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 курсу "Анализ алгоритмов"

Тема Алгоритмы умножения матриц

Студент Леонов В.В.

Группа ИУ7-56Б

Оценка (баллы) _____

Преподаватель Волкова Л.Л.

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.

Целью данной лабораторной работы являются изучение и реализация алгоритмов умножения матриц с оценкой их трудоемкости.

Для достижения указанной выше цели следует выполнить следующие задачи:

- изучить алгоритмы умножения матриц: стандартный алгоритм и алгоритм Винограда;
- оптимизировать алгоритм Винограда;
- дать теоритическую оценку трудоемкости классического алгоритма умножения матриц, алгоритма Винограда и улучшеного алгоритма Винограда;
- провести сравнительный анализ алгоритмов на основе эксперементальных данных;
- описание и обоснование полученных результатов в отчете о выполненной лабораторной работе.

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):

$$V \cdot W = (v_1 + w_2)(v_2 + w_1) + (v_3 + w_4)(v_4 + w_3) - v_1 v_2 - v_3 v_4 - w_1 w_2 - w_3 w_4. \quad (1.4)$$

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 Вывод

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 представлены схемы алгоритмов, описанных в аналитическом разделе, а также описана их трудоемкость.

2.1 Схемы алгоритмов

Схема классического алгоритма приведена на рисунке 2.1.

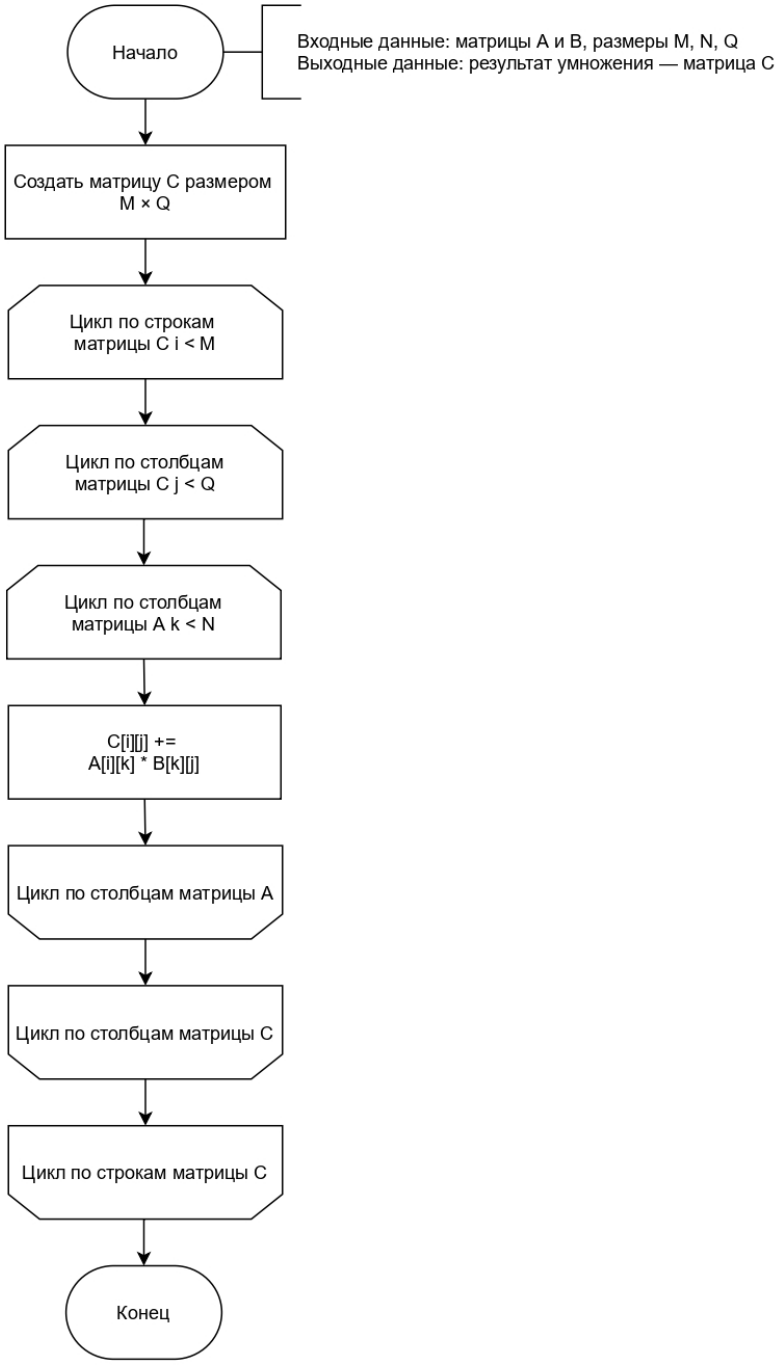

Рисунок 2.1 – Схема классического алгоритма

Схема алгоритма Винограда без оптимизаций приведена на рисунках 2.2-2.3.

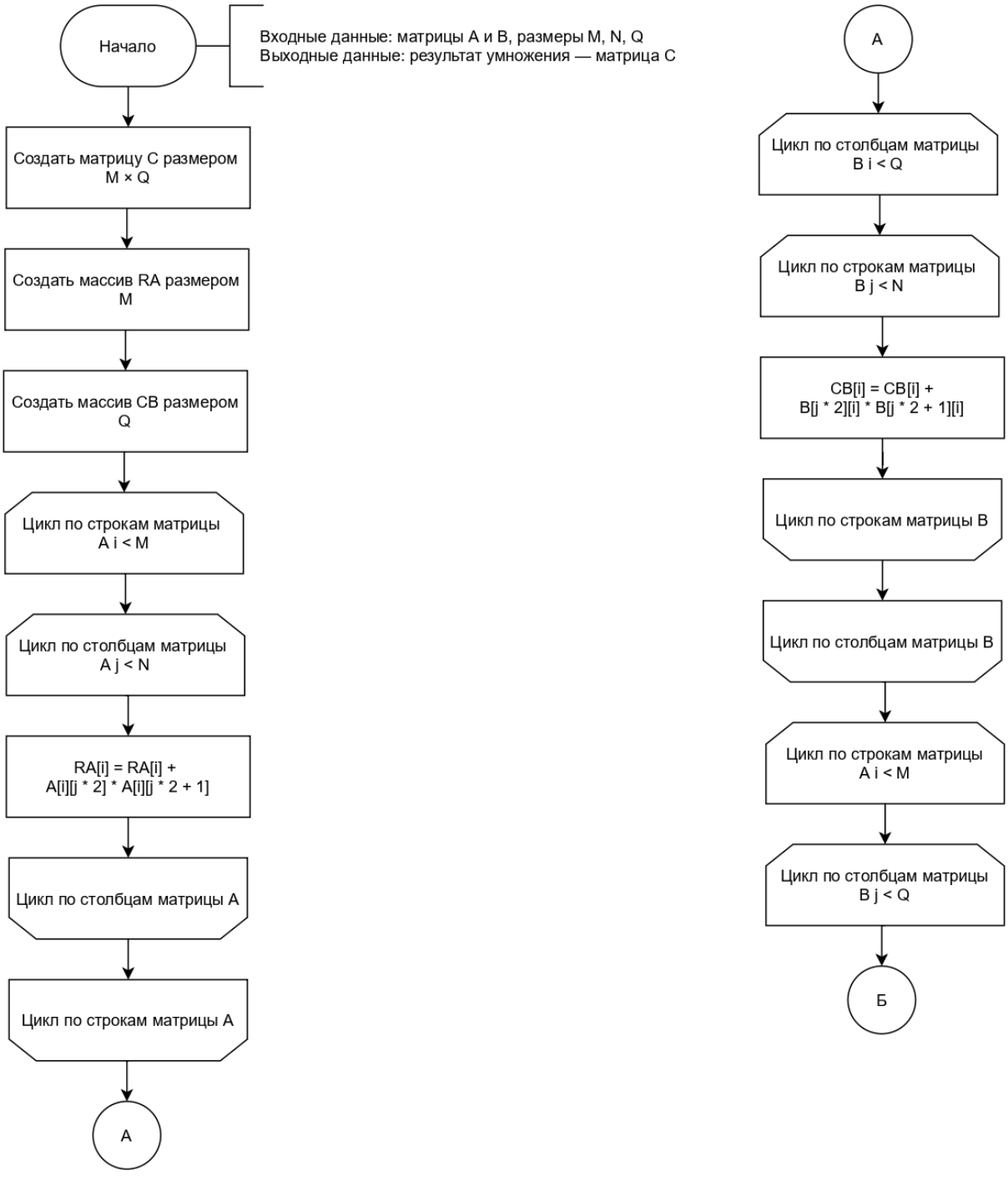

Рисунок 2.2 – Схема алгоритма Винограда (часть 1)

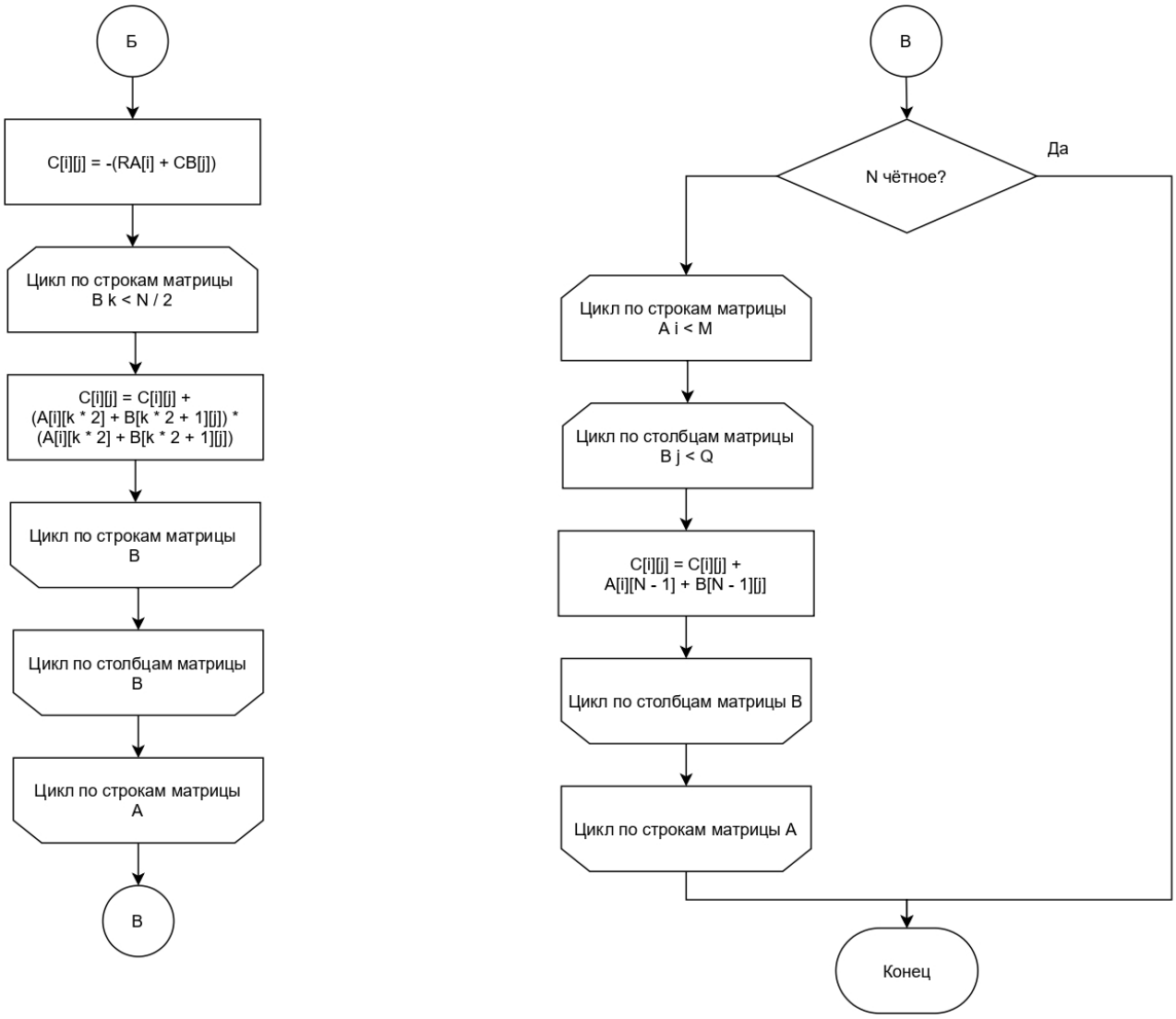

Рисунок 2.3 – Схема алгоритма Винограда (часть 2)

Схема алгоритма Винограда с оптимизациями приведена на рисунках 2.4-2.5.

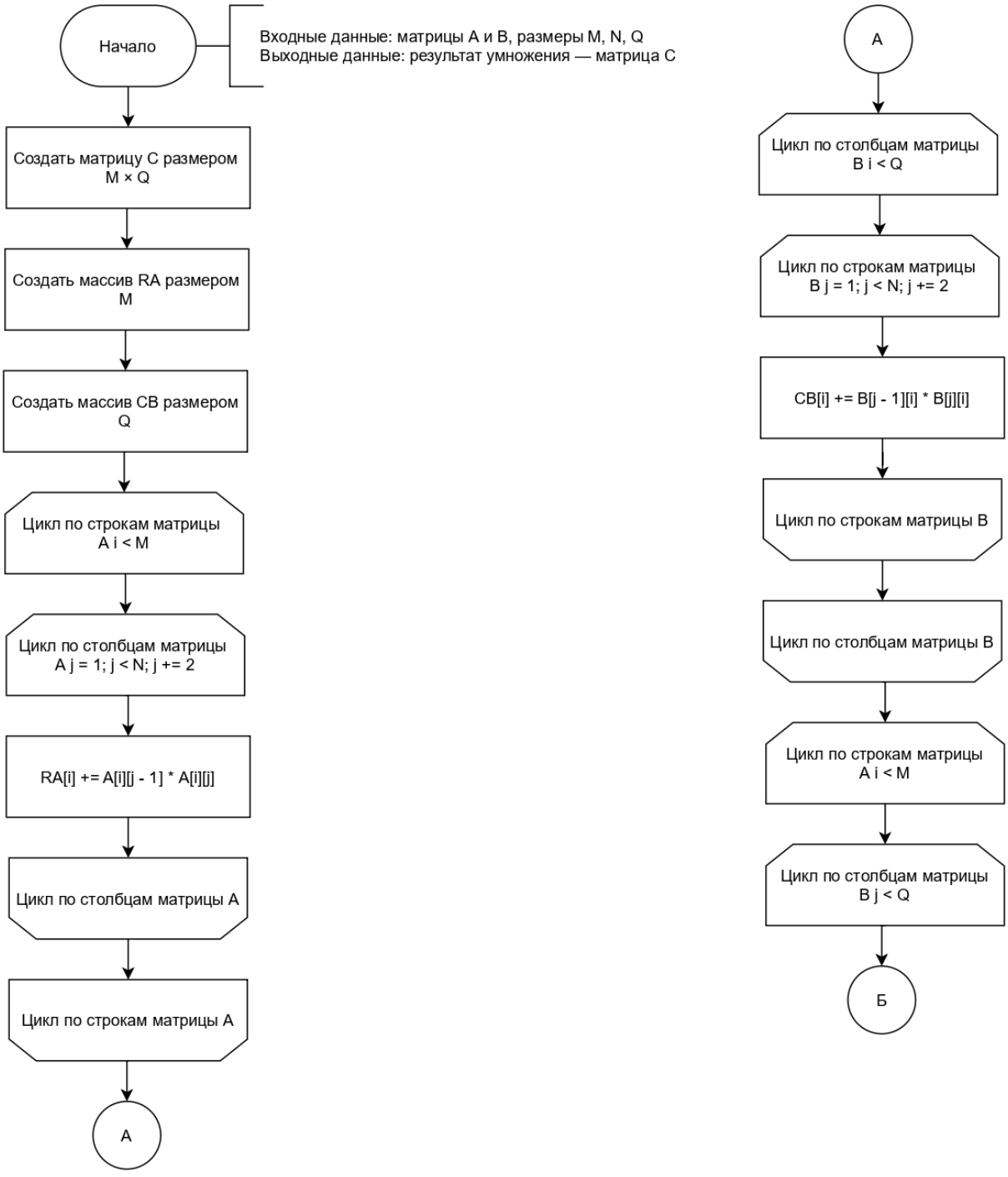

Рисунок 2.4 – Схема алгоритма Винограда с оптимизациями (часть 1)

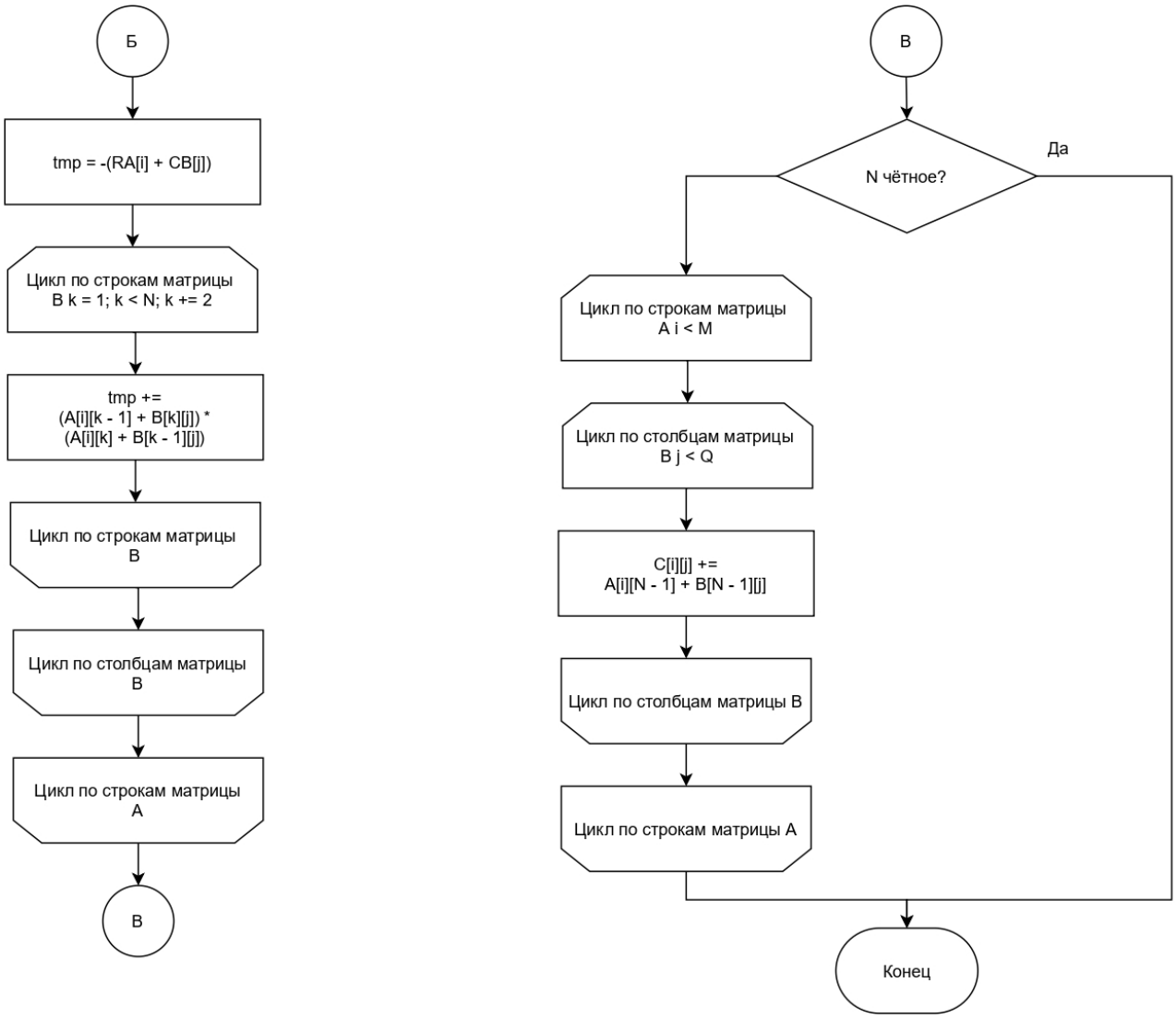

Рисунок 2.5 – Схема алгоритма Винограда с оптимизациями (часть 2)

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

Данный алгоритм можно оптимизировать следующими методами [2]:

- предварительно получить строки столбцы соответствующих матриц;
- заменить вызов функции вычисления длины вектора на заранее вычисленное значение;
- уменьшить количество обращений к элементам матрицы путем добавления буферной переменной;
- вынести конструкции if-then-else за пределы лямбда-функции.

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).

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

4. Трудоемкость вызова функции равна 0.

2.3 Трудоемкость алгоритмов

2.3.1 Стандартный алгоритм умножения матриц

Трудоёмкость стандартного алгоритма умножения матриц состоит из:

- внешнего цикла по $i \in [1..A]$, трудоёмкость которого: $f = 2 + A \cdot (2 + f_{body})$;
- цикла по $j \in [1..C]$, трудоёмкость которого: $f = 2 + C \cdot (2 + f_{body})$;
- скалярного умножения двух векторов - цикл по $k \in [1..B]$, трудоёмкость которого: $f = 2 + 10B$.

Трудоёмкость стандартного алгоритма равна трудоёмкости внешнего цикла, можно вычислить ее, подставив циклы тела (2.4):

$$f_{base} = 2 + A \cdot (4 + C \cdot (4 + 10B)) = 2 + 4A + 4AC + 10ABC \approx 10ABC \quad (2.4)$$

2.3.2 Алгоритм Копперсмита — Винограда

Трудоёмкость алгоритма Копперсмита — Винограда состоит из:

1. Создания векторов rows и cols (2.5):

$$f_{create} = A + C; \quad (2.5)$$

2. Заполнения вектора rows (2.6):

$$f_{rows} = 3 + \frac{B}{2} \cdot (5 + 12A); \quad (2.6)$$

3. Заполнения вектора cols (2.7):

$$f_{cols} = 3 + \frac{B}{2} \cdot (5 + 12C); \quad (2.7)$$

4. Цикла заполнения матрицы для чётных размеров (2.8):

$$f_{cycle} = 2 + A \cdot (4 + C \cdot (11 + \frac{25}{2} \cdot B)); \tag{2.8}$$

5. Цикла, для дополнения умножения суммой последних нечётных строки и столбца, если общий размер нечётный (2.9):

$$f_{last} = \begin{cases} 2, & \text{чётная,} \\ 4 + A \cdot (4 + 14C), & \text{иначе.} \end{cases} \tag{2.9}$$

Итого, для худшего случая (нечётный размер матриц):

$$f_{wino\_w} = A+C+12+8A+5B+6AB+6CB+25AC+\frac{25}{2}ABC \approx 12.5 \cdot MNK \tag{2.10}$$

Для лучшего случая (чётный размер матриц):

$$f_{wino\_b} = A+C+10+4A+5B+6AB+6CB+11AC+\frac{25}{2}ABC \approx 12.5 \cdot MNK \tag{2.11}$$

2.3.3 Оптимизированный алгоритм Копперсмита — Винограда

Трудоёмкость улучшенного алгоритма Копперсмита — Винограда состоит из:

1. Создания векторов rows и cols (2.12):

$$f_{init} = A + C; \tag{2.12}$$

2. Заполнения вектора rows (2.13):

$$f_{rows} = 2 + \frac{B}{2} \cdot (4 + 8A); \tag{2.13}$$

3. Заполнения вектора cols (2.14):

$$f_{cols} = 2 + \frac{B}{2} \cdot (4 + 8A); \tag{2.14}$$

4. Цикла заполнения матрицы для чётных размеров (2.15):

$$f_{cycle} = 2 + A \cdot (4 + C \cdot (8 + 9B)) \tag{2.15}$$

5. Цикла, для дополнения умножения суммой последних нечётных строки и столбца, если общий размер нечётный (2.16):

$$f_{last} = \begin{cases} 2, & \text{чётная,} \\ 4 + A \cdot (4 + 12C), & \text{иначе.} \end{cases} \tag{2.16}$$

Итого, для худшего случая (нечётный общий размер матриц) имеем (2.17):

$$f = A + C + 10 + 4B + 4BC + 4BA + 8A + 20AC + 9ABC \approx 9ABC \tag{2.17}$$

Для лучшего случая (чётный общий размер матриц) имеем (2.18):

$$f = A + C + 8 + 4B + 4BC + 4BA + 4A + 8AC + 9ABC \approx 9ABC \tag{2.18}$$

2.4 Вывод

На основе теоретических данных, полученных из аналитического раздела, были построены схемы требуемых алгоритмов и проведена теоретическая оценка их трудоемкости.

3 Технологическая часть

В данном разделе приведены требования к программному обеспечению, средства реализации и листинги кода.

3.1 Требования к ПО

К программе предъявляется ряд требований:

- взаимодействие программы и пользователя реализованно в виде меню;
- пользователь может ввести матрицы для умножения ручным способом или сгенерировать, заполнив случайными числами;
- элементами матрицы могут быть только целые числа;
- считается, что пользователь вводит корректные данные;
- на выходе – результат умножения матриц, вычисленный стандартным способом, алгоритмом Винограда и его оптимизированной версии.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран многопоточный язык GO [3]. Данный выбор обусловлен моим желанием расширить свои знания в области применения данного язкыа. Так же данный язык предоставляет средства тестирования разработанного ПО.

3.3 Листинг кода

В листингах 3.1–3.5 приведены реализации вспомогательных утилит, а также алгоритма вычисления произведения двух матриц классическим способом, алгоритма Винограда без оптимизации и с оптимизацией.

Листинг 3.1 – Генератор матрицы

```
func Gen(rows int, columns int) MatrixInt {
    if rows == 0 && columns == 0 {
        fmt.Print("Rows:")
        fmt.Scanln(&rows)
        fmt.Print("Columns:")
        fmt.Scanln(&columns)
    }
    mtr := matrix_init(rows, columns)

    for i := 0; i < mtr.rows; i++ {
        for j := 0; j < columns; j++ {
            mtr.values[i][j] = rand.Intn(100)
        }
    }
    return mtr
}
```

Листинг 3.2 – Функция для выделения памяти

```
func matrix_init(rows int, colunms int) MatrixInt {
    var mtr MatrixInt
    mtr.rows = rows
    mtr.columns = colunms
    mtr.values = make([][]int, mtr.rows)
    for i := range mtr.values {
        mtr.values[i] = make([]int, mtr.columns)
    }
    return mtr
}
```

Листинг 3.3 – Классический алгоритм

```
func Classic_multi(mtr1 MatrixInt, mtr2 MatrixInt) MatrixInt {
    res := matrix_init(mtr1.rows, mtr2.columns)
    for i := 0; i < mtr1.rows; i++ {
        for j := 0; j < mtr2.columns; j++ {
            for k := 0; k < mtr1.columns; k++ {
                res.values[i][j] += mtr1.values[i][k] * mtr2.values[k][j]
            }
        }
    }
    return res
}
```

Листинг 3.4 – Алгоритм Винограда без оптимизации

```
func Winograd_multi(mtr1 MatrixInt, mtr2 MatrixInt) MatrixInt {
    res := matrix_init(mtr1.rows, mtr2.columns)
    wrowcf := calc_wrows(mtr1)
    wcolumncf := calc_wcolumns(mtr2)
    for i := 0; i < res.rows; i++ {
        for j := 0; j < res.columns; j++ {
            res.values[i][j] = -wrowcf[i] - wcolumncf[j]
            for k := 0; k < mtr2.rows/2; k++ {
                res.values[i][j] = res.values[i][j] +
                    (mtr1.values[i][k*2]+mtr2.values[k*2+1][j])*(mtr1.values[i][k*2+1]+
                    mtr2.values[k*2][j])
            }
        }
    }
    if mtr2.rows%2 != 0 {
        for i := 0; i < res.rows; i++ {
            for j := 0; j < res.columns; j++ {
                res.values[i][j] += mtr1.values[i][mtr1.columns-1] *
                    mtr2.values[mtr2.rows-1][j]
            }
        }
    }
    return res
}

func calc_wrows(mtr MatrixInt) []int {
    wrowcf := make([]int, mtr.rows)
    for i := 0; i < mtr.rows; i++ {
        for j := 0; j < mtr.columns/2; j++ {
            wrowcf[i] += mtr.values[i][j*2] * mtr.values[i][j*2+1]
        }
    }
    return wrowcf
}

func calc_wcolumns(mtr MatrixInt) []int {
    wcolumncf := make([]int, mtr.columns)
    for i := 0; i < mtr.columns; i++ {
        for j := 0; j < mtr.rows/2; j++ {
            wcolumncf[i] += mtr.values[j*2][i] * mtr.values[j*2+1][i]
        }
    }
    return wcolumncf
}
```

Листинг 3.5 – Алгоритм Винограда с оптимизацией

```
func Winograd_optimised_multi(mtr1 MatrixInt, mtr2 MatrixInt) MatrixInt {
    res := matrix_init(mtr1.rows, mtr2.columns)
    wrowcf := calc_wrows_optimised(mtr1)
    wcolumncf := calc_wcolumns_optimised(mtr2)
    for i := 0; i < res.rows; i++ {
        for j := 0; j < res.columns; j++ {
            buf := 0
            for k := 0; k < mtr2.rows/2; k++ {
                buf += (mtr1.values[i][k*2] + mtr2.values[k*2+1][j]) *
                    (mtr1.values[i][k*2+1] +
                    mtr2.values[k*2][j])
            }
            res.values[i][j] = buf - wrowcf[i] - wcolumncf[j]
        }
    }
    if mtr2.rows%2 != 0 {
        k := mtr1.columns - 1
        for i := 0; i < res.rows; i++ {
            for j := 0; j < res.columns; j++ {
                res.values[i][j] += mtr1.values[i][k] * mtr2.values[k][j]
            }
        }
    }
    return res
}
func calc_wrows_optimised(mtr MatrixInt) []int {
    wrowcf := make([]int, mtr.rows)
    for i := 0; i < mtr.rows; i++ {
        buf := 0
        for j := 0; j < mtr.columns/2; j++ {
            buf += mtr.values[i][j*2] * mtr.values[i][j*2+1]
        }
        wrowcf[i] = buf
    }
    return wrowcf
}
func calc_wcolumns_optimised(mtr MatrixInt) []int {
    wcolumncf := make([]int, mtr.columns)
    for i := 0; i < mtr.columns; i++ {
        buf := 0
        for j := 0; j < mtr.rows/2; j++ {
            buf += mtr.values[j*2][i] * mtr.values[j*2+1][i]
        }
        wcolumncf[i] = buf
    }
    return wcolumncf
}
```

В таблице 3.1 приведены функциональные тесты для алгоритмов матриц. Все тесты пройдены успешно (таблица 3.2).

Таблица 3.1 – Функциональные тесты

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

Таблица 3.2 – Результат работы программы

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

3.4 Вывод

Были разработаны и протестированы спроектированные алгоритмы: вычисления произведения двух матриц классическим способом, алгоритмом Винограда без оптимизации и с оптимизацией.

4 Исследовательская часть

4.1 Пример работы

На рисунке 4.1 приведен пример работы программы.

```
Menu:
1.Multiply matricies from standart input.
2.Multiply auto-generated matricies.
0.Exit
Choose an option:2
The first matrix:
Rows:3
Columns:7
  81   87   47   59   81   18   25
  40   56    0   94   11   62   89
  28   74   11   45   37    6   95
The second matrix:
Rows:7
Columns:3
  66   28   58
  47   47   87
  88   90   15
  41    8   87
  31   29   56
  37   31   85
  26   13   90
The classic multiplication:
19817 14291 26421
14075 7902 29266
11978 8106 23274
The Winograd multiplication:
19817 14291 26421
14075 7902 29266
11978 8106 23274
The optimised Winograd multiplication:
19817 14291 26421
14075 7902 29266
11978 8106 23274
```

Рисунок 4.1 – Демонстрация работы программы

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система: Windows 10 64-bit [4].
- Память: 16 GB.
- Процессор: AMD Ryzen 5 4600H [5] @ 3.00 GHz

Тестирование проводилось на ноутбуке при включённом режиме производительности. Во время тестирования ноутбук был нагружен только системными процессами.

4.3 Время выполнения алгоритмов

Алгоритмы тестировались при помощи написания «бенчмарков» [6], предоставляемых встроенными в Go средствами. Для такого рода тестирования не нужно самостоятельно указывать количество повторов. В библиотеке для тестирования существует константа N, которая динамически варьируется в зависимости от того, был ли получен стабильный результат или нет.

Листинг 4.1 – Пример реализации бенчмарка

```
func benchmark_multi(size int, b *testing.B, f func(MatrixInt, MatrixInt) MatrixInt) {
    b.StopTimer()
    mtr1 := Gen(size, size)
    mtr2 := Gen(size, size)
    b.StartTimer()

    for i := 0; i < b.N; i++ {
        f(mtr1, mtr2)
    }
}

func BenchmarkCM100(b *testing.B) { benchmark_multi(100, b, Classic_multi) }
```

На рисунках 4.2 и 4.3 приведены графики зависимостей времени работы алгоритмов умножения матриц от их размеров.

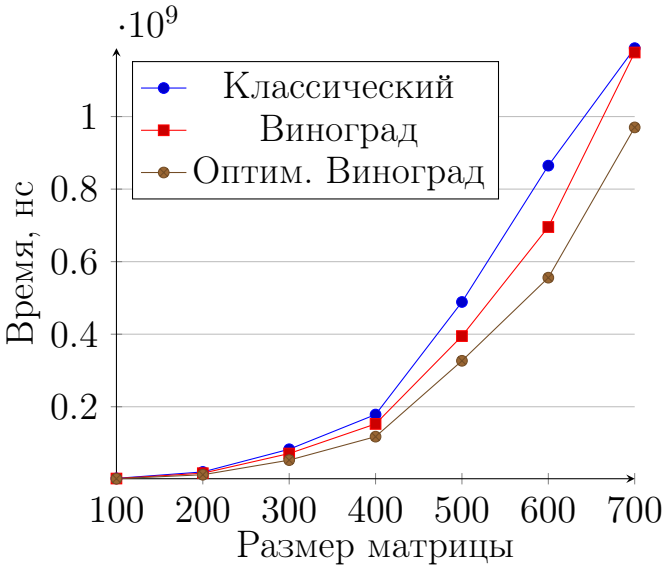

Рисунок 4.2 – Зависимость времени работы алгоритма умножения матриц от размера матрицы. Четные матрицы.

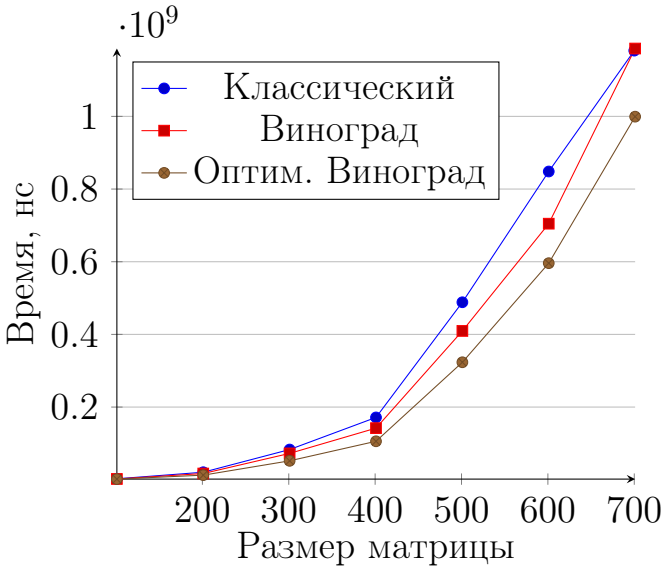

Рисунок 4.3 – Зависимость времени работы алгоритма умножения матриц от размера матрицы. Нечетные матрицы.

4.4 Вывод

Исходя из полученных эксперементальных данных можно сделать несколько выводов.

На матрицах четных размеров классический алгоритм показывает худшие результаты по сравнению с алгоритмом Винограда и его оптимизированной версией (работает дольше в 1.24 и в 1.55 раз соответственно, т.е. около 24% и 55%). Следует отметить, что выполненные оптимизации позволили уменьшить время выполнения алгоритма Винограда в 1.25 раз, т.е. 25%.

На матрицах нечетных размеров классический алгоритм также показывает показатель хуже, чем алгоритм Винограда и его оптимизированная версия. Однако отставание становится меньше, т.к. в последних приходится выполнять один дополнительный цикл. Классический алгоритм работает медленее в 1.20 и в 1.42 раз, т.е. около 20% и 42%.

Заключение

В ходе выполнения работы были выполнены все поставленные задачи и изучены методы динамического программирования на основе алгоритмов умножения матриц.

Экспериментально были подтверждены вычисленные теоритически значения трудоемкости и установлены различия в производительности различных алгоритмов.

На матрицах четных размеров классический алгоритм показывает худшие результаты по сравнению с алгоритмом Винограда и его оптимизированной версией (работает дольше в 1.24 и в 1.55 раз соответственно, т.е. около 24% и 55%). Следует отметить, что выполненные оптимизации позволили уменьшить время выполнения алгоритма Винограда в 1.25 раз, т.е. 25%.

На матрицах нечетных размеров классический алгоритм также показывает показатель хуже, чем алгоритм Винограда и его оптимизированная версия. Однако отставание становится меньше, т.к. в последних приходится выполнять один дополнительный цикл. Классический алгоритм работает медленее в 1.20 и в 1.42 раз, т.е. около 20% и 42%.

Список литературы

[1] Погорелов Д. А. Таразанов А. М. Волкова Л. Л. Оптимизация классического алгоритма Винограда для перемножения матриц // Журнал №1. 2019. Т. 49.

[2] Winograd Fast Convolution [Электронный ресурс]. Режим доступа: `https://medium.com/@dmangla3/understanding-winograd-fast-convolution-a75458744ff` (дата обращения: 21.10.2021).

[3] The Go Programming Language [Электронный ресурс]. Режим доступа: `https://golang.org/` (дата обращения: 01.10.2021).

[4] Explore Windows 10. [Электронный ресурс]. Режим доступа: `https://www.microsoft.com/en-us/windows/` (дата обращения: 02.10.2021).

[5] AMD Processors [Электронный ресурс]. Режим доступа: `https://www.amd.com/en/products/apu/amd-ryzen-5-4600h` (дата обращения: 02.10.2021).

[6] Testing – The Go Programming Language [Электронный ресурс]. Режим доступа: `https://golang.org/pkg/testing/` (дата обращения: 08.10.2021).