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»

НАПРАВЛЕНИЕ ПОДГОТОВКИ «09.03.04 Программная инженерия»

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ №2

Название: Умножение матриц. Формула Копперсмита – Винограда

Дисциплина: Анализ алгоритмов

Студент	ИУ7-56Б		Т. А. Казаева
	Группа	Подпись, дата	И. О. Фамилия
Преподаватель			Л. Л. Волкова
		Подпись, дата	И. О. Фамилия

Москва, 2021 г.

Содержание

Введение

Разработка и совершенствование матричных алгоритмов является важнейшей алгоритмической задачей. Непосредственное применение классического матричного умножения требует времени порядка $O(n^3)$. Однако существуют алгоритмы умножения матриц, работающие быстрее очевидного. В линейной алгебре алгоритм Копперсмита – Винограда[1], названный в честь Д. Копперсмита и Ш. Винограда , был асимптотически самый быстрый из известных алгоритмов умножения матриц с 1990 по 2010 год. В данной работе внимание акцентируется на алгоритме Копперсмита – Винограда и его улучшениях.

Алгоритм не используется на практике, потому что он дает преимущество только для матриц настолько больших размеров, что они не могут быть обработаны современным вычислительным оборудованием. Если матрица не велика, эти алгоритмы не приводят к большой разнице во времени вычислений.

Цель лабораторной работы – теоретическое изучение алгоритма Копперсмита – Винограда, разработка его улучшений и сравнение с классическим алгоритмом на основе полученных в ходе лабораторной работы экспериментальных данных. Для этого необходимо проанализировать изучаемый алгоритм, выделить основные особенности и недостатки для дальнейшей оптимизации.

1. Аналитический раздел

1.1 Применение математического подхода

Даны матрицы $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p}$, произведение матриц $B \in \mathbb{R}^{m \times p}$ – такая матрица, $= A \times B$, каждый элемент которой вычисляется согласно формуле 1.1:

$$c_{i,j} = \sum_{n}^{k=1} a_{i,k} \cdot b_{k,j}, \text{ где } i = \overline{1,m}, j = \overline{1,p} \tag{1.1}$$

Стандартный алгоритм умножения матриц реализует формулу(1.1).

Операция умножения двух матриц выполнима только в том случае, если число столбцов в первом сомножителе равно числу строк во втором. В частности, умножение всегда выполнимо, если оба сомножителя — квадратные матрицы одного и того же порядка.

1.2 Алгоритм Копперсмита – Винограда

Для начала стоит обратить внимание на альтернативный подсчет выражения $a_1 \cdot b_1 + a_2 \cdot b_2$ осуществляется согласно 1.2:

$$\begin{aligned} &\lambda_1 = a_1 \cdot a_1 \\ &\lambda_2 = b_1 \cdot b_1 \\ &\lambda_3 = (a_1 + b_2) \cdot (a_2 + b_1) \\ &\text{результат: } \lambda_3 - \lambda_1 - \lambda_2 \end{aligned} \tag{1.2}$$

Классическое умножение матриц, по своей сути, является нахождением некоторого числа скалярных произведений каждого столбца первого множителя с каждой строкой второго. Процедура может быть усовершенствована: если один вектор V встречается множество раз, то операция нахождения векторного произведения для него может быть выполнена единожды. Идея препроцессирования в случае перемножения квадратных матриц $n \times n$ приводит к определению алгоритма Копперсмита – Винограда. Для векто-

ра $x = (x_1 \cdots x_n)$ можно записать(1.3):

$$W(x) = x_1 \cdot x_2 + x_3 \cdot x_4 + \cdots + x_{n-1} \cdot x_n \tag{1.3}$$

Тогда умножения матрицы выполняется согласно следующему алгоритму:

1. Для каждой строки R_i матрицы M вычислить $W(R_i)$ и для каждого столбца C_i матрицы M вычислить $W(C_i)$;

2. Для каждой пары (i, j), где r соответствует R_i и C соответствует C_i, вычислить 1.4:

$$\begin{gathered} r \cdot c = (r_1 + c_2) + (r_2 + c_1) \cdot (r_3 + c_4) \cdot \\ (r_4 + c_3) + \cdots + (r_{n-1} + c_n) + (r_n + c_{n-1}) - W(r) - W(c). \end{gathered} \tag{1.4}$$

Если оценивать подход Копперсмита – Винограда опуская идею препроцессирования, то можно заметить, что арифметических операций в ней больше, чем в формуле классического скалярного произведения Однако, сохранение результатов $W(C_i)$ и $W(R_i)$ позволяет выполнять меньше операций, чем при нахождении матричного произведения математически. Разница при таком подходе, очевидно, будет заметна на матрицах настолько больших размеров, что они не могут быть обработаны ЭВМ.

1.3 Вывод

Была выявлена основная особенность подхода Копперсмита – Винограда – идея предварительной обработки. Разница во времени выполнения при такой оптимизации будет экспериментально вычислена в исследовательском разделе.

2. Конструкторский раздел

2.1 Трудоемкость алгоритмов

Для получения функции трудоемкости алгоритма необходимо ввести модель оценки трудоемкости. Трудоемкость "элементарных"операций оценивается следующим образом:

1. Трудоемкость 1 имеют операции:

$$+, -, =, <, >, <=, >=, ==, + =, - =,$$
$$++, --, [], \&\&, ||, >>, <<$$

2. Трудоемкость 2 имеют операции:

$$*, /, \backslash, \%$$

3. Трудоемкость конструкции ветвления определяется согласно формуле 2.1

$$f_{if} = f_{condition} + \begin{cases} min\left(f_{true}, f_{false}\right) & \text{в лучшем случае,} \\ max\left(f_{true}, f_{false}\right) & \text{в худшем случае.} \end{cases} \quad (2.1)$$

4. Трудоемкость цикла расчитывается по формуле 2.2

$$f_{loop} = f_{init} + f_{cmp} + N\left(f_{body} + f_{inc} + f_{cmp}\right), \quad (2.2)$$

где
f_{init} — трудоемкость инициализации,
f_{body} — трудоемкость тела цикла,
f_{iter} — трудоемкость инкремента,
f_{cmp} — трудоемкость сравнения,
N — количество повторов.

5. Трудоемкость вызова функции равна 0.

2.2 Оптимизация алгоритма Копперсмита – Винограда

Алгоритм Копперсмита – Винограда можно оптимизировать следующим образом:

- Счетчики цикла можно объявить единожды и обнулять их по требованию. В этом случае стоимость инициализации при расчете трудоемкости цикла сократится;
- Увеличение числа на определенное число можно заменить на операцию $+=$, поскольку она имеет меньший вес чем в сумме сложение и присвоение;
- Для алгоритма худшим случаем являются матрицы с нечётным общим размером, а лучшим - с четным. Соответственно, алгоритм можно ускорить в соответствии с четностью размера.

2.3 Трудоемкость алгоритмов

Классический алгоритм

Пусть на вход алгоритму поступают матрицы M_{left} и M_{right} с размерностью $n \times m$ и $m \times q$. Тогда трудоемкость классического алгоритма определяется по формуле 2.3

$$f_{alg} = f_{loop_i} = 2 + n\left(2 + f_{loop_j}\right) = 2 + n\left(2 + 2 + q\left(2 + f_{loop_k}\right)\right) = \\ = 2 + n\left(2 + 2 + q\left(2 + 2 + 12 \cdot m\right)\right) \approx 10mnq = 10MNK \quad (2.3)$$

Алгоритм Копперсмита – Винограда

Пусть на вход алгоритму поступают матрицы M_{left} и M_{right} с размерностью $n \times m$ и $m \times q$. Тогда трудоемкость алгоритма Копперсмита – Вино-

града расчитывается как 2.4:

$$f_{CW} = f_{init} + f_{precomp} + f_{fill} + f_{even}, \tag{2.4}$$

где:

- f_{init} – инициализация массивов для препроцессирования, стоимость которой равна 2.5

$$f_{init} = n + q \tag{2.5}$$

- $f_{precomp}$ – трудоемкость заполнения массивов , которая вычисляется по формуле 2.6:

$$f_{precomp} = f_{rows} + f_{cols}, \tag{2.6}$$

  где:

  * f_{rows} – трудоемкость заполнения массива строк, состоящая из двух циклов, трудоемкость которых, согласно модели вычисления в разделе 2.2 равна 2.7:

$$f_{rows} = 2 + n\,(8m - 6) \tag{2.7}$$

  * f_{cols} – трудоемкость заполнения массива столбцов, состоящая из двух циклов, трудоемкость которых, согласно модели вычисления в разделе 2.2 равна 2.8

$$f_{cols} = 2 + q\,(8m - 6) \tag{2.8}$$

- f_{fill} – цикл заполнения матрицы, трудоемкость которого, согласно модели вычисления в разделе 2.2 равна 2.9:

$$f_{fill} = 2 + n\left(2 + q\left(2 + 21\frac{m-1}{2}\right)\right) \tag{2.9}$$

- f_{even} – цикл для дополнения умножения в случае нечетной размерно-

сти, который, согласно модели вычисления в разделе 2.2 равна 2.10:

$$f_{fill} = \begin{cases} 2 & \text{л. с,} \\ 2 + n\,(2 + 14q) & \text{х. с.} \end{cases} \tag{2.10}$$

Значит, трудоемкость алгоритма Копперсмита – Винограда в лучшем случае равна 2.11:

$$\begin{aligned} f_{CW} = n + q + 2 + n\,(8m - 6) + 2 + q\,(8m - 6) + \\ + 2 + n\left(2 + q\left(2 + 21\frac{m-1}{2}\right)\right) + \\ + 2 \approx \frac{21}{2}mnq = 10.5MNK \end{aligned} \tag{2.11}$$

В худшем случае соответственно 2.12:

$$\begin{aligned} f_{CW} = n + q + 2 + n\,(8m - 6) + 2 + q\,(8m - 6) + \\ + 2 + n\left(2 + q\left(2 + 21\frac{m-1}{2}\right)\right) + \\ + 2 + n\,(2 + 14q) \approx \frac{21}{2}mnq = 10.5MNK \end{aligned} \tag{2.12}$$

Оптимизация алгоритма Копперсмита – Винограда

Пусть на вход алгоритму поступают матрицы M_{left} и M_{right} с размерностью $n \times m$ и $m \times q$. Тогда трудоемкость оптимизированного алгоритма Копперсмита – Винограда расчитывается как 2.13:

$$f_{CW} = f_{init} + f_{precomp} + f_{fill} \tag{2.13}$$

Здесь f_{init} расчитывается по формуле 2.5. Однако, отличаться будет стоимость $f_{precomp}$ благодаря замене присваивания и сложения на оператор $+=$ и предварительной инициализации индексов. Соответственно, $f_{precomp}$

удовлетворяет формуле2.14:

$$f_{precomp} = f_{rows} + f_{cols} = 2 + n\left(9\frac{m-1}{2}\right) + q\left(9\frac{m-1}{2}\right) \tag{2.14}$$

Трудоемкость цикла заполнения матрицы расчитывается следующим образом (2.15):

$$f_{fill} = \begin{cases} 1 + n\left(2 + q\left(2 + 17\frac{m-1}{2}\right)\right) & \text{л. с,} \\ 1 + n\left(2 + q\left(9 + 17\frac{m-1}{2}\right)\right) & \text{х. с.} \end{cases} \tag{2.15}$$

Значит, трудоемкость алгоритма Копперсмита – Винограда в лучшем случае равна 2.16:

$$f_{CW} = n + q + 2 + n\left(9\frac{m-1}{2}\right) + q\left(9\frac{m-1}{2}\right) + 1 + n\left(2 + q\left(2 + 17\frac{m-1}{2}\right)\right) \approx \frac{17}{2}mnq = 9.5MNK \tag{2.16}$$

В худшем случае соответственно 2.17:

$$f_{CW} = n + q + 2 + n\left(9\frac{m-1}{2}\right) + q\left(9\frac{m-1}{2}\right) + 1 + n\left(2 + q\left(9 + 17\frac{m-1}{2}\right)\right) \approx \frac{17}{2}mnq = 9.5MNK \tag{2.17}$$

2.4 Схемы алгоритмов

На рисунке 2.1 приведена схема классического алгоритма умножения матриц. На рисунках 2.2, 2.3, 2.4 приведена схема алгоритма Копперсмита – Винограда, на рисунках 2.5, 2.6 приведена схема алгоритмов препроцессирования. Рисунки 2.7, 2.8 демонстрируют схему оптимизированного алгоритма Копперсмита – Винограда.

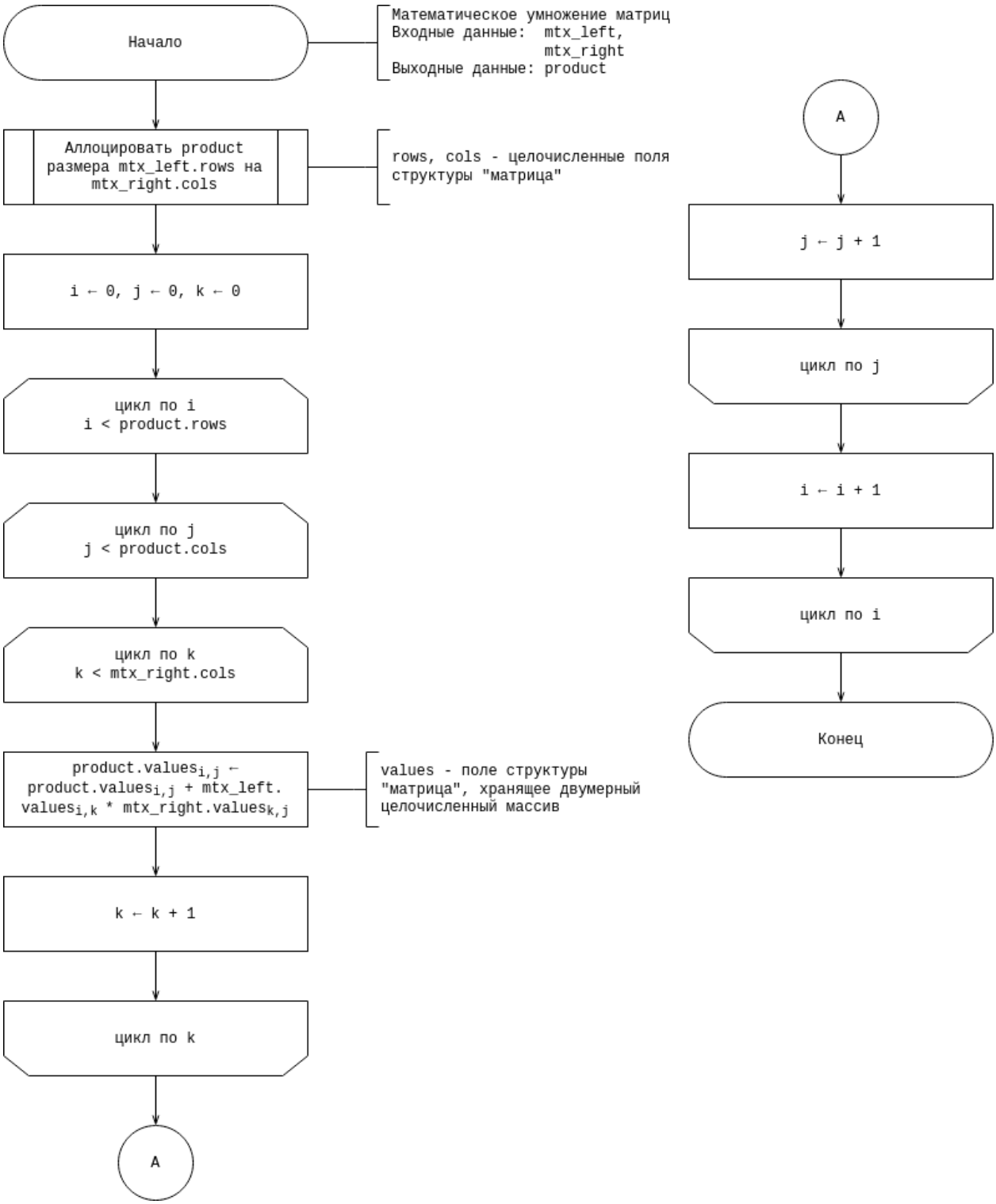

Рисунок 2.1 – Схема классического алгоритма умножения матриц

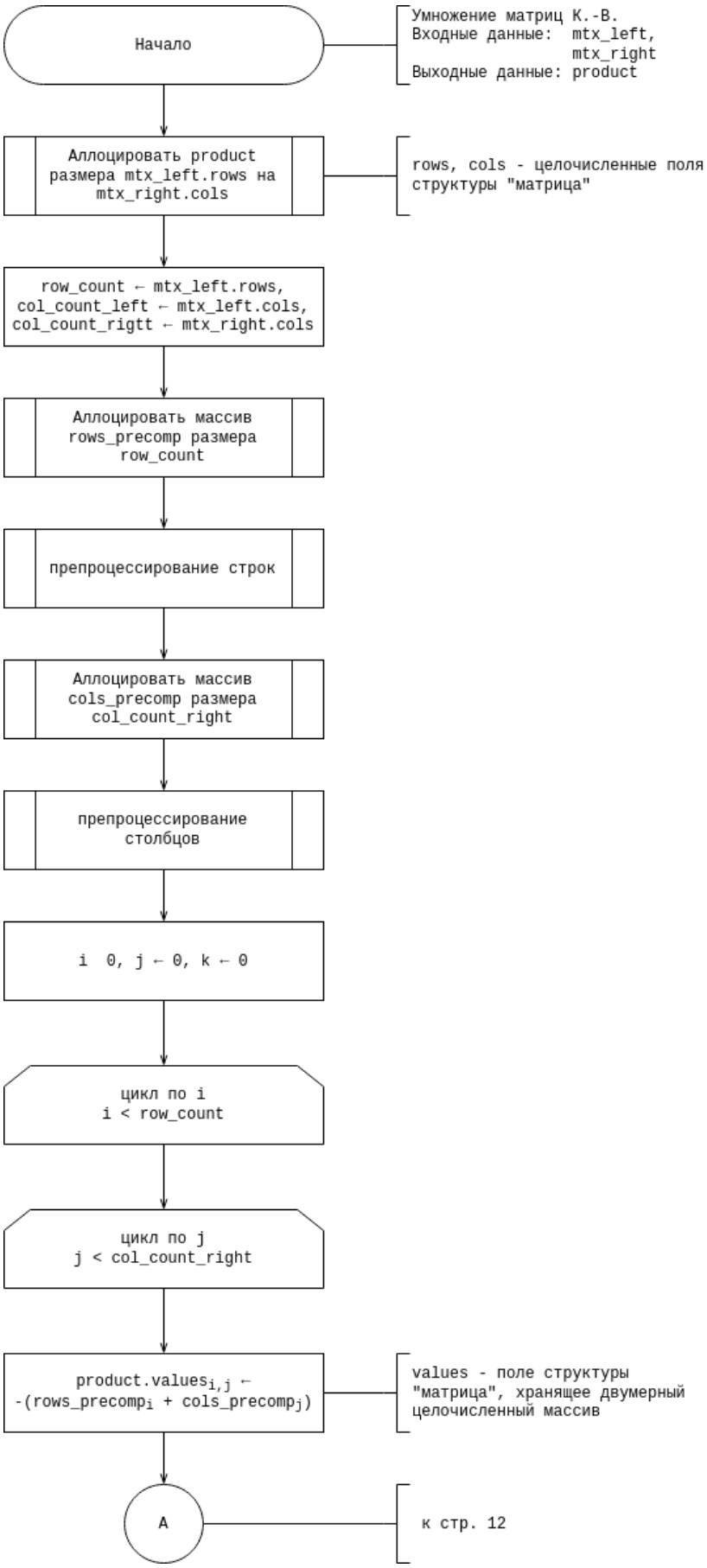

Рисунок 2.2 – Схема алгоритма умножения матриц Копперсмита – Винограда (начало)

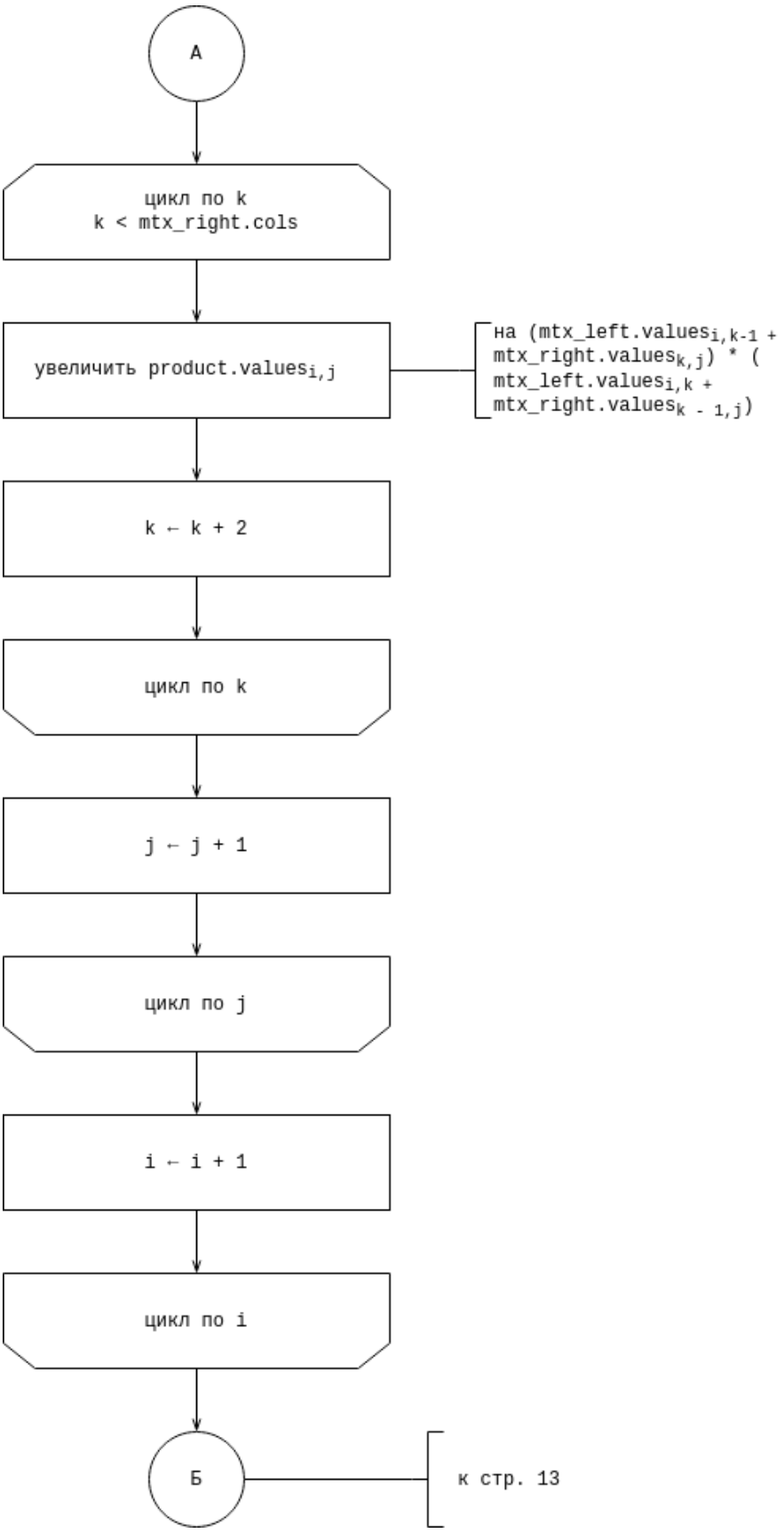

Рисунок 2.3 – Схема алгоритма умножения матриц Копперсмита – Винограда (продолжение)

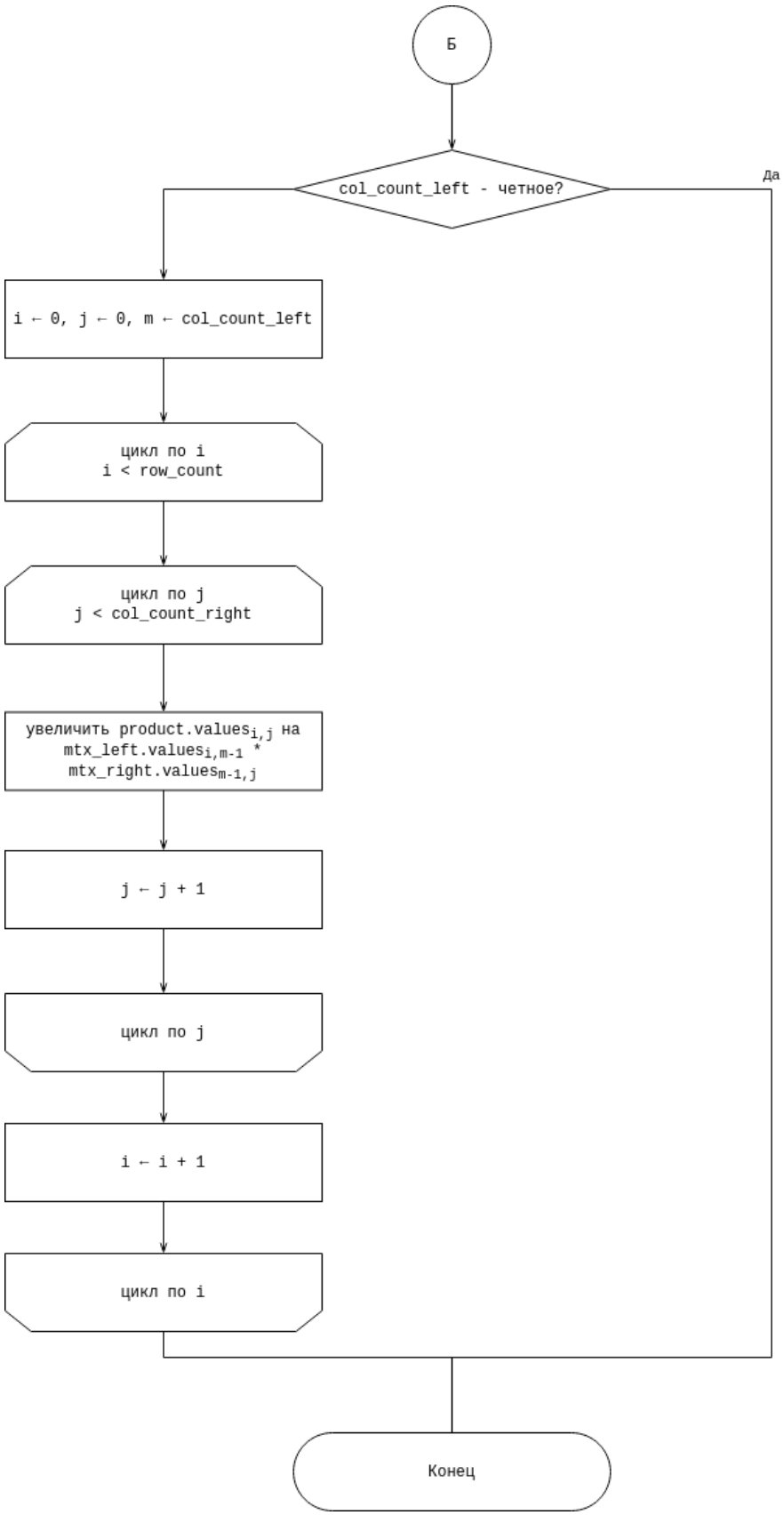

Рисунок 2.4 – Схема алгоритма умножения матриц Копперсмита – Винограда (конец)

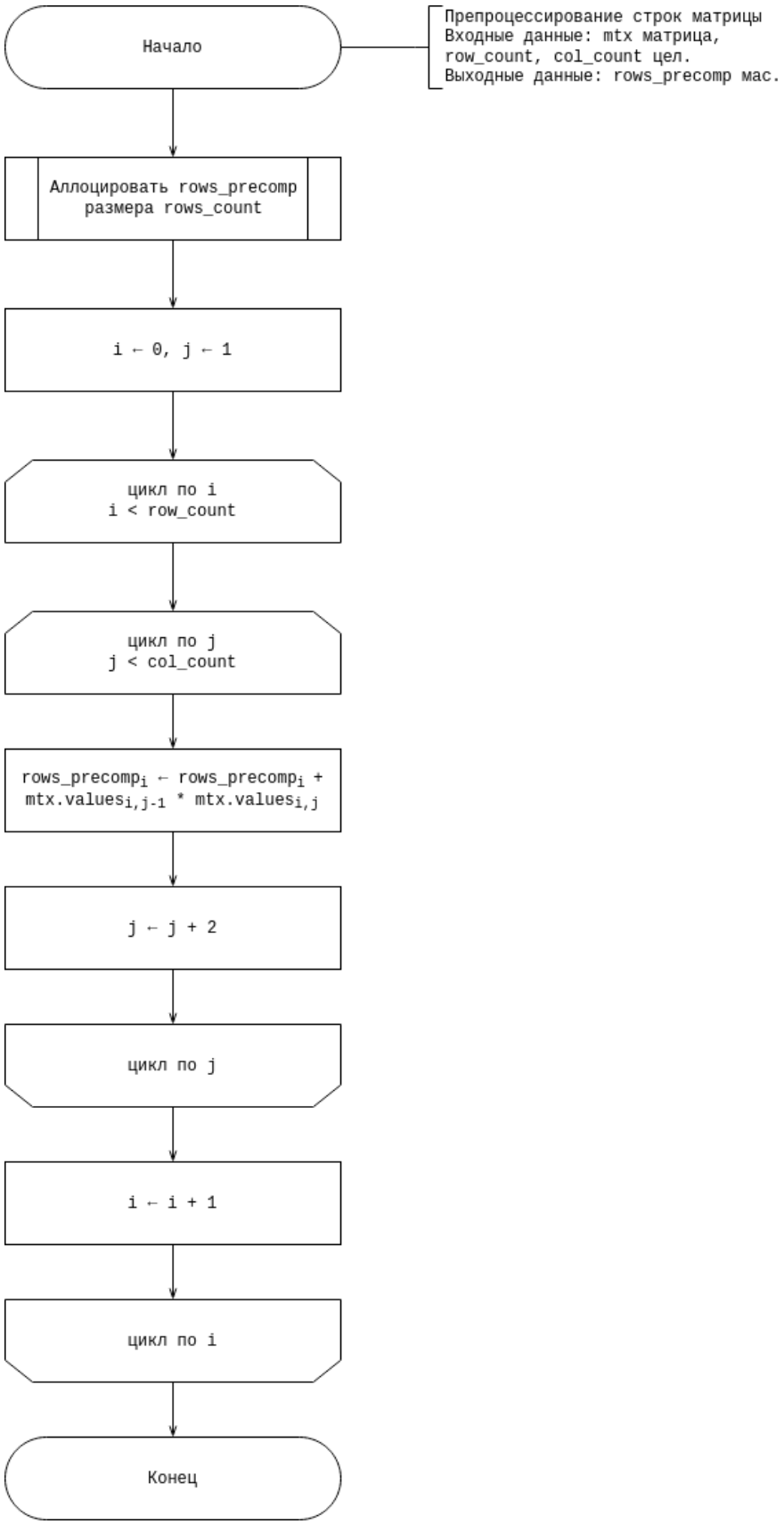

Рисунок 2.5 – Алгоритм препроцессирования сторк

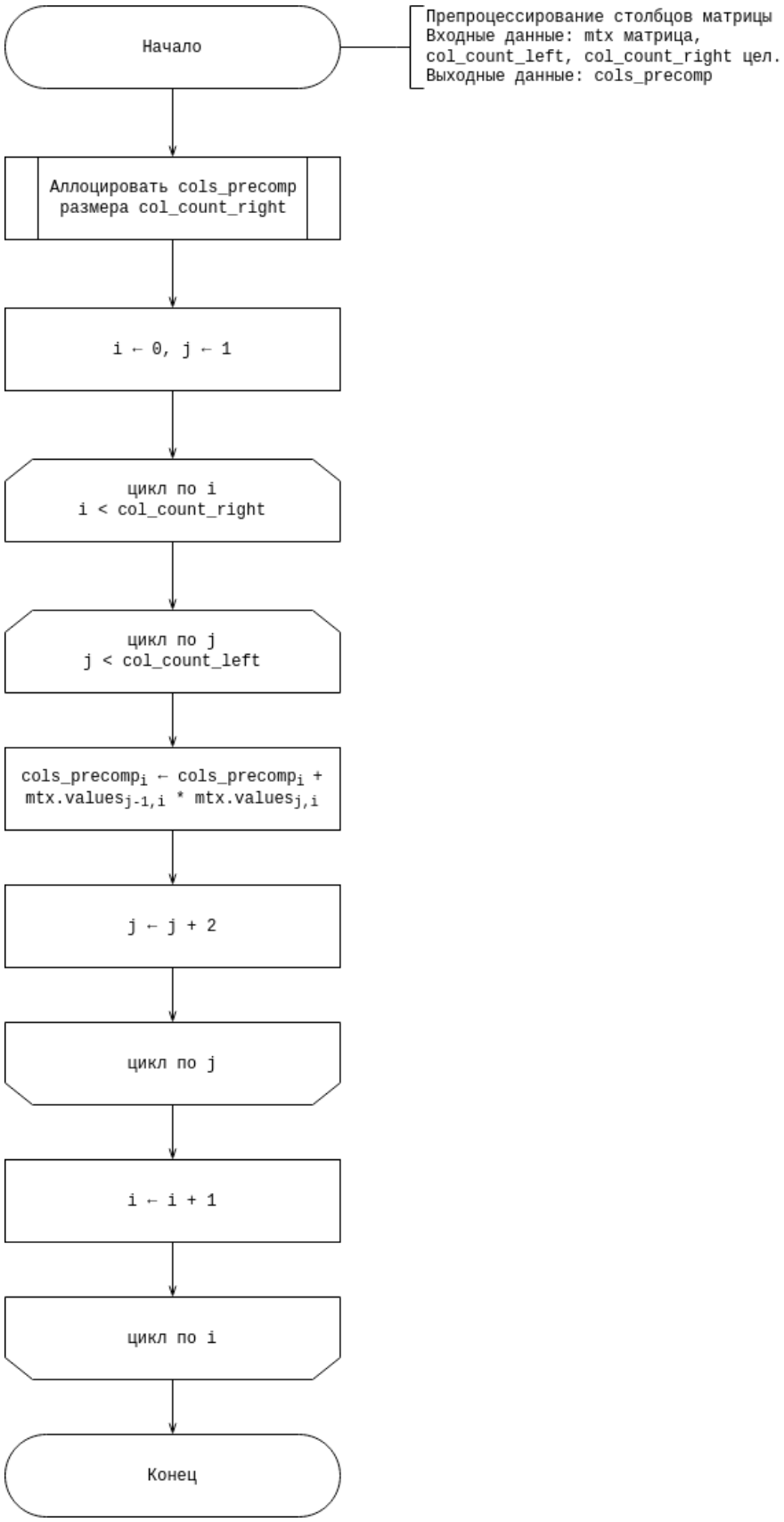

Рисунок 2.6 – Схема алгоритма препроцессирования столбцов

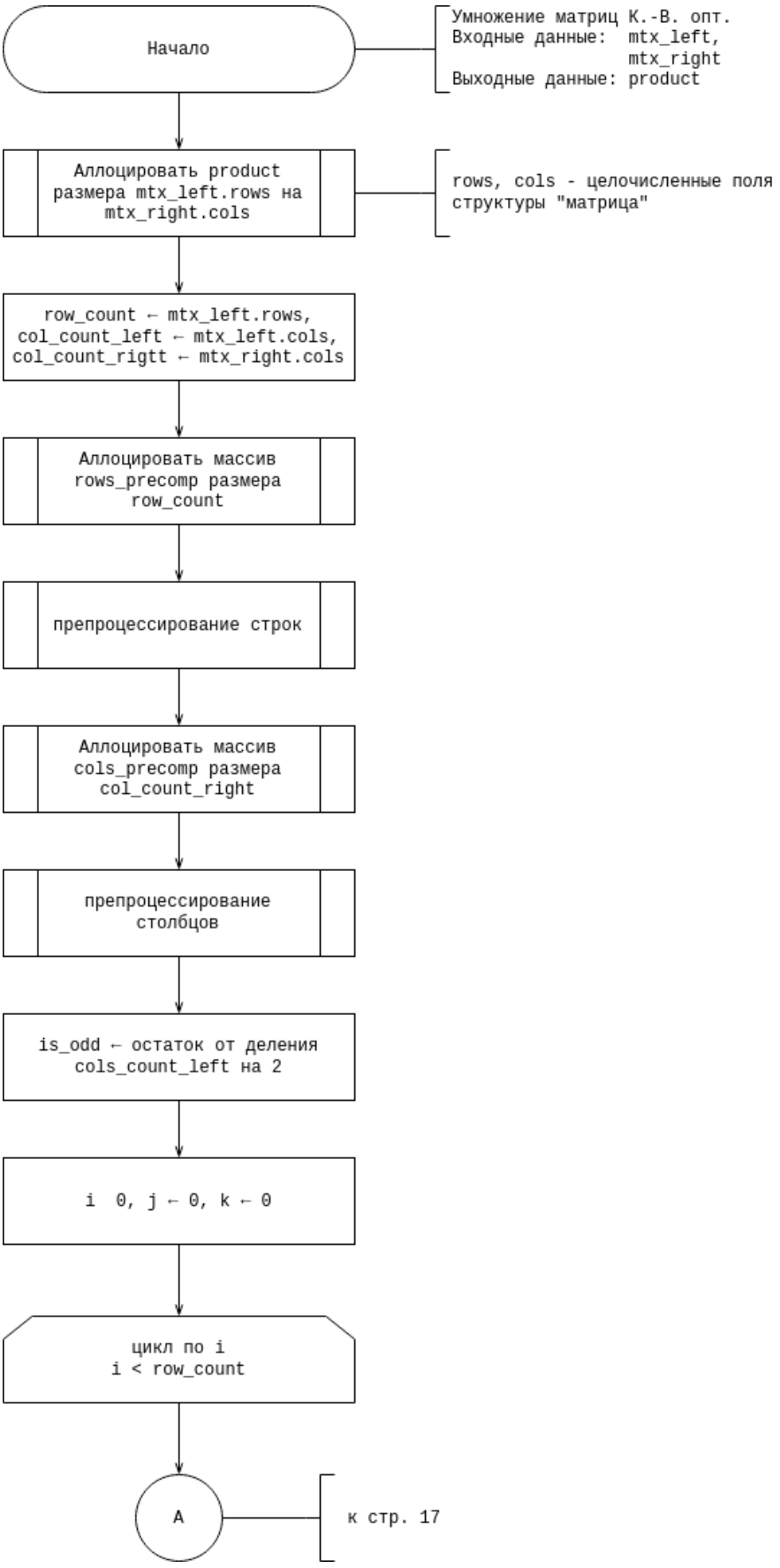

Рисунок 2.7 – Схема оптимизированного алгоритма умножения матриц Копперсмита – Винограда (начало)

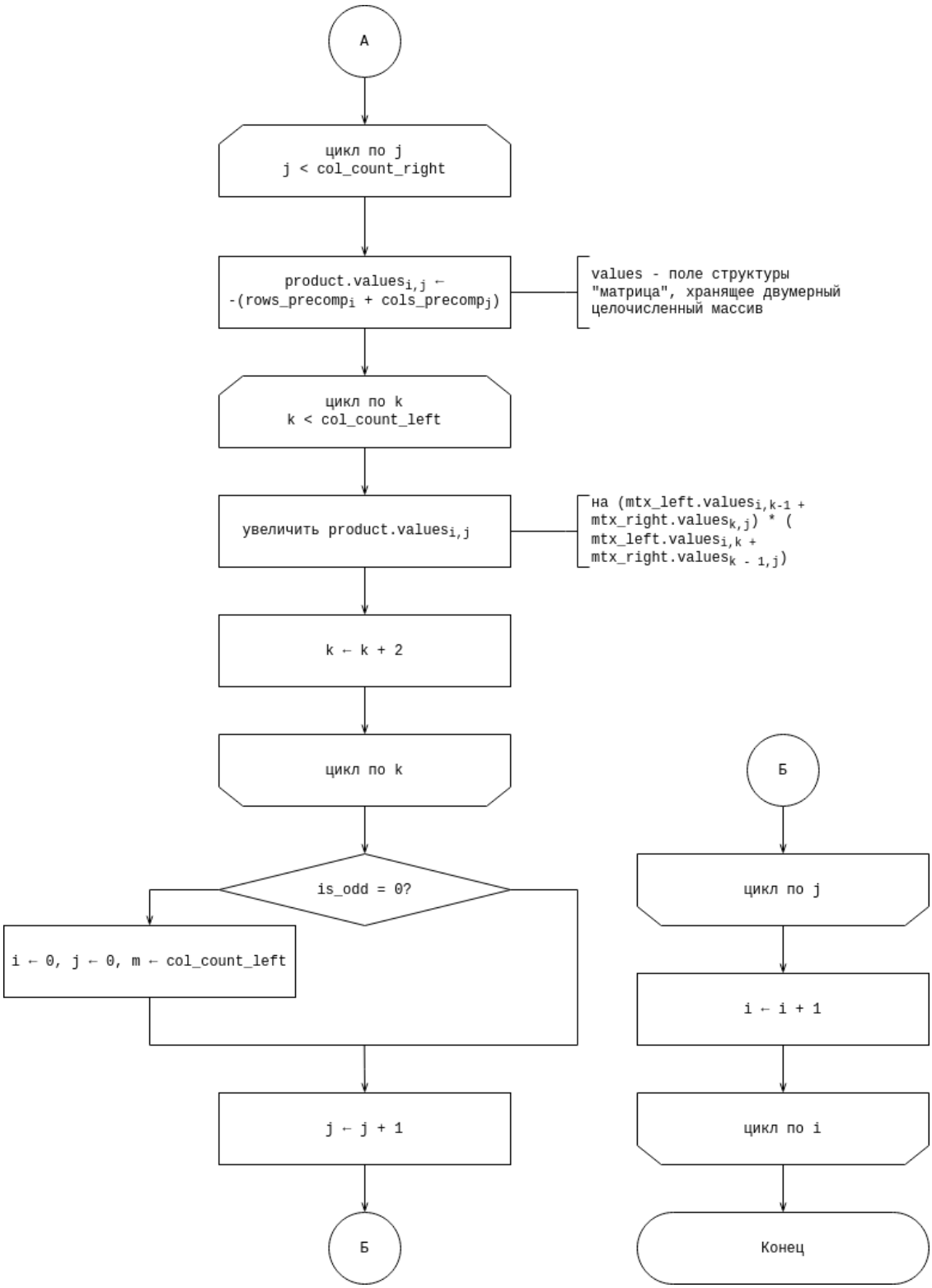

Рисунок 2.8 – Схема оптимизированного алгоритма умножения матриц Копперсмита – Винограда (конец)

Вывод

Алгоритмы были проанализированы с точки зрения временных затрат. Было выявлено, что алгоритм Копперсмита – Винограда работает на $0.5MNK$ быстрее, чем классический матричный. Оптимизация алгоритма же дает выигрыш на $1.5MNK$.

Были построены схемы алгоритмов. Теоретически были исследованы способы оптимизации алгоритма Копперсмита – Винограда. Было получено достаточно теоретических сведений для разработки ПО, решающего поставленную задачу.

3. Технологический раздел

3.1 Требования к ПО

Программное обеспечение должно удовлетворять следующим требованиям:

- Программа получает на вход с клавиатуры две матрицы размеров в пределах 10000×10000 либо получает два числа – размерность матрицы в пределах 10000;
- Программа выдает матрицу - произведение двух полученных матриц.

3.2 Средства реализации

Для реализации ПО был выбран язык программирования Golang, поскольку язык отличается легкой и быстрой сборкой программ и автоматическим управлением памяти. Также язык обеспечивает возможность вставки на языке C, что позволит исследовать реализованные алгоритмы с точки зрения временных затрат.

3.3 Листинги кода

Листинг 3.1 демонстрирует классический алгоритм умножения.

Листинг 3.1 – Классический алгоритм умножения

```
1  func Algebraic(mtx_left, mtx_right mtx.Matrix) mtx.Matrix {
2      var product mtx.Matrix
3      product = mtx.Allocate(mtx_left.Rows, mtx_right.Cols)
4
5      for i := 0; i < product.Rows; i++ {
6          for j := 0; j < product.Cols; j++ {
7              for k := 0; k < mtx_right.Rows; k++ {
8                  product.Values[i][j] += mtx_left.Values[i][k] +
9                   * mtx_right.Values[k][j]
10             }
```

```
        }
    }

    return product
}
```

Листинг 3.2 – умножение матриц алгоритмом Винограда.

Листинг 3.2 – Алгоритм умнложения Виноградом

```
func Winograd(mtx_left, mtx_right mtx.Matrix) mtx.Matrix {
    var (
    row_count, col_count_left, col_count_right int
    product mtx.Matrix
    rows_precomp, cols_precomp []int
    )
    row_count = mtx_left.Rows
    col_count_left = mtx_left.Cols
    col_count_right = mtx_right.Cols

    product = mtx.Allocate(row_count, col_count_right)

    rows_precomp = __precomputeRows(rows_precomp, mtx_left,
                                    row_count, col_count_left)
    cols_precomp = __precomputeCols(cols_precomp, mtx_right,
                                    col_count_left, col_count_right)
    for  i := 0; i < row_count; i++ {
        for j := 0; j < col_count_right; j++ {
            product.Values[i][j]= -(rows_precomp[i] + cols_precomp[j])
            for k := 1; k < col_count_left; k+= 2 {
                product.Values[i][j] = product.Values[i][j] +
                + (mtx_left.Values[i][k - 1] + +
                + mtx_right.Values[k][j]) * +
                (mtx_left.Values[i][k] + mtx_right.Values[k - 1][j])
            }
        }
    }
    if (col_count_left % 2 != 0) {
    for  i := 0; i < row_count; i++ {
        for j := 0; j < col_count_right; j++ {
            product.Values[i][j] = product.Values[i][j] + +
            mtx_left.Values[i][col_count_left - 1] * +
            mtx_right.Values[col_count_left - 1][j]
        }
    }
}
return product
}
```

Листинг 3.3 – умножение оптимизированным согласно 2.2 алгоритмом винограда.

Листинг 3.3 – Оптимизированный алгоритм умнложения Виноградом

```
func WinogradOptimized(mtx_left, mtx_right mtx.Matrix) mtx.Matrix {
    var (
    row_count, col_count_left, col_count_right int
    product mtx.Matrix
    rows_precomp, cols_precomp []int
    )
    row_count = mtx_left.Rows
    col_count_left = mtx_left.Cols
    col_count_right = mtx_right.Cols

    product = mtx.Allocate(row_count, col_count_right)

    rows_precomp =
                __precomputeRowsOptimized(rows_precomp, mtx_left,
                                          row_count, col_count_left)
    cols_precomp =
                __precomputeColsOptimized(cols_precomp, mtx_right,
                                          col_count_left, col_count_right)

    var i, j, k int
    is_odd := col_count_left % 2
    for  i = 0; i < row_count; i++ {
        for j = 0; j < col_count_right; j++ {
            product.Values[i][j]= -(rows_precomp[i] + cols_precomp[j])
            for k = 1; k < col_count_left; k+= 2 {
                product.Values[i][j] += (mtx_left.Values[i][k - 1] +
                + mtx_right.Values[k][j]) * (mtx_left.Values[i][k] +
                + mtx_right.Values[k - 1][j])
            }
            if (is_odd == 1) {
                product.Values[i][j] +=
                mtx_left.Values[i][col_count_left - 1] * +
                + mtx_right.Values[col_count_left - 1][j]
            }
        }
    }

    return product
}
```

Функции оптимизации, реализующие формулу 1.3 представлены на листинге 3.4

Листинг 3.4 – Функции препроцессирования для алгоримов Винограда

```
func __precomputeRowsOptimized(rows_precomp []int, mtx_left mtx.Matrix,
row_count, col_count int) []int {
    rows_precomp = make([]int, row_count)
    for i := 0; i < row_count; i++ {
        for j := 1; j < col_count; j+= 2 {
            rows_precomp[i] += mtx_left.Values[i][j - 1] +
            * mtx_left.Values[i][j]
        }
    }
    return rows_precomp
}

func __precomputeColsOptimized(cols_precomp []int, mtx_right mtx.Matrix,
col_count_left, col_count_right int) []int {
    cols_precomp = make([]int, col_count_right)
    for i := 0; i < col_count_right; i++ {
        for j := 1; j < col_count_left; j+=2 {
            cols_precomp[i] += (mtx_right.Values[j - 1][i] +
            * mtx_right.Values[j][i])
        }
    }
    return cols_precomp
}
```

Вспомогательные функции для работы с многомерным целочисленным массивом представлены ниже(3.5):

Листинг 3.5 – Пакет реализующий функции для работы с матрицами

```
type Matrix struct {
    Rows int
    Cols int
    Values [][]int
}

func Allocate(rows, cols int) Matrix {
    var dest Matrix

    dest.Rows = rows
    dest.Cols = cols
    dest.Values = make([][]int, dest.Rows)
    for i := range dest.Values {
        dest.Values[i] = make([]int, dest.Cols)
    }

    return dest
}
```

```
19
20 func Randomize(src Matrix) {
21     rand.Seed(time.Now().Unix())
22     for i := 0; i < src.Rows; i++ {
23         for j := 0; j < src.Cols; j++ {
24             src.Values[i][j] = rand.Intn(10) - rand.Intn(10)
25         }
26     }
27 }
```

3.4 Тестирование ПО

Таблица 3.1 – Тестовые случаи

№	Входная матрица №1	Входная матрица №2	Результат		
			AG	CW	CW OPT
1	$\begin{matrix} -3 & 5 & -1 & 7 \\ -8 & 2 & -2 & 1 \\ 0 & -3 & -4 & 0 \\ -6 & 0 & 5 & 1 \end{matrix}$	$\begin{matrix} -3 & 5 & -1 & 7 \\ -8 & 2 & -2 & 1 \\ 0 & -3 & -4 & 0 \\ -6 & 0 & 5 & 1 \end{matrix}$	$\begin{matrix} -73 & -2 & 32 & -9 \\ 2 & -30 & 17 & -53 \\ 24 & 6 & 22 & -3 \\ 12 & -45 & -9 & -41 \end{matrix}$	$\begin{matrix} -73 & -2 & 32 & -9 \\ 2 & -30 & 17 & -53 \\ 24 & 6 & 22 & -3 \\ 12 & -45 & -9 & -41 \end{matrix}$	$\begin{matrix} -73 & -2 & 32 & -9 \\ 2 & -30 & 17 & -53 \\ 24 & 6 & 22 & -3 \\ 12 & -45 & -9 & -41 \end{matrix}$
2	$\begin{matrix} 0 & 0 & -2 \\ 2 & 3 & 2 \\ -4 & 0 & -1 \end{matrix}$	$\begin{matrix} 0 & 0 & -2 \\ 2 & 3 & 2 \\ -4 & 0 & -1 \end{matrix}$	$\begin{matrix} 8 & 0 & 2 \\ -2 & 9 & 0 \\ 4 & 0 & 9 \end{matrix}$	$\begin{matrix} 8 & 0 & 2 \\ -2 & 9 & 0 \\ 4 & 0 & 9 \end{matrix}$	$\begin{matrix} 8 & 0 & 2 \\ -2 & 9 & 0 \\ 4 & 0 & 9 \end{matrix}$
3	$\begin{matrix} 5 & 7 \\ 7 & 0 \\ 3 & 1 \end{matrix}$	$\begin{matrix} 5 & 7 & 7 & 0 \\ 3 & 1 & -7 & 5 \end{matrix}$	$\begin{matrix} 46 & 42 & -14 & 35 \\ 35 & 49 & 49 & 0 \\ 18 & 22 & 14 & 5 \end{matrix}$	$\begin{matrix} 46 & 42 & -14 & 35 \\ 35 & 49 & 49 & 0 \\ 18 & 22 & 14 & 5 \end{matrix}$	$\begin{matrix} 46 & 42 & -14 & 35 \\ 35 & 49 & 49 & 0 \\ 18 & 22 & 14 & 5 \end{matrix}$
4	$\begin{matrix} -4 & 9 \\ -4 & -1 \\ -1 & 5 \\ 5 & 3 \end{matrix}$	$\begin{matrix} -4 & 9 & -4 \\ -1 & -1 & 5 \end{matrix}$	$\begin{matrix} 7 & -45 & 61 \\ 17 & -35 & 11 \\ -1 & -14 & 29 \\ -23 & 42 & -5 \end{matrix}$	$\begin{matrix} 7 & -45 & 61 \\ 17 & -35 & 11 \\ -1 & -14 & 29 \\ -23 & 42 & -5 \end{matrix}$	$\begin{matrix} 7 & -45 & 61 \\ 17 & -35 & 11 \\ -1 & -14 & 29 \\ -23 & 42 & -5 \end{matrix}$

3.5 Вывод

Было написано и протестировано программное обеспечение для решения поставленной задачи.

4. Исследовательский раздел

4.1 Технические характеристики

Тестирование выполнялось на устройстве со следующими техническими характеристиками:

- Операционная система Ubuntu 20.04.1 LTS;
- Память 7 GiB
- Процессор Intel(R) Core(TM) i3-8145U CPU @ 2.10GHz [2]

Во время тестирования устройство было подключено к блоку питания и не нагружено никакими приложениями, кроме встроенных приложений окружения, окружением и системой тестирования.

4.2 Описание системы тестирования

Для замеров процессорного времени была написана вставка на языке C, используя пакет cgo [3]. Утилита для замера времени приведена на листинге 4.1.

Листинг 4.1 – Вставка на языке C для замера времени

```
/*
static double GetCpuTime() {
    struct timespec time;
    if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &time)) {
        perror("clock_gettime");
        return 0;
    }

    long seconds = time.tv_sec;
    long nanoseconds = time.tv_nsec;
    double elapsed = seconds + nanoseconds*1e-9;

    return elapsed;
}
*/
```

4.3 Таблица времени выполнения алгоритмов

Результаты профилирования алгоритмов приведены в таблице 4.1. Результаты тестирования приведены в таблице 3.1. Здесь CW – алгоритм Копперсмита – Винограда, ALG – классический и CW OPT – оптимизированный Копперсмита – Винограда.

Таблица 4.1 – Время выполнения алгоритмов

Таблица 4.2.1 – Четная размерность матриц

n	Время, ns		
	ALG	CW	CW OPT
10	4.49e-06	3.91e-06	4.01e-06
20	2.74e-05	2.24e-05	2.30e-05
30	8.84e-05	6.72e-05	7.01e-05
40	2.09e-04	1.49e-04	1.58e-04
50	3.96e-04	2.98e-04	3.14e-04
60	6.58e-04	4.95e-04	5.15e-04
70	1.07e-03	7.95e-04	8.43e-04
80	1.60e-03	1.20e-03	1.22e-03
90	2.26e-03	1.68e-03	1.77e-03
100	3.14e-03	2.25e-03	2.38e-03
150	1.06e-02	7.88e-03	8.33e-03
200	2.84e-02	2.20e-02	2.34e-02
250	6.73e-02	5.34e-02	5.74e-02
300	1.14e-01	8.97e-02	9.60e-02
350	1.80e-01	1.39e-01	1.50e-01
400	2.67e-01	2.05e-01	2.21e-01
450	4.17e-01	3.23e-01	3.48e-01
500	5.71e-01	4.42e-01	4.75e-01

Таблица 4.2.2 – Нечетная размерность матриц

n	Время, ns		
	ALG	CW OPT	CW
11	5.17e-06	4.56e-06	5.05e-06
21	3.25e-05	2.58e-05	2.59e-05
31	9.63e-05	7.23e-05	7.69e-05
41	2.14e-04	1.64e-04	1.73e-04
51	4.10e-04	3.15e-04	3.38e-04
61	7.13e-04	5.37e-04	5.70e-04
71	1.11e-03	8.36e-04	8.79e-04
81	1.67e-03	1.28e-03	1.35e-03
91	2.33e-03	1.73e-03	1.85e-03
101	3.23e-03	2.36e-03	2.49e-03
151	1.08e-02	8.03e-03	8.50e-03
201	2.74e-02	2.13e-02	2.29e-02
251	6.80e-02	5.27e-02	5.66e-02
301	1.16e-01	9.13e-02	9.82e-02
351	1.82e-01	1.40e-01	1.52e-01
401	2.58e-01	1.99e-01	2.14e-01
451	4.21e-01	3.26e-01	3.50e-01
501	5.74e-01	4.43e-01	4.76e-01

4.4 Графики функций

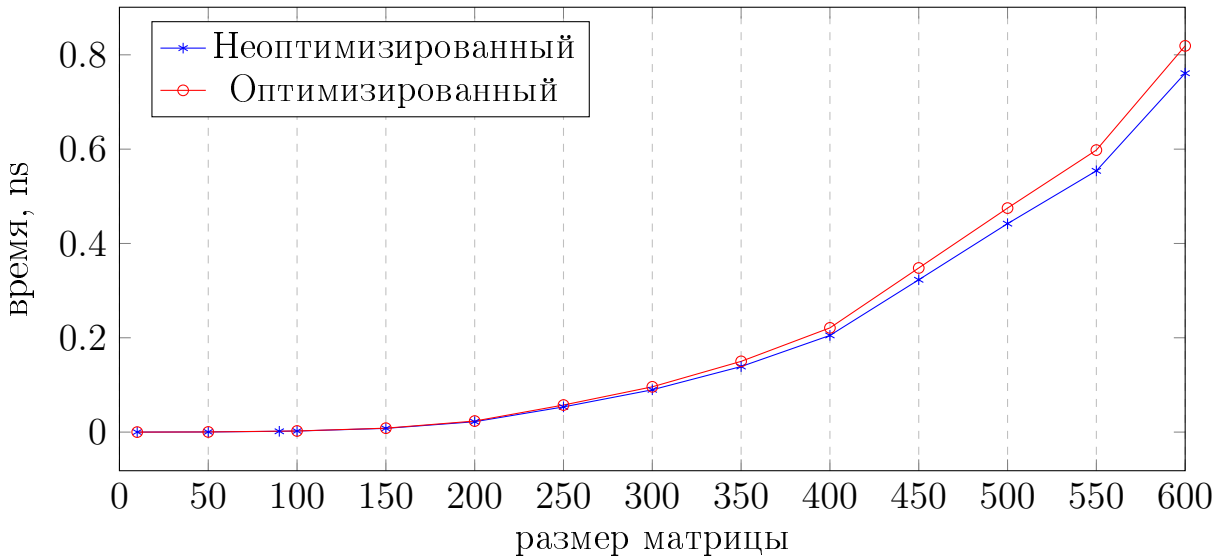

Рисунок 4.1.1 – Четная размерность матрицы

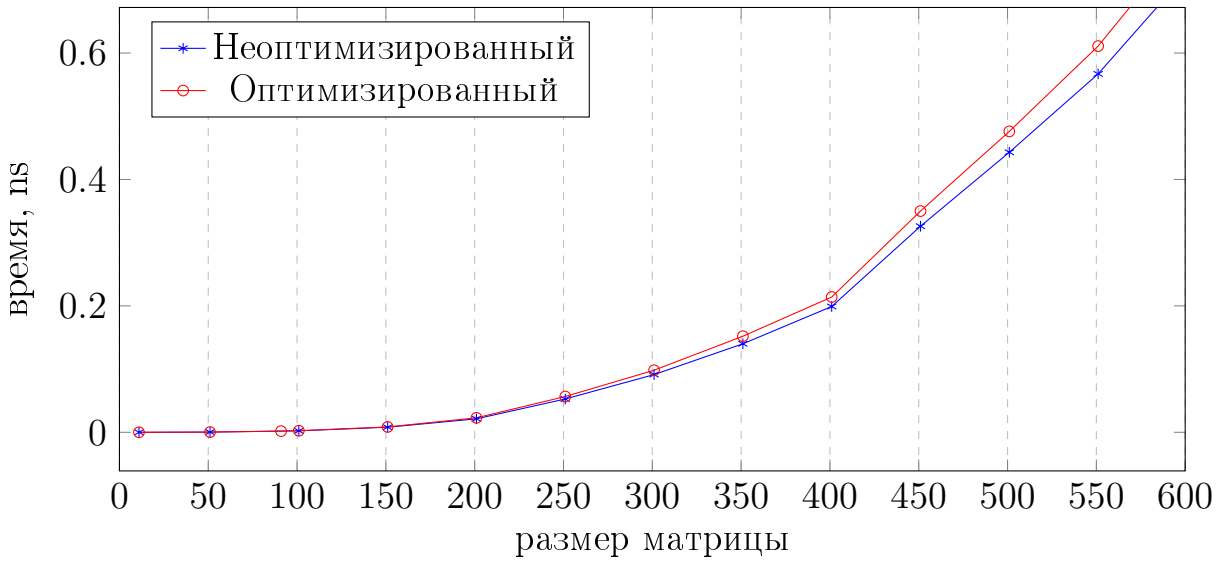

Рисунок 4.1.2 – Нечетная размерность матрицы

Рисунок 4.1 – Время выполнения алгоритма Копперсмита – Винограда

Как было доказано в разделе 2.3, оптимизация дает выигрыш по времени практически незначительный. Выигрыш растет с ростом размерности матрицы – в случае матриц с четной размерностью, выигрыш на матрице 550×550 на 7,94%, а на матрицах с нечетной (551×551) – на 7,76%.

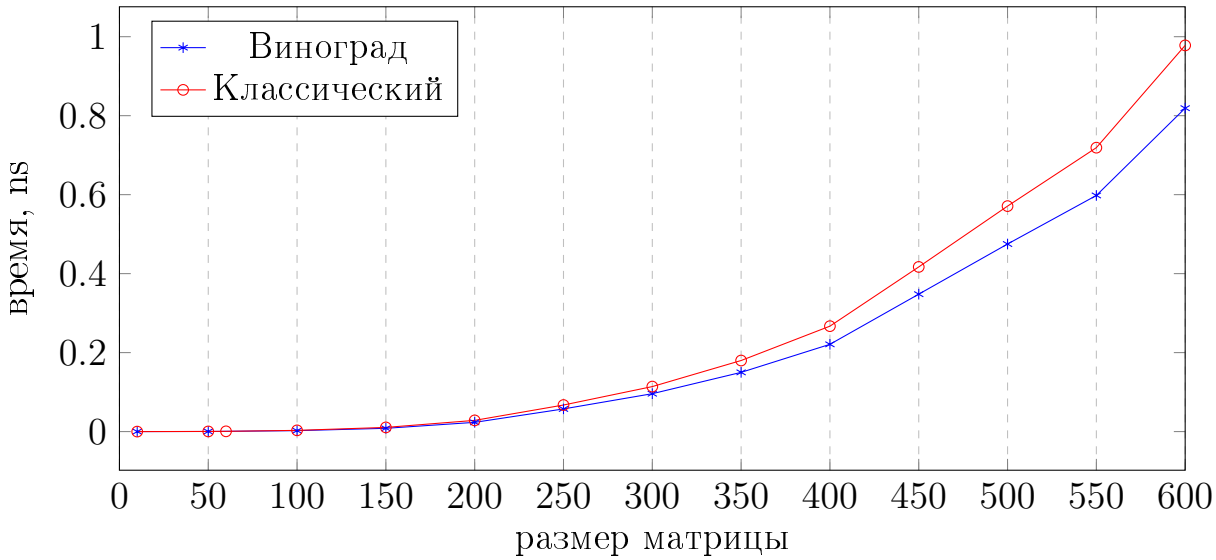

Рисунок 4.2.1 – Четная размерность матрицы

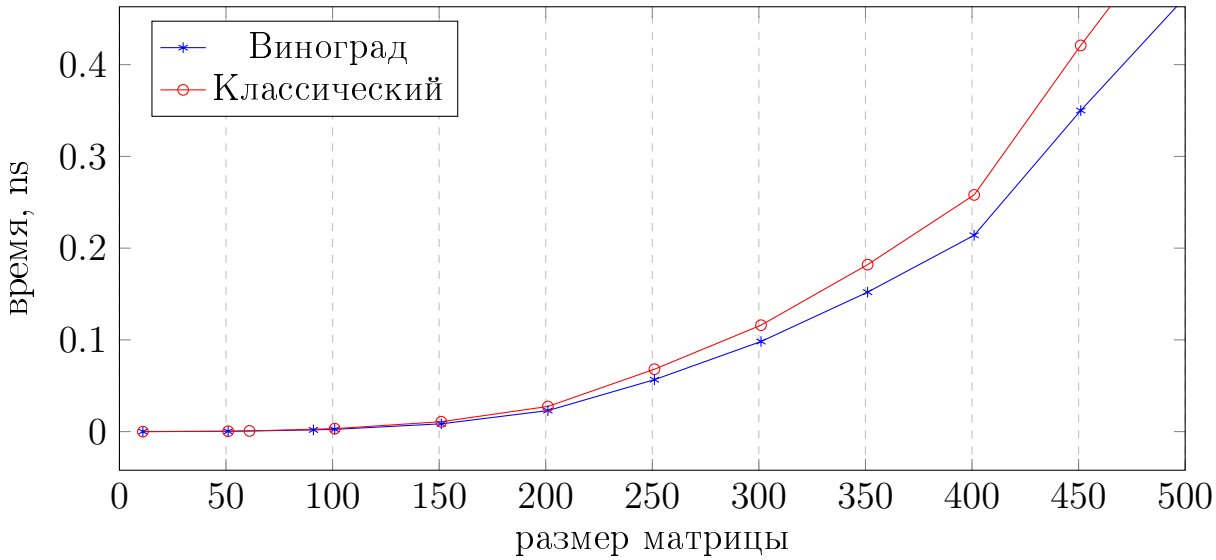

Рисунок 4.2.2 – Нечетная размерность матрицы

Рисунок 4.2 – Сравнение времени выполнения алгоритма Копперсмита – Винограда с классическим матричным

На практике классический показывает результат хуже (4.2), чем алгоритм Копперсмита – Винограда. Связано это с меньшим количеством более затратных операций. Эта разница, в среднем, составляет 3%

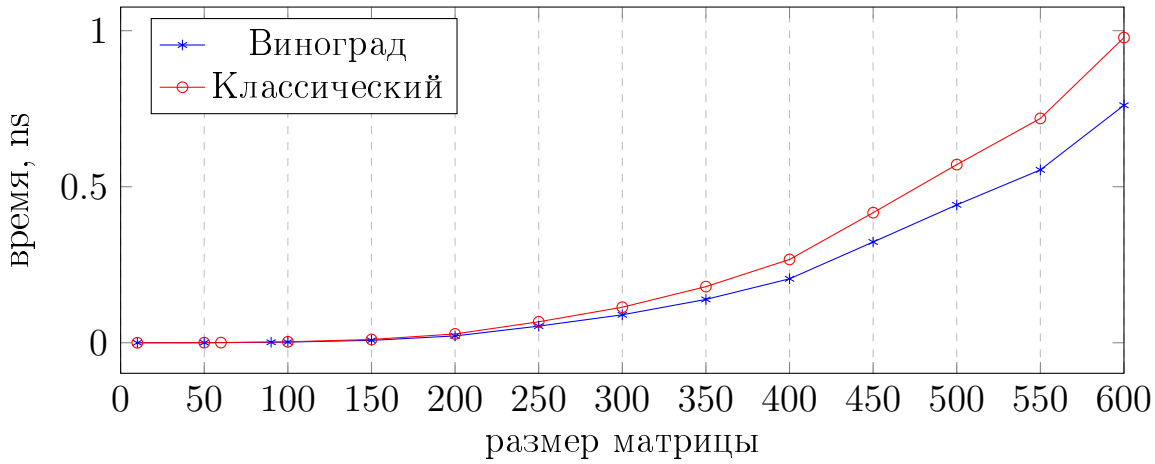

Рисунок 4.3.1 – Четная размерность матрицы

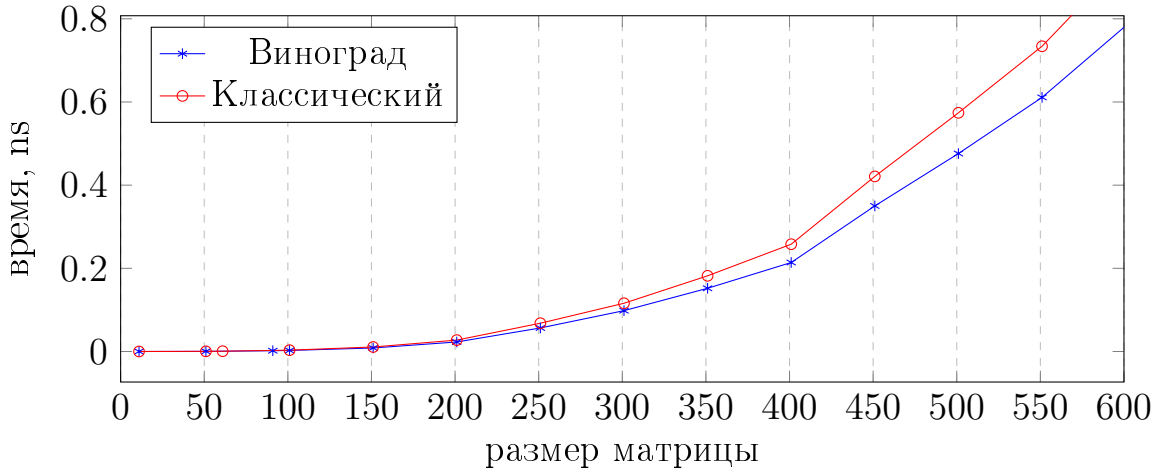

Рисунок 4.3.2 – Нечетная размерность матрицы

Рисунок 4.3 – Сравнение времени выполнения оптимизированного алгоритма Копперсмита – Винограда с классическим матричным

Аналогичная ситуации 4.2 ситуация наблюдается в случае оптимизированного алгоритма (4.3) – алгоритм имеет примерно такой же выигрыш.

4.5 Вывод

Экспериментально было подтверждено утверждение, установленное в разделе 1 – алгоритм Копперсмита – Винограда имеет незначительный выигрыш на матрицах, способных храниться на ЭВМ. Оптимизация стандартного алгоритма так же обеспечивает небольшой выигрыш $\approx 7\%$.

Заключение

Электронная вычислительная техника эволюционирует непрерывно и стремительно. Но, даже при таких темпах, возникают ситуации, когда алгоритмы имеют значительный выигрыш только на таких данных, хранение которых невозможно современными ЭВМ. Проанализированный алгоритм на небольших данных имеет выигрыш во времени в среднем на 5%, что связано с меньшим количеством трудоемких операций и препроцессировании части данных. В этом и состоит главная идея алгоритма.

Алгоритм Копперсмита – Винограда считался самым быстрым до 2010 года. На декабрь 2020 года самым быстрым алгоритмом считается алгоритм, доказанный математиком Вирджинией Василевской-Вильямс[4]. Поэтому в настоящее время алгоритм не имеет преимуществ и используется только в теории для доказательства некоторых математических тождеств.

Список литературы

[1] Don Coppersmith и Shmuel Winograd. “Matrix Multiplication via Arithmetic Progressions”. в: *Journal of Symbolic Computation* (1990).

[2] *Процессор Intel® Core™ i3-8145U*. URL: `https://ark.intel.com/content/www/ru/ru/ark/products/149090/intel-core-i3-8145u-processor-4m-cache-up-to-3-90-ghz.html` (дата обр. 29.09.2021).

[3] *Using cgo with the go command*. URL: `https://pkg.go.dev/cmd/cgo` (дата обр. 29.09.2021).

[4] Virginia Vassilevska Williams Josh Alman. “A Refined Laser Method and Faster Matrix Multiplication”. в: *32nd Annual ACM-SIAM Symposium on Discrete Algorithms* (2021).