



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 №1 по курсу "Анализ алгоритмов"

Тема Алгоритм Кошперсмита-Винограда

Студент Пересторонин П.Г.

Группа ИУ7-53Б

Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва — 2020 г.

# Оглавление

<b>Введение</b>	<b>2</b>
<b>1 Аналитическая часть</b>	<b>3</b>
1.1 Описание алгоритмов . . . . .	3
1.1.1 Стандартный алгоритм . . . . .	3
1.1.2 Алгоритм Копперсмита — Винограда . . . . .	3
<b>2 Конструкторская часть</b>	<b>5</b>
2.1 Разработка алгоритмов . . . . .	5
2.2 Модель вычислений . . . . .	10
2.3 Трудоемкость алгоритмов . . . . .	11
2.3.1 Стандартный алгоритм умножения матриц . . . . .	11
2.3.2 Алгоритм Копперсмита — Винограда . . . . .	12
2.3.3 Оптимизированный алгоритм Копперсмита — Вино- града . . . . .	13
<b>3 Технологическая часть</b>	<b>15</b>
3.1 Требования к ПО . . . . .	15
3.2 Средства реализации . . . . .	15
3.3 Листинг кода . . . . .	15
3.4 Тестирование функций . . . . .	19
<b>4 Исследовательская часть</b>	<b>20</b>
4.1 Технические характеристики . . . . .	20
4.2 Время выполнения алгоритмов . . . . .	20
<b>Заключение</b>	<b>23</b>
<b>Литература</b>	<b>24</b>

# Введение

Алгоритм Копперсмита — Винограда — алгоритм умножения квадратных матриц, предложенный в 1987 году Д. Копперсмитом и Ш. Виноградом [1]. В исходной версии асимптотическая сложность алгоритма составляла  $O(n^{2,3755})$ , где  $n$  — размер стороны матрицы. Алгоритм Копперсмита — Винограда, с учетом серии улучшений и доработок в последующие годы, обладает лучшей асимптотикой среди известных алгоритмов умножения матриц [2].

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 [3]. Поэтому пользуются алгоритмом Штрассена по причинам простоты реализации и меньшей константе в оценке трудоемкости.

Алгоритм Штрассена [4] предназначен для быстрого умножения матриц. Он был разработан Фолькером Штрассеном в 1969 году и является обобщением метода умножения Карацубы на матрицы.

В отличие от традиционного алгоритма умножения матриц, алгоритм Штрассена умножает матрицы за время  $\Theta(n^{\log_2 7}) = O(n^{2,81})$ , что даёт выигрыш на больших плотных матрицах начиная, примерно, от  $64 \times 64$ .

Несмотря на то, что алгоритм Штрассена является асимптотически не самым быстрым из существующих алгоритмов быстрого умножения матриц, он проще программируется и эффективнее при умножении матриц относительно малого размера.

Задачи лабораторной работы:

1. изучение и реализация 3 алгоритмов перемножения матриц: обычный, Копперсмита-Винограда, улучшенный Копперсмита-Винограда;
2. сравнительный анализ трудоёмкости алгоритмов на основе теоретических расчетов и выбранной модели вычислений;
3. сравнительный анализ алгоритмов на основе экспериментальных данных;
4. подготовка отчета по лабораторной работе.

# 1 Аналитическая часть

## 1.1 Описание алгоритмов

### 1.1.1 Стандартный алгоритм

Пусть даны две прямоугольные матрицы

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

тогда матрица  $C$

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

где

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

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

### 1.1.2 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):

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

Несмотря на то, что второе выражение требует вычисления большего количества операций, чем стандартный алгоритм: вместо четырех умножений - шесть, а вместо трех 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 [5].

## Вывод

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

### 2.1 Разработка алгоритмов

На рисунке 2.1 приведена схема стандартного алгоритма умножения матриц.

На рисунках 2.2 и 2.3 представлена схема алгоритма Копперсмита — Винограда и его функций заполнения сумм строк и столбцов.

На рисунках 2.4 и 2.5 представлена схема улучшенного алгоритма Копперсмита — Винограда и его функций заполнения сумм строк и столбцов.

На схеме видно, что для стандартного алгоритма не существует лучшего и худшего случаев как таковых ввиду отсутствия ветвлений.

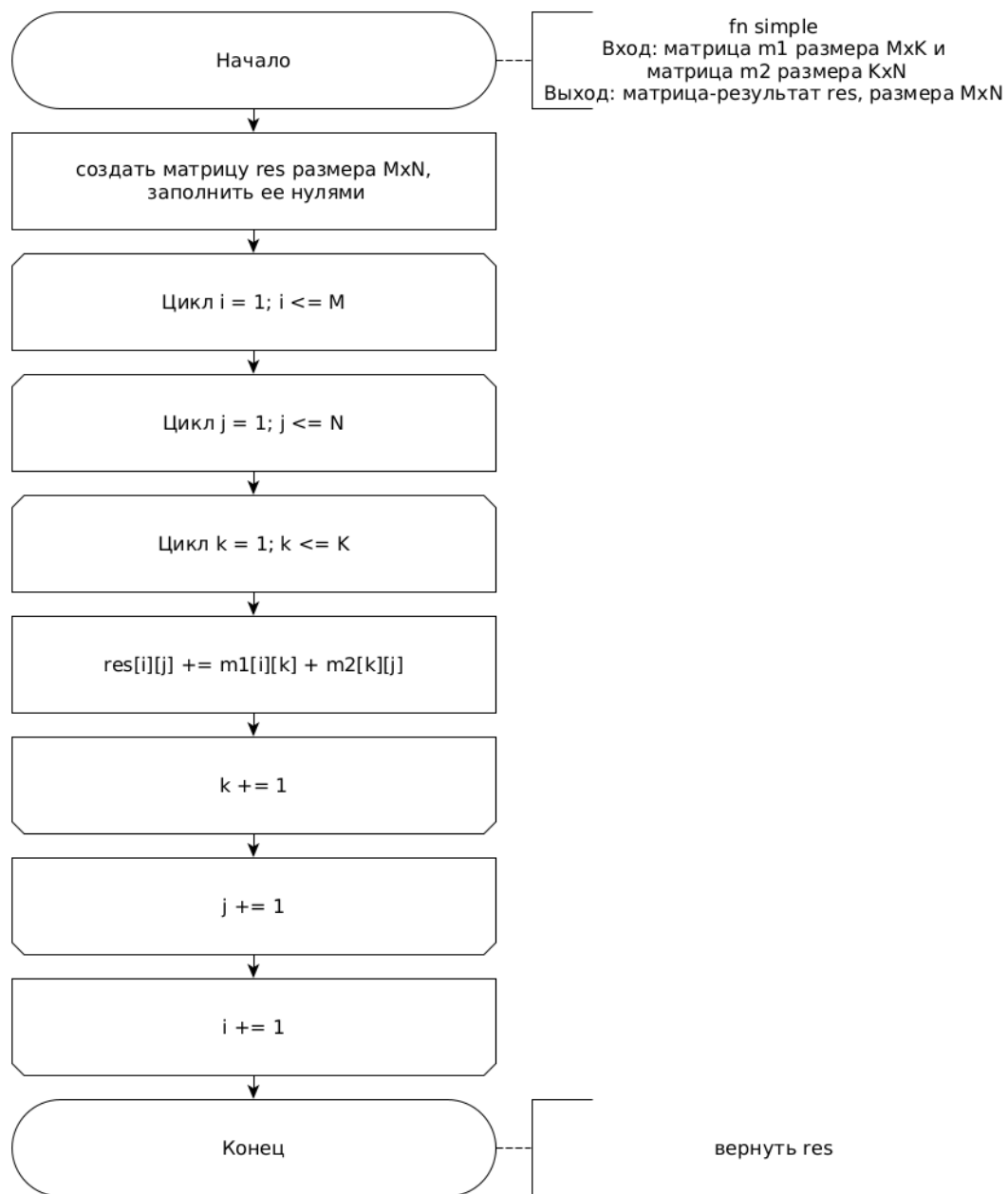


Рис. 2.1: Стандартный алгоритм заполнения матриц

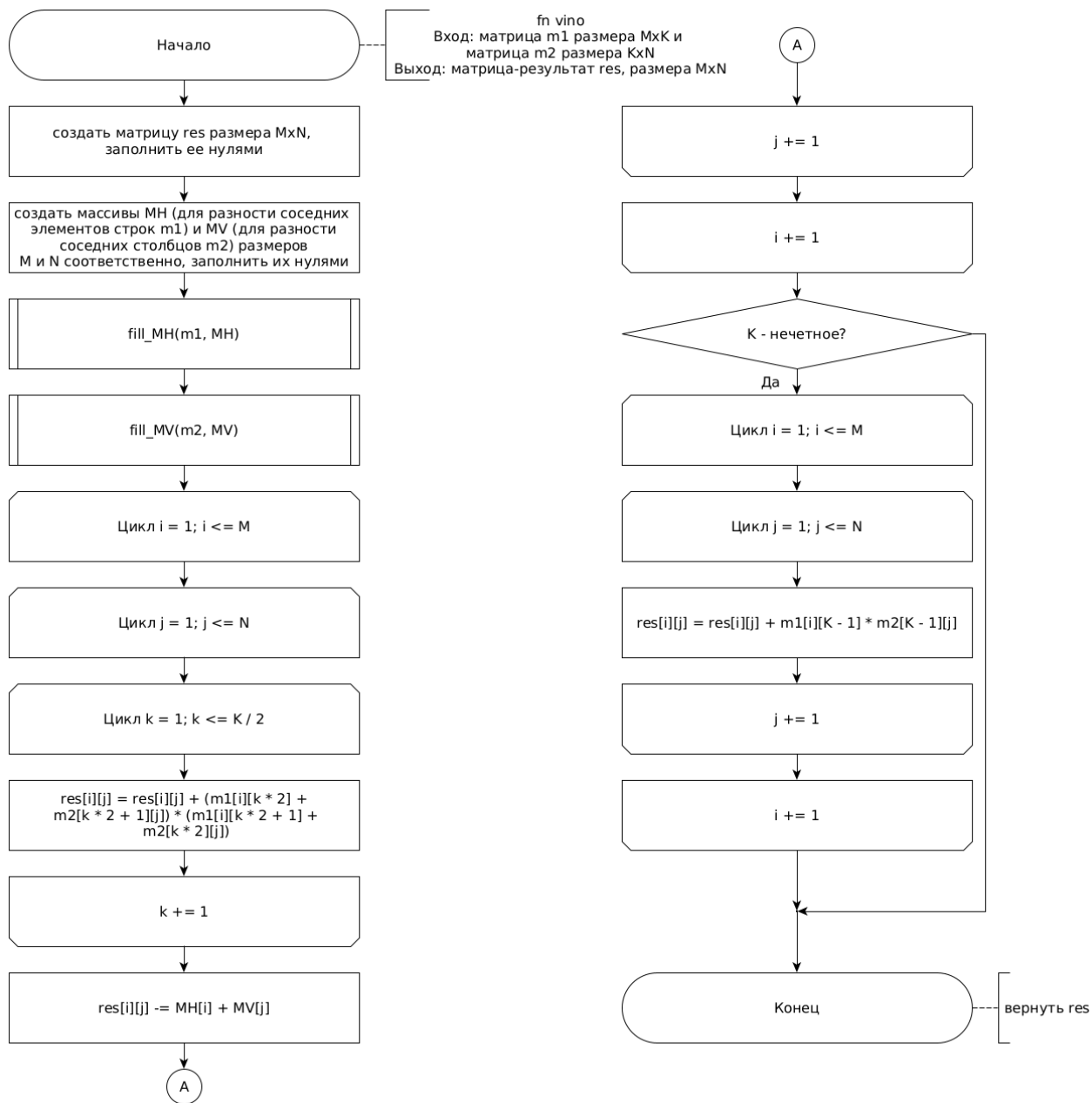


Рис. 2.2: Схема алгоритма Копперсмита — Винограда



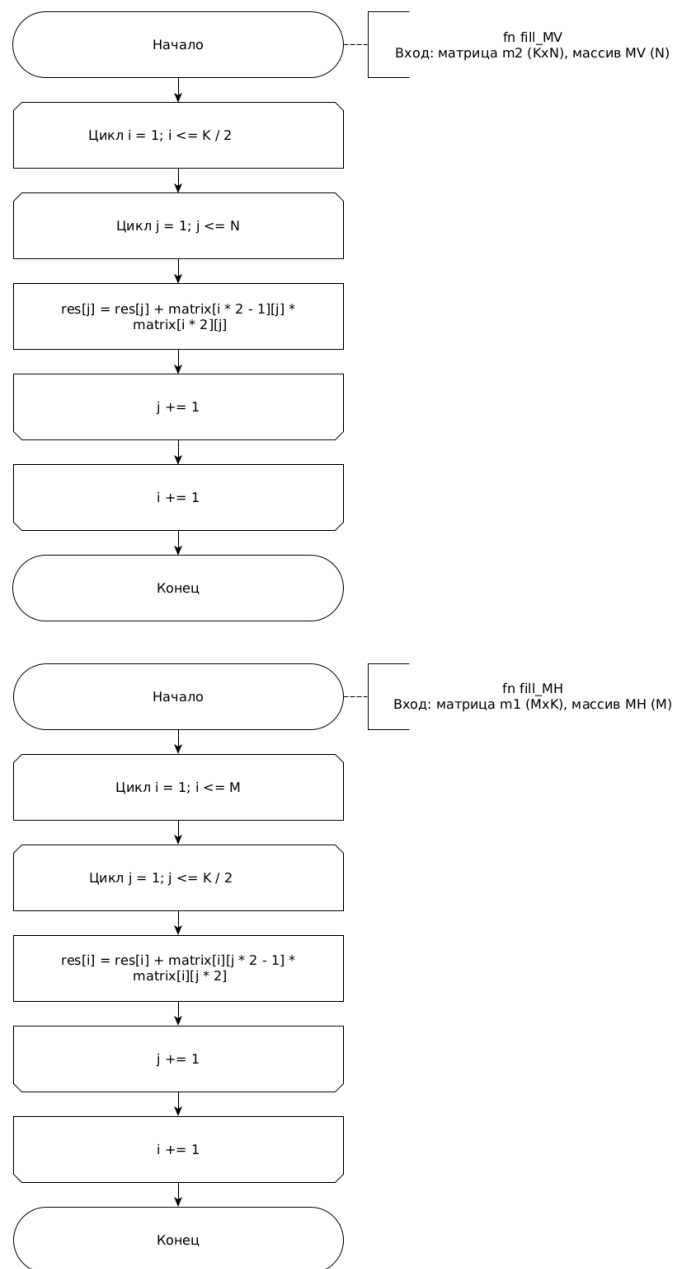


Рис. 2.3: Схемы функций алгоритма Копперсмита — Винограда

Пусть в дальнейшем  $K$  - общий размер при умножении матриц размеров  $M \times K$  и  $K \times N$ .

Видно, что для алгоритма Винограда худшим случаем являются матрицы с нечётным общим размером, а лучшим - с чётным, т. к. отпадает необходимость в последнем цикле.

Данный алгоритм можно оптимизировать:

- заменой операции деления на 2 побитовым сдвигом на 1 вправо;
- заменой выражения вида  $a = a + \dots$  на  $a += \dots$ ;

- сделав в циклах по  $k$  шаг 2, избавившись тем самым от двух операций умножения на каждую итерацию.

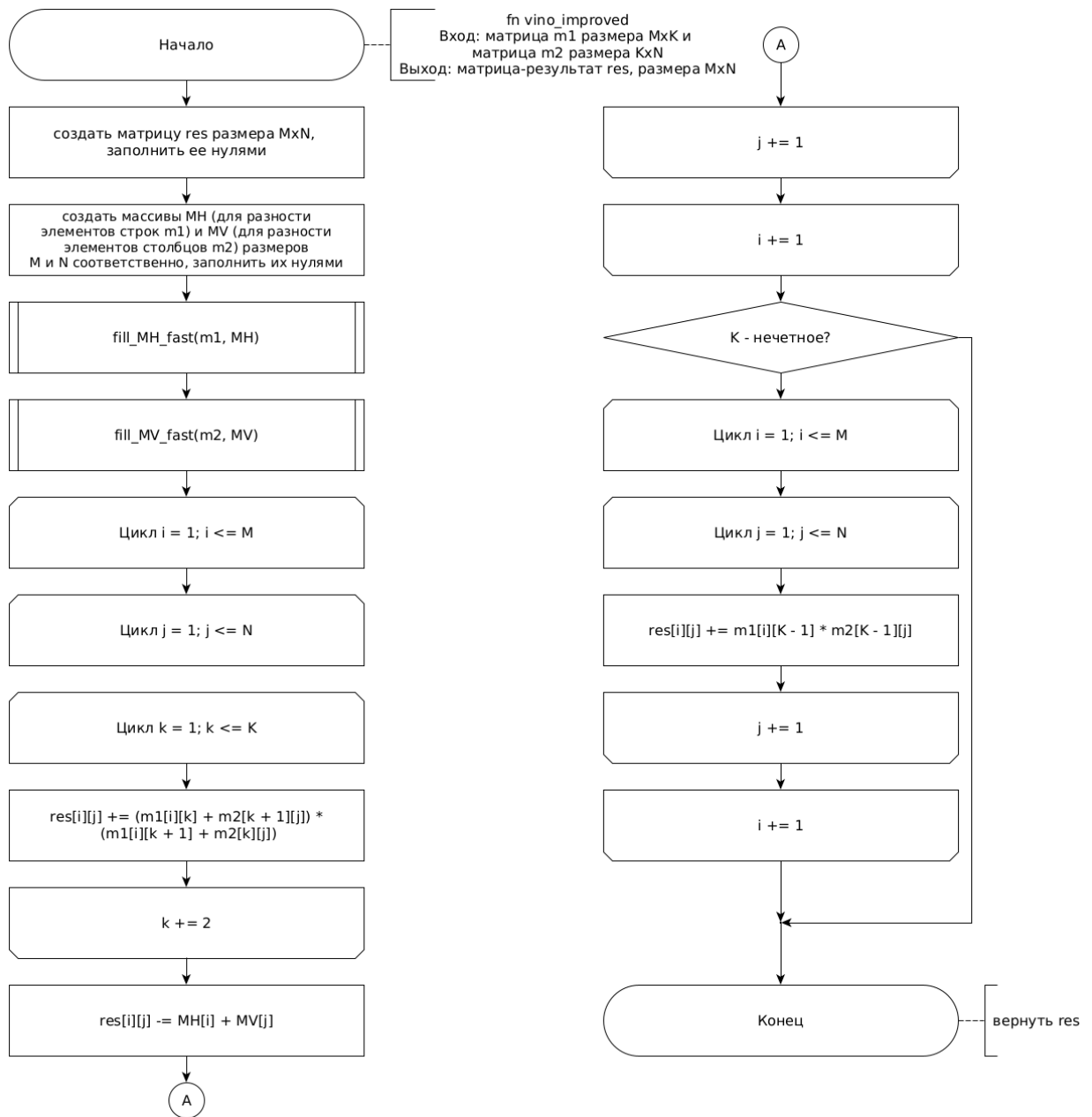


Рис. 2.4: Схема алгоритма улучшенного Копперсмита — Винограда

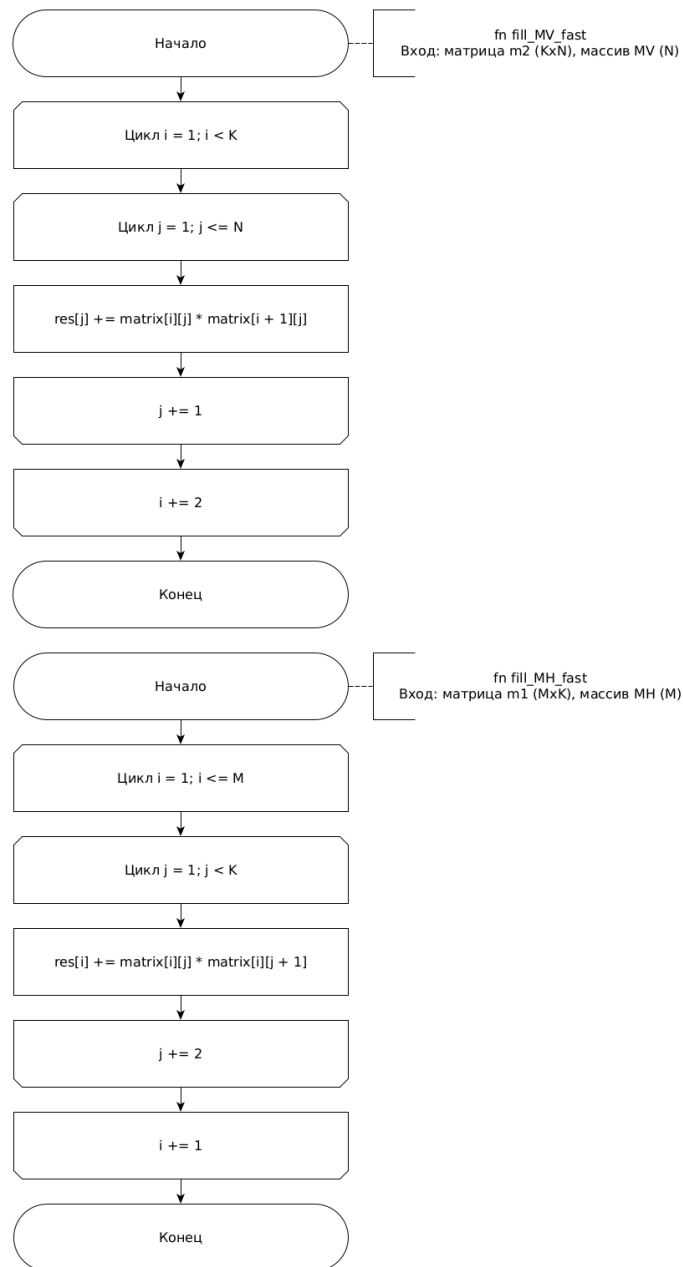


Рис. 2.5: Схемы функций алгоритма улучшенного Копперсмита — Винограда

## 2.2 Модель вычислений

Для последующего вычисления трудоемкости необходимо ввести модель вычислений:

1. операции из списка (2.1) имеют трудоемкость 1;

$$+, -, /, %, ==, !=, <, >, <=, >=, [], ++, -- \quad (2.1)$$

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

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

3. трудоемкость цикла рассчитывается, как (2.3);

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

4. трудоемкость вызова функции равна 0.

## 2.3 Трудоемкость алгоритмов

### 2.3.1 Стандартный алгоритм умножения матриц

Во всех последующих алгоритмах не будем учитывать инициализацию матрицу, в которую записывается результат, потому что данное действие есть во всех алгоритмах и при этом не является самым трудоёмким.

Трудоёмкость стандартного алгоритма умножения матриц состоит из:

- внешнего цикла по  $i \in [1..M]$ , трудоёмкость которого:  $f = 2 + M \cdot (2 + f_{body})$ ;
- цикла по  $j \in [1..N]$ , трудоёмкость которого:  $f = 2 + N \cdot (2 + f_{body})$ ;
- цикла по  $k \in [1..K]$ , трудоёмкость которого:  $f = 2 + 10K$ ;

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

$$f_{standard} = 2 + M \cdot (4 + N \cdot (4 + 10K)) = 2 + 4M + 4MN + 10MNK \approx 10MNK \quad (2.4)$$

### 2.3.2 Алгоритм Копперсмита — Винограда

Трудоёмкость алгоритма Копперсмита — Винограда состоит из:

1. создания и инициализации массивов  $MH$  и  $MV$ , трудоёмкость которого (2.5):

$$f_{init} = M + N; \quad (2.5)$$

2. заполнения массива  $MH$ , трудоёмкость которого (2.6):

$$f_{MH} = 3 + \frac{K}{2} \cdot (5 + 12M); \quad (2.6)$$

3. заполнения массива  $MV$ , трудоёмкость которого (2.7):

$$f_{MV} = 3 + \frac{K}{2} \cdot (5 + 12N); \quad (2.7)$$

4. цикла заполнения для чётных размеров, трудоёмкость которого (2.8):

$$f_{cycle} = 2 + M \cdot (4 + N \cdot (11 + \frac{25}{2} \cdot K)); \quad (2.8)$$

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

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

Итого, для худшего случая (нечётный общий размер матриц) имеем (2.10):

$$f = M + N + 12 + 8M + 5K + 6MK + 6NK + 25MN + \frac{25}{2}MNK \approx 12.5 \cdot MNK \quad (2.10)$$

Для лучшего случая (чётный общий размер матриц) имеем (2.11):

$$f = M + N + 10 + 4M + 5K + 6MK + 6NK + 11MN + \frac{25}{2}MNK \approx 12.5 \cdot MNK \quad (2.11)$$

### 2.3.3 Оптимизированный алгоритм Копперсмита — Винограда

Трудоёмкость улучшенного алгоритма Копперсмита — Винограда состоит из:

1. создания и инициализации массивов  $MH$  и  $MV$ , трудоёмкость которого (2.12):

$$f_{init} = M + N; \quad (2.12)$$

2. заполнения массива  $MH$ , трудоёмкость которого (2.13):

$$f_{MH} = 2 + \frac{K}{2} \cdot (4 + 8M); \quad (2.13)$$

3. заполнения массива  $MV$ , трудоёмкость которого (2.14):

$$f_{MV} = 2 + \frac{K}{2} \cdot (4 + 8N); \quad (2.14)$$

4. цикла заполнения для чётных размеров, трудоёмкость которого (2.15):

$$f_{cycle} = 2 + M \cdot (4 + N \cdot (8 + 9K)) \quad (2.15)$$

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

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

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

$$f = M + N + 10 + 4K + 4KN + 4KM + 8M + 20MN + 9MNK \approx 9MNK \quad (2.17)$$

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

$$f = M + N + 8 + 4K + 4KN + 4KM + 4M + 8MN + 9MNK \approx 9MNK \quad (2.18)$$

## Вывод

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

## 3 Технологическая часть

В данном разделе приведены средства реализации и листинг кода.

### 3.1 Требования к ПО

К программе предъявляется ряд требований:

- на вход подаются размеры 2 матриц, а также их элементы;
- на выходе — матрица, которая является результатом умножения входных матриц.

### 3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран современный компилируемый ЯП Rust [6]. Данный выбор обусловлен моим желанием расширить свои знания в области применения данного языка, а также тем, что данный язык предоставляет широкие возможности для написания тестов [7].

### 3.3 Листинг кода

В листингах 3.1–3.5 приведены реализации алгоритмов умножения матриц, в листингах 3.6–3.8 представлены примеры написания тестов и бенчмарков.

Листинг 3.1: Стандартный алгоритм умножения матриц

```
1 pub fn simple_mult(m1: &[Vec<MatInner>], m2: &[Vec<MatInner>]) -> Vec<Vec<MatInner>> {  
2     let mut matrix = get_result_matrix(m1, m2);  
3  
4     for i in 0..m1.len() {  
5         for j in 0..m2[0].len() {  
6             for k in 0..m2.len() {  
7                 matrix[i][j] += m1[i][k] * m2[k][j];  
8             }  
9         }  
10    }
```



```

9     }
10  }
11
12  matrix
13 }

```

Листинг 3.2: Алгоритм Копперсмита — Винограда

```

1  pub fn vinograd_mult(m1: &[Vec<MatInner>], m2: &[Vec<MatInner>]) -> Vec<Vec<MatInner>> {
2      let mut matrix = get_result_matrix(m1, m2);
3      let precomputed = precompute_values(m1, m2);
4
5      for i in 0..matrix.len() {
6          for j in 0..matrix[0].len() {
7              matrix[i][j] = precomputed.0[i] + precomputed.1[j];
8
9              for k in 0..(m2.len() / 2) {
10                 matrix[i][j] += (m1[i][k * 2] + m2[k * 2 + 1][j]) * (m1[i][k * 2 + 1] +
11                    m2[k * 2][j]);
12             }
13         }
14     }
15
16     if m2.len() % 2 != 0 {
17         for i in 0..matrix.len() {
18             for j in 0..matrix[0].len() {
19                 matrix[i][j] += m1[i][m2.len() - 1] * m2[m2.len() - 1][j];
20             }
21         }
22     }
23
24     matrix
25 }

```

Листинг 3.3: Функции алгоритма Копперсмита — Винограда

```

1  fn precompute_rows(matrix: &[Vec<MatInner>]) -> Vec<MatInner> {
2      let mut res = vec![0; matrix.len()];
3
4      for i in 0..matrix.len() {
5          for j in 0..((matrix[0].len() - 1) / 2) {
6              res[i] = res[i] - matrix[i][j * 2] * matrix[i][j * 2 + 1];
7          }
8      }
9
10     res
11 }
12

```

```

13 fn precompute_cols(matrix: &[Vec<MatInner>]) -> Vec<MatInner> {
14     let mut res = vec![0; matrix[0].len()];
15
16     for i in 0..((matrix.len() - 1) / 2) {
17         for j in 0..matrix[0].len() {
18             res[j] = res[j] - matrix[i * 2][j] * matrix[i * 2 + 1][j];
19         }
20     }
21
22     res
23 }

```

Листинг 3.4: Оптимизированный алгоритм Копперсмита — Винограда

```

1 pub fn vinograd_improved(m1: &[Vec<MatInner>], m2: &[Vec<MatInner>]) ->
2     Vec<Vec<MatInner>> {
3     let mut matrix = get_result_matrix(m1, m2);
4     let precomputed = precompute_values_fast(m1, m2);
5
6     let m = matrix.len();
7     let n = matrix[0].len();
8     let k_iteration = m2.len();
9
10    for i in 0..m {
11        for j in 0..n {
12            matrix[i][j] = precomputed.0[i] + precomputed.1[j];
13            for k in (0..(k_iteration - 1)).step_by(2) {
14                matrix[i][j] += (m1[i][k] + m2[k + 1][j]) * (m1[i][k + 1] + m2[k][j]);
15            }
16        }
17    }
18
19    if m2.len() & 1 != 0 {
20        let max_k = m2.len() - 1;
21
22        for i in 0..matrix.len() {
23            for j in 0..matrix[0].len() {
24                matrix[i][j] += m1[i][max_k] * m2[max_k][j];
25            }
26        }
27    }
28
29    matrix
30 }

```

### Листинг 3.5: Функции оптимизированного алгоритма Копперсмита — Винограда

```
1 fn precompute_rows_fast(matrix: &[Vec<MatInner>]) -> Vec<MatInner> {
2     let mut res = vec![0; matrix.len()];
3
4     for i in 0..matrix.len() {
5         for j in (0..(matrix[0].len() - 1)).step_by(2) {
6             res[i] -= matrix[i][j] * matrix[i][j + 1];
7         }
8     }
9
10    res
11 }
12
13 fn precompute_cols_fast(matrix: &[Vec<MatInner>]) -> Vec<MatInner> {
14     let mut res = vec![0; matrix[0].len()];
15
16     for i in (0..(matrix.len() - 1)).step_by(2) {
17         for j in 0..matrix[0].len() {
18             res[j] -= matrix[i][j] * matrix[i + 1][j];
19         }
20     }
21
22    res
23 }
```

### Листинг 3.6: Пример написания теста для алгоритмов

```
1 #[test]
2 fn check_random() {
3     let matrices = [generate_matrix(S1, S2), generate_matrix(S2, S3)];
4     run_check(&matrices);
5 }
```

### Листинг 3.7: Функция run\_check

```
1 fn run_check(matrices: &[Vec<Vec<MatInner>>]) {
2     let mut results = Vec::new();
3     for i in 0..MULTS_ARRAY.len() {
4         results.push(MULTS_ARRAY[i](&matrices[0], &matrices[1]));
5     }
6
7     for i in 1..MULTS_ARRAY.len() {
8         assert_eq!(results[i], results[0]);
9     }
10 }
```

### Листинг 3.8: Пример написания бенчмарка для алгоритма

```

1 #[bench]
2 fn check_simple1(b: &mut Bencher) {
3     let matrices = [generate_matrix(T1, T1), generate_matrix(T1, T1)];
4     b.iter(|| simple_mult(&matrices[0], &matrices[1]));
5 }

```

## 3.4 Тестирование функций

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

Матрица 1	Матрица 2	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)
$\begin{pmatrix} 1 & -2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} -1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 0 & 4 & 6 \\ 4 & 12 & 18 \\ 4 & 12 & 18 \end{pmatrix}$
(1 2)	(1 2)	Не могут быть перемножены

Таблица 3.1: Тестирование функций

## Вывод

Правильный выбор инструментов разработки позволил эффективно реализовать алгоритмы, настроить модульное тестирование и выполнить исследовательский раздел лабораторной работы.

## 4 Исследовательская часть

### 4.1 Технические характеристики

- Операционная система: Manjaro [8] Linux [9] x86\_64.
- Память: 8 GiB.
- Процессор: Intel® Core™ i7-8550U[10].

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, окружением, а также непосредственно системой тестирования.

### 4.2 Время выполнения алгоритмов

Результаты замеров приведены в таблицах 4.1 и 4.2. На рисунках 4.1 и 4.2 приведены графики зависимостей времени работы алгоритмов от размеров матриц. Здесь и далее: С — стандартный алгоритм, КВ — алгоритм Копперсмита — Винограда, УКВ — улучшенный алгоритм Копперсмита — Винограда.

Размер матрицы	Время, нс		
	С	КВ	УКВ
100	2169144	2424132	1811016
200	17680401	22032658	15485909
300	65838464	76854395	55530661
400	210892129	207347115	162146803
500	364922866	350837776	327348293
600	666673788	655599427	631239232
700	1369979155	1332273841	1207834761
800	3205123980	3108582931	2921219674

Таблица 4.1: Время работы алгоритмов при чётных размерах матриц

Размер матрицы	Время, нс		
	С	КВ	УКВ
101	2259672	2796926	2018691
201	20071583	21688038	18481951
301	74128394	77705821	61183913
401	221489215	218382194	158158291
501	371283268	372238416	361183214
601	672286913	671599427	648239232
701	1382138491	1384273841	1324143862
801	3291482918	3389582931	3217248194

Таблица 4.2: Время работы алгоритмов при нечётных размерах матриц

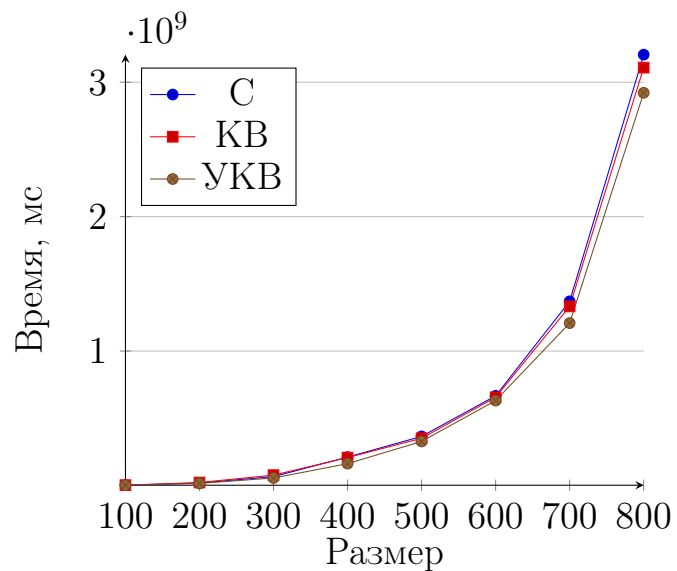


Рис. 4.1: Зависимость времени работы алгоритма от чётного размера квадратной матрицы

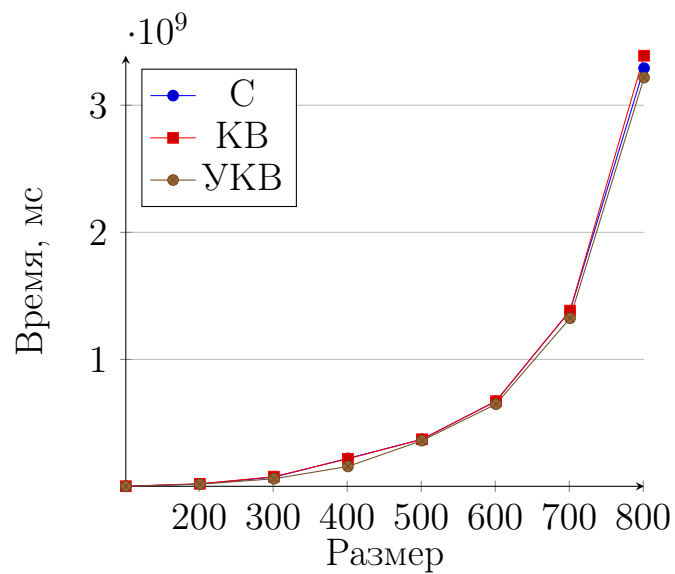


Рис. 4.2: Зависимость времени работы алгоритма от нечётного размера квадратной матрицы

## Вывод

Время работы алгоритма Винограда незначительно меньше стандартного алгоритма умножения, однако при размере  $800 \times 800$  время вычислений алгоритма Винограда на 0.3 секунды меньше, нежели у стандартного алгоритма (что составляет в данном случае примерно 10%).

# Заключение

В рамках данной лабораторной работы:

1. были изучены и реализованы 3 алгоритма перемножения матриц: обычный, Копперсмита-Винограда, улучшенный Копперсмита-Винограда;
2. был произведен анализ трудоёмкости алгоритмов на основе теоретических расчетов и выбранной модели вычислений;
3. был сделан сравнительный анализ алгоритмов на основе экспериментальных данных;
4. был подготовлен отчет по проделанной работе.

На основании анализа трудоёмкости алгоритмов в выбранной модели вычислений было показано, что улучшенный алгоритм Винограда имеет меньшую сложность, нежели простой алгоритм перемножения матриц, что при размерах матриц  $800 \times 800$  дало практический выигрыш во времени в 10%.



# Литература

- [1] Coppersmith D., Winograd S. Matrix multiplication via arithmetic progressions // Journal of Symbolic Computation. 1990. no. 9. P. 251–280.
- [2] Group-theoretic Algorithms for Matrix Multiplication / H. Cohn, R. Kleinberg, B. Szegedy et al. // Proceedings of the 46th Annual Symposium on Foundations of Computer Science. 2005. October. P. 379–388.
- [3] Robinson S. Toward an Optimal Algorithm for Matrix Multiplication // SIAM News. 2005. November. Vol. 38, no. 9.
- [4] Strassen V. Gaussian Elimination is not Optimal // Numerische Mathematik. 2005. Vol. 13, no. 9. P. 354–356.
- [5] Погорелов Дмитрий Александрович Таразанов Артемий Михайлович Волкова Лилия Леонидовна. Оптимизация классического алгоритма Винограда для перемножения матриц // Журнал №1. 2019. Т. 49.
- [6] Rust Programming Language [Электронный ресурс]. URL: <https://doc.rust-lang.org/std/index.html>. 2017.
- [7] Документация по ЯП Rust: бенчмарки [Электронный ресурс]. Режим доступа: <https://doc.rust-lang.org/1.7.0/book/benchmark-tests.html> (дата обращения: 21.09.2020).
- [8] Manjaro – enjoy the simplicity [Электронный ресурс]. Режим доступа: <https://manjaro.org/> (дата обращения: 21.09.2020).
- [9] Русская информация об ОС Linux [Электронный ресурс]. Режим доступа: <https://www.linux.org.ru/> (дата обращения: 21.09.2020).
- [10] Процессор Intel® Core™ i7-8550U [Электронный ресурс]. Режим доступа: <https://ark.intel.com/content/www/ru/ru/ark/products/122589/intel-core-i7-8550u-processor-8m-cache-up-to-4-00-ghz.html> (дата обращения: 21.09.2020).