res;
41 }
```

3.4 Тестовые данные

В таблице 3.1 приведены тесты для функций, реализующих стандартный алгоритм умножения матриц, алгоритм Винограда и оптимизированный алгоритм Винограда. Все тесты пройдены успешно.

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

Таблица 3.1: Тестирование функций

3.5 Вывод

В данном разделе были разработаны исходные коды четырёх алгоритмов перемножения матриц: обычный алгоритм, алгоритм с транспонированием, алгоритм Копперсмита — Винограда, оптимизированный алгоритм Копперсмита — Винограда.

4 | Исследовательская часть

4.1 Технические характеристики

Ниже приведеные технические характеристики устройства, на котором было проведенно тестирование ПО.

- Операционная система: Windows 11 64-bit.
- Оперативная память: 8 ГБ.
- Процессор: Intel(R) Core(TM) i5-8250 CPU @ 1.60GHz.

4.2 Время выполнения алгоритмов

Время выполнения алгоритм замерялось с помощью применения технологии профайлинга. Данный инстрмуент даёт детальное описание количества вызовов и количества времени CPU, занятого каждой функцией.

В таблицах 4.1 и 4.2 представлены замеры времени работы для каждого из алгоритмов на чётных размерах матриц. Здесь и далее: С — стандартный алгоритм, КВ — алгоритм Копперсмита – Винограда, ОКВ – оптимизированный алгоритм Копперсмита – Винограда.

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

Размер матрицы	С	КВ	ОКВ
1000	67.20	52.30	43.50
1100	76.40	56.10	47.80
1200	87.30	68.50	56.20
1300	94.90	70.80	58.20
1400	102.60	76.20	63.50
1500	111.70	84.70	69.20
1600	116.40	86.60	72.70
1700	128.80	93.80	82.70
1800	134.30	99.20	83.00
1900	145.60	108.00	89.20
2000	154.40	120.00	96.70

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

Размер матрицы	С	КВ	ОКВ
1001	70.30	53.70	43.20
1101	90.30	72.30	56.30
1201	119.00	74.70	58.80
1301	95.80	70.70	62.10
1401	106.70	76.80	64.10
1501	111.90	86.00	70.10
1601	118.50	88.50	77.00
1701	130.20	94.70	78.40
1801	133.40	99.50	82.70
1901	145.30	111.30	89.70
2001	159.80	141.30	100.50

4.3 Вывод из исследовательской части

Реализация умножения матриц с помощью алгоритма Копперсмита – Винограда в среднем выполняется в 1.3 раза быстрее, чем умножение обычным способом. Учлучшенный алгоритм же работает в 1.5 раз быстрее обычного перемножения и в 1.2 быстрее классического Копперсмита – Винограда.

Заключение

В рамках данной лабораторной работы были выполнены нижеуказанные задачи.

1. Были изучены и реализованы 3 алгоритма перемножения матриц: обычный, Копперсмита – Винограда, оптимизированный Копперсмита – Винограда.
2. Был произведён анализ трудоёмкости алгоритмов на основе теоретических расчётов и выбранной модели вычислений.
3. Был сделан сравнительный 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 перемножения матриц. На основании замеров времени исполнения алгоритмов, был сделан вывод, что алгоритм Копперсмита – Винограда в среднем в 3.5 раза быстрее чем обычный алгоритм умножения матриц. Кроме этого, я решил добавить в сравнение алгоритм с предварительным транспонированием матрицы. Оказалось, что такая реализация быстрее алгоритма Копперсмита – Винограда в 1.5 раза и обгоняет классический алгоритм умножения в 3.6 раза.

Литература

[1] Кормен Т. Алгоритмы: построение и анализ [Текст] / Корм ен Т. - Вильямс, 2014. - 198 с. - 219 с.

[2] GetProcessTimes function. URL: https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-getprocesstimes, 01.10.2021

[3] Алгоритм Копперсмита — Винограда. URL: https://ru.wikipedia.org/wiki/Алгоритм-Копперсмита—Винограда