

МГТУ им. БАУМАНА

ЛАБОРАТОРНАЯ РАБОТА №4

По курсу: "АНАЛИЗ АЛГОРИТМОВ"

## **Параллельное умножение матриц**

Работу выполнил: Гаврилов Дмитрий, ИУ7-56Б

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

*Москва, 2019*

# Оглавление

<b>Введение</b>	<b>2</b>
<b>1 Аналитическая часть</b>	<b>3</b>
1.1 Алгоритм Винограда . . . . .	3
1.2 Параллельный алгоритм Винограда . . . . .	4
1.3 Распараллеливание задачи . . . . .	5
1.3.1 Вывод . . . . .	5
<b>2 Конструкторская часть</b>	<b>6</b>
2.1 Схемы алгоритмов . . . . .	6
<b>3 Технологическая часть</b>	<b>12</b>
3.1 Выбор ЯП . . . . .	12
3.2 Реализация алгоритма . . . . .	12
<b>4 Исследовательская часть</b>	<b>18</b>
4.1 Постановка эксперимента . . . . .	18
4.2 Сравнительный анализ на основе замеров времени работы алгоритмов . . . . .	18
4.3 Вывод . . . . .	20
<b>Заключение</b>	<b>21</b>

# Введение

Цель работы: изучение алгоритмов умножения матриц. В данной лабораторной работе рассматривается стандартный алгоритм умножения матриц, алгоритм Винограда и модифицированный алгоритм Винограда. Также требуется изучить расчет сложности алгоритмов, получить навыки в улучшении алгоритмов. Эти алгоритмы активно применяются во всех областях, применяющих линейную алгебру, таких как:

- компьютерная графика
- физика
- экономика

В ходе лабораторной работы предстоит:

- Изучение и реализация параллельного алгоритма Винограда для умножения матриц
- Сравнить зависимость времени работы алгоритма от числа параллельных потоков исполнения и размера матриц
- Провести сравнение стандартного и параллельного алгоритма.

# 1 | Аналитическая часть

Матрицей  $A$  размера  $[m * n]$  называется прямоугольная таблица чисел, функций или алгебраических выражений, содержащая  $m$  строк и  $n$  столбцов. Числа  $m$  и  $n$  определяют размер матрицы. [?] Если число столбцов в первой матрице совпадает с числом строк во второй, то эти две матрицы можно перемножить. У произведения будет столько же строк, сколько в первой матрице, и столько же столбцов, сколько во второй.

Пусть даны две прямоугольные матрицы  $A$  и  $B$  размеров  $[m * n]$  и  $[n * k]$  соответственно. В результате произведения матриц  $A$  и  $B$  получим матрицу  $C$  размера  $[m * k]$ .

$C_{i,j} = \sum_{r=1}^n A_{i,r} \cdot B_{r,j}$  называется произведением матриц  $A$  и  $B$  [?].

## 1.1 Алгоритм Винограда

Подход Алгоритма Винограда является иллюстрацией общей методологии, начатой в 1979-х годах на основе билинейных и трилинейных форм, благодаря которым большинство усовершенствований для умножения матриц были получены [?].

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 (1.1)

$$V \cdot W = v_1 \cdot w_1 + v_2 \cdot w_2 + v_3 \cdot w_3 + v_4 \cdot w_4 \quad (1.1)$$

Равенство (1.1) можно переписать в виде (1.2)

$$V \cdot W = (v_1 + w_2) \cdot (v_2 + w_1) + (v_3 + w_4) \cdot (v_4 + w_3) - v_1 \cdot v_2 - v_3 \cdot v_4 - w_1 \cdot w_2 - w_3 \cdot w_4 \quad (1.2)$$

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. Это означает, что над предварительно обработанными элементами нам придется выполнять лишь первые два умножения и последующие пять сложений, а также дополнительно два сложения.

В алгоритме Винограда, результатом умножение матриц  $A$  размером  $[m * n]$  и  $B$  размером  $[n * k]$  получим матрицу  $C$  размера  $[m * k]$  которая вычисляется по следующим формулам:

Вычисление строковых множителей (1.3)

$$Rows_i = \sum_{j=1}^{n/2} A_{i,2j} \cdot A_{i,2j+1} \quad (1.3)$$

Вычисление множителей столбцов (1.4)

$$Cols_i = \sum_{j=1}^{n/2} B_{2j,i} \cdot B_{2j+1,i} \quad (1.4)$$

Вычисление результирующей матрицы  $C$  (1.5)

$$C_{i,j} = -Rows_i - Cols_j + \sum_{k=1}^{n/2} (A_{i,2k+1} + B_{2k,j}) \cdot (A_{i,2k} + B_{2k+1,j}) \quad (1.5)$$

Следует отметить, что в худшем случае (при условии того что кол-во столбцов в матрице  $A$  либо кол-во строк в матрице  $B$  - нечётное) к результирующей формуле (1.5) применяется следующее выражение (1.6)

$$C_{i,j}+ = A_{i,n} \cdot B_{n,j} \quad (1.6)$$

## 1.2 Параллельный алгоритм Винограда

Трудоёмкость алгоритма Винограда имеет сложность  $O(nmk)$  для умножения матриц  $n1 \times m1$  на  $n2 \times m2$ . Чтобы улучшить алгоритм, следует распараллелить ту часть алгоритма, которая содержит 3 вложенных цикла.

Вычисление результата для каждой строки не зависит от результата выполнения умножения для других строк. Поэтому можно распараллелить часть кода, где происходят эти действия. Каждый поток будет выполнять вычисления определенных строк результирующей матрицы.

## 1.3 Распараллеливание задачи

В рамках данной лабораторной работы производилось распараллеливание задачи по потокам. В CPU для данной цели используются tread.

CPU – central processing unit – это универсальный процессор, также именуемый процессором общего назначения. Он оптимизирован для достижения высокой производительности единственного потока команд. Доступ к памяти с данными и инструкциями происходит преимущественно случайным образом. Для того, чтобы повысить производительность CPU еще больше, они проектируются специально таким образом, чтобы выполнять как можно больше инструкций параллельно. Например для этого в ядрах процессора используется блок внеочередного выполнения команд. Но несмотря на это, CPU все равно не в состоянии осуществить параллельное выполнение большого числа инструкций, так как расходы на распараллеливание инструкций внутри ядра оказываются очень существенными. Именно поэтому процессоры общего назначения имеют не очень большое количество исполнительных блоков. Реальную многопоточность можно обеспечить, если количество созданных потоков в программе будет равным количеству логических ядер процессора. В рамках данной работы будет выполнено распараллеливание задачи для процессора который имеет 4 логических ядра.

### 1.3.1 Вывод

Были рассмотрены стандартная и параллельная реализации алгоритма Винограда.

## 2 | Конструкторская часть

### **Требования к вводу:**

На вход подаются две матрицы и их размерности

### **Требования к программе:**

Корректное умножение двух матриц

## 2.1 Схемы алгоритмов

Далее были представлены схемы алгоритмов.

На рис. 2.1-2.2 представлена схема стандартного алгоритма Винограда:

На рис. 2.3-2.4 представлена схема распараллеленного алгоритма Винограда:

На рис. 2.5 представлена схема функции, которая выполняет параллельное вычисление матрицы:

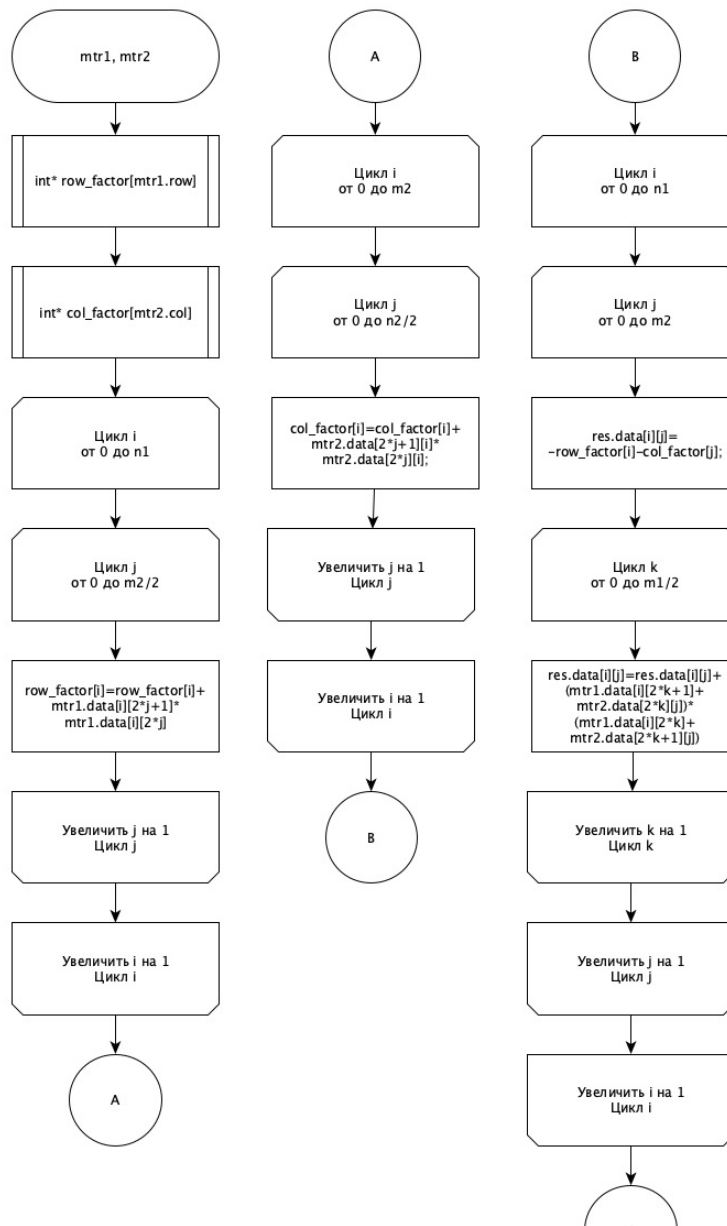


Рис. 2.1: Схема стандартного алгоритма Винограда



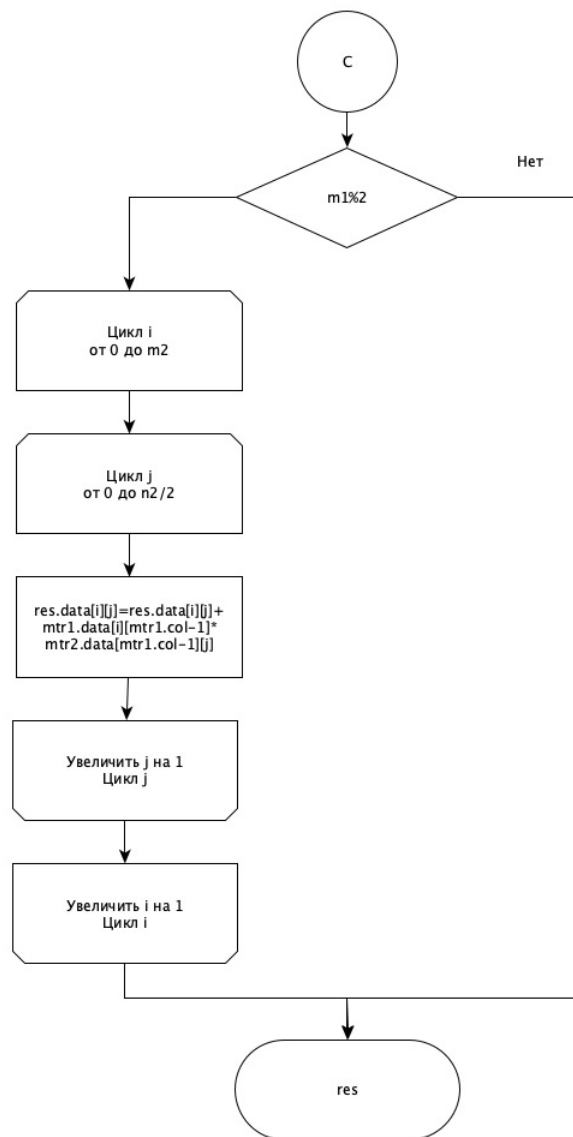


Рис. 2.2: Схема стандартного алгоритма Винограда (продолжение)

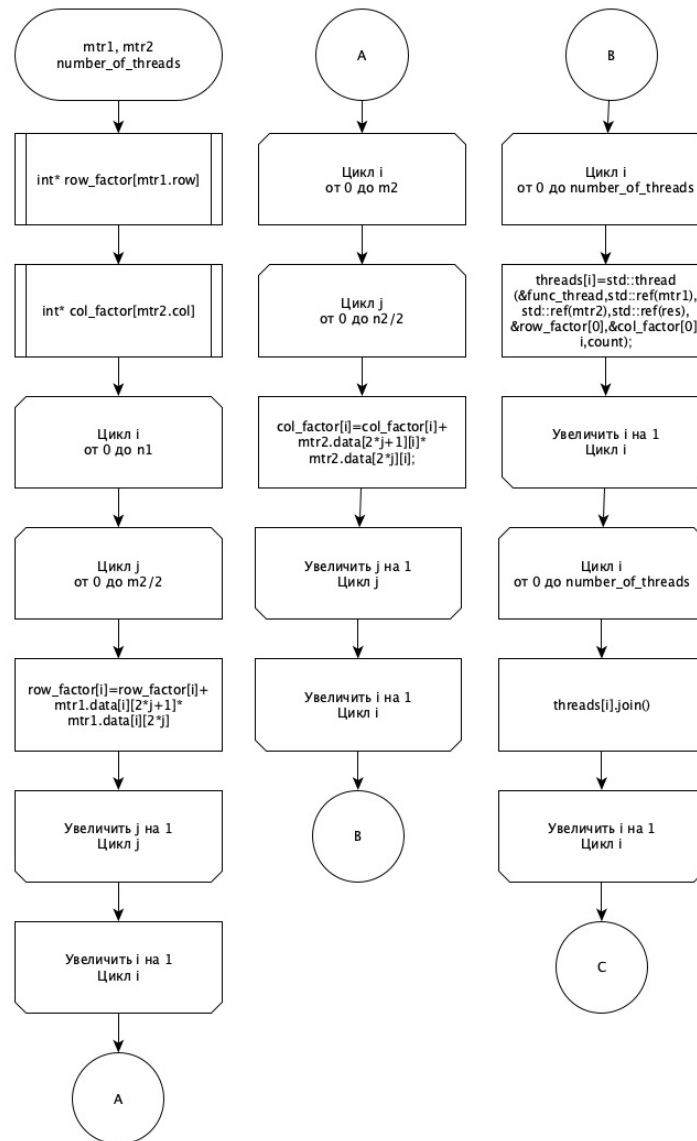


Рис. 2.3: Схема распараллеленного алгоритма Винограда

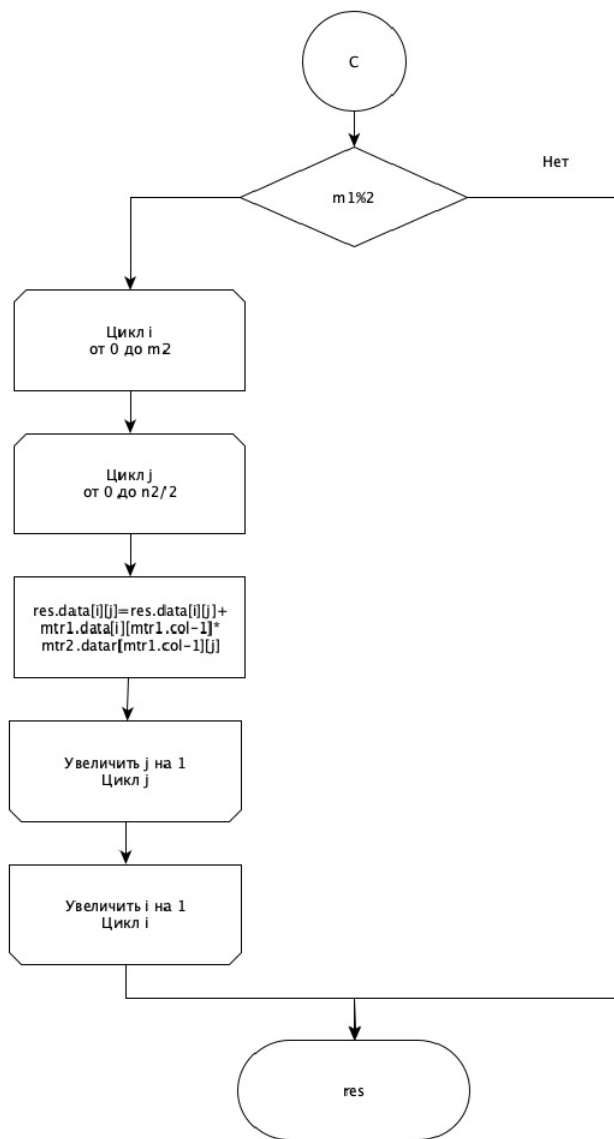


Рис. 2.4: Схема распараллеленного алгоритма Винограда (продолжение)

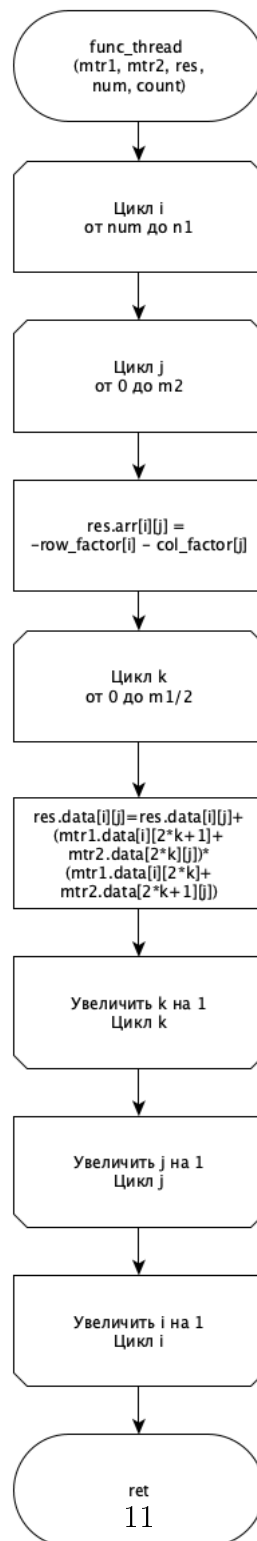


Рис. 2.5: Схема функции, которая выполняет параллельное вычисление матрицы

## 3 | Технологическая часть

### 3.1 Выбор ЯП

Для реализации программ я выбрал язык программирования Java, так имею большой опыт работы с ним. Среда разработки - IntelliJ IDEA.

Для работы с потоками используется библиотека поддержки потоков `java.util.concurrent`.

Для замера времени используется системный вызов `System.currentTimeMillis`, возвращающий количество миллисекунд.

### 3.2 Реализация алгоритма

В листинге 3.1 будет рассмотрена реализация стандартного алгоритма Винограда

Листинг 3.1: Стандартный алгоритм Винограда

```
1 public class MultiplyVinograd implements MatrixMultiplier {  
2     @Override  
3     public int [][] multiply(int [][] firstMatrix, int [][]  
4         secondMatrix) {  
5         int firstRowCount = firstMatrix.length;  
6         int secondRowCount = secondMatrix.length;  
7  
8         if (firstRowCount == 0 || secondRowCount == 0)  
9             return null;  
10  
11         int firstColumnCount = firstMatrix[0].length;
```

```

11      int secondColumnCount = secondMatrix[0].length;
12
13      if (firstColumnCount != secondRowCount)
14          return null;
15
16      int[] rowFactors = new int[firstRowCount];
17      int[] columnFactors = new int[secondColumnCount];
18
19      int[][] result = new int[firstRowCount][
20          secondColumnCount];
21
22      for (int i = 0; i < firstRowCount; i++) {
23          for (int j = 0; j < firstColumnCount / 2; j++) {
24              {
25                  rowFactors[i] += firstMatrix[i][j * 2] *
26                      firstMatrix[i][j * 2 + 1];
27              }
28          }
29
30      for (int i = 0; i < secondColumnCount; i++) {
31          for (int j = 0; j < secondRowCount / 2; j++) {
32              columnFactors[i] += secondMatrix[j * 2][i]
33                  * secondMatrix[j * 2 + 1][i];
34          }
35      }
36
37      for (int i = 0; i < firstRowCount; i++) {
38          for (int j = 0; j < secondColumnCount; j++) {
39              result[i][j] = -rowFactors[i] -
40                  columnFactors[j];
41              for (int k = 0; k < firstColumnCount / 2; k
42                  ++){
43                  result[i][j] += (firstMatrix[i][2 * k +
44                      1] + secondMatrix[2 * k][j]) * (
45                      firstMatrix[i][2 * k] + secondMatrix
46                      [2 * k + 1][j]);
47              }
48          }
49      }
50
51      }

```

```

42         if (firstColumnCount % 2 == 1) {
43             for (int i = 0; i < firstRowCount; i++) {
44                 for (int j = 0; j < secondColumnCount; j++)
45                     {
46                         result[i][j] += firstMatrix[i][
47                             firstColumnCount - 1] * secondMatrix
48                             [firstColumnCount - 1][j];
49                     }
50             }
51         }
52     }

```

Можно заметить, что вычисление результата для каждой строки происходит независимо от результата выполнения умножения для других строк. Поэтому возможно распараллелить участок кода, соответствующий строкам 33-40 листинга 3.1. Каждый поток будет выполнять вычисление некоторых строк результирующей матрицы. Это сделано потому, что проход по строкам матрицы является более эффективным с точки зрения организации данных в памяти.

В листинге 3.2 представлен параллельный алгоритм Винограда:

Листинг 3.2: Параллельный алгоритм Винограда

```

1 public class MultiplyVinogradParallel{
2     public int [][] multiplyParallel(int [][] firstMatrix ,
3         int [][] secondMatrix , int threadsCount) throws
4         InterruptedException {
5         int firstRowCount = firstMatrix.length;
6         int secondRowCount = secondMatrix.length;
7
8         if (firstRowCount == 0 || secondRowCount == 0) {
9             return null;
10        }
11
12        int firstColumnCount = firstMatrix[0].length;
13        int secondColumnCount = secondMatrix[0].length;

```

```

13     if (firstColumnCount != secondRowCount) {
14         return null;
15     }
16
17     int[] rowFactors = new int[firstRowCount];
18     int[] columnFactors = new int[secondColumnCount];
19
20     int[][] result = new int[firstRowCount][
21         secondColumnCount];
22
23     for (int i = 0; i < firstRowCount; i++) {
24         for (int j = 0; j < firstColumnCount / 2; j++) {
25             {
26                 rowFactors[i] += firstMatrix[i][j * 2] *
27                     firstMatrix[i][j * 2 + 1];
28             }
29         }
30
31     for (int i = 0; i < secondColumnCount; i++) {
32         for (int j = 0; j < secondRowCount / 2; j++) {
33             columnFactors[i] += secondMatrix[j * 2][i]
34                 * secondMatrix[j * 2 + 1][i];
35         }
36     }
37
38     ExecutorService service = Executors.
39         newFixedThreadPool(threadsCount);
40
41     for (int i = 0; i < threadsCount; i++) {
42         service.execute(new LoopExecutor(
43             firstMatrix,
44             secondMatrix,
45             result,
46             rowFactors,
47             columnFactors,
48             i,
49             threadsCount));
50     }
51     service.shutdown();

```



```

48
49     while (!service.awaitTermination(24L, TimeUnit.
50         HOURS)) {
51         System.out.println("Not yet. Still waiting for
52             termination");
53     }
54
55     if (firstColumnCount % 2 == 1) {
56         for (int i = 0; i < firstRowCount; i++) {
57             for (int j = 0; j < secondColumnCount; j++)
58                 {
59                     result[i][j] += firstMatrix[i][
60                         firstColumnCount - 1] * secondMatrix
61                         [firstColumnCount - 1][j];
62                 }
63         }
64     }

```

В листинге 3.3 представлено распараллеленное вычисление тройного цикла:

Листинг 3.3: Распараллеленный тройной цикл

```

1  class LoopExecutor implements Runnable {
2      private int [][] firstMatrix;
3      private int [][] secondMatrix;
4      private int [][] result;
5      private int [] rowFactors;
6      private int [] colFactors;
7      private int startIndex;
8      private int step;
9
10     public LoopExecutor(int [][] firstMatrix, int [][]
        secondMatrix, int [][] result, int [] rowFactors,
        int [] colFactors, int startIndex, int step) {
11         this.firstMatrix = firstMatrix;
12         this.secondMatrix = secondMatrix;

```

```

13         this.result = result;
14         this.rowFactors = rowFactors;
15         this.colFactors = colFactors;
16         this.startIndex = startIndex;
17         this.step = step;
18     }
19
20     @Override
21     public void run() {
22         for (int i = startIndex; i < firstMatrix.length
23             ; i+=step) {
24             for (int j = 0; j < secondMatrix[0].length;
25                 j++) {
26                 result[i][j] = -rowFactors[i] -
27                     colFactors[j];
28                 for (int k = 0; k < firstMatrix[0].
29                     length / 2; k++) {
30                     result[i][j] += (firstMatrix[i][2 *
31                         k + 1] + secondMatrix[2 * k][j]
32                         ) * (firstMatrix[i][2 * k] +
33                         secondMatrix[2 * k + 1][j]);
34                 }
35             }
36         }
37     }

```

## 4 | Исследовательская часть

### 4.1 Постановка эксперимента

Были проведены исследования зависимости времени работы трех алгоритмов от размеров перемножаемых матриц и количества использованных потоков. Замеры времени проводились для матриц четной размерности размером от 100 до 1000 с шагом 100 и матриц нечетной размерности размером от 101 до 1001 с шагом 100. Количество потоков - от 1 до 16, т.к. компьютер, на котором проводились вычисления, содержит 4 логических ядра.

Временные замеры проводятся путём многократного проведения эксперимента и деления результирующего времени на количество итераций эксперимента.

### 4.2 Сравнительный анализ на основе замеров времени работы алгоритмов

На рис. 4.1 представлен график времени работы алгоритма на матрицах четной размерности:

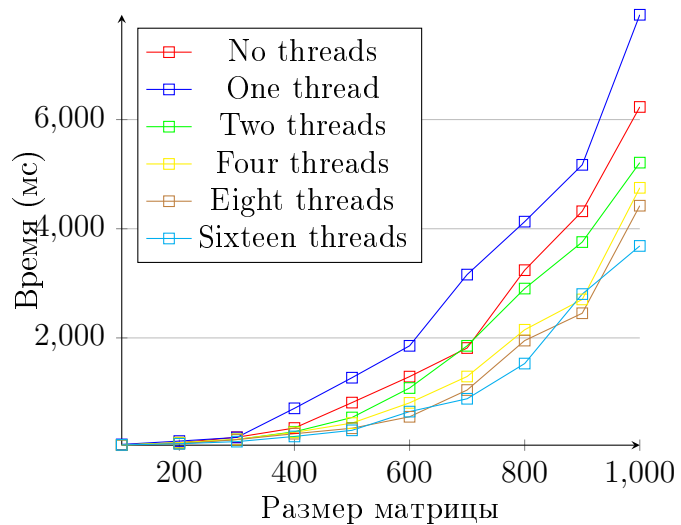


Рис. 4.1: Сравнение времени на матрицах чётной размерности

На рис. 4.2 представлен график времени работы алгоритма на матрицах четной размерности:

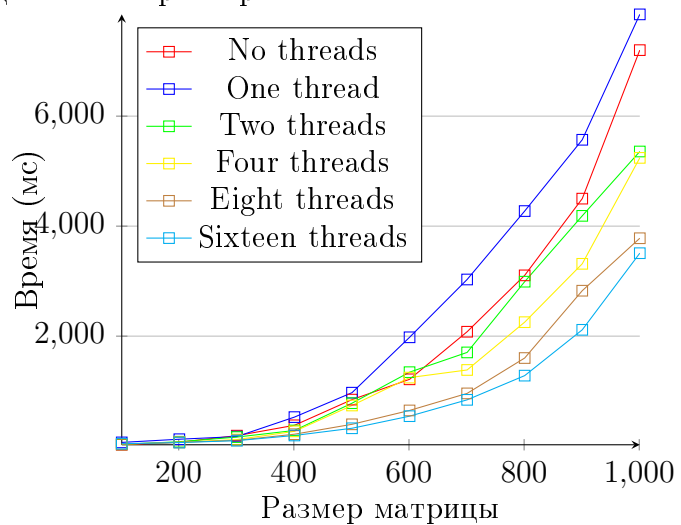


Рис. 4.2: Сравнение времени на матрицах нечётной размерности

### 4.3 Вывод

Эксперименты замера времени показали, что при последовательной и параллельной (с одним рабочим потоком) реализациях оптимизированного алгоритма Винограда совсем немного выигрывает последовательная реализация (в ней не тратится время на выделение рабочего потока).

При сравнении замеров времени для параллельной реализации алгоритма с 1, 2, 4, 8 и 16 рабочими потоками выяснилось, что максимальная производительность достигается на 8-ми и 16-и рабочих потоках. Выполнение алгоритма на 8-ми рабочих потоках быстрее в 3,82 раз, по сравнению с выполнением на 1-м потоке для матриц размера  $1000 \times 1000$ .

## Заключение

В ходе работы был изучен параллельный алгоритм умножения матриц: алгоритм Винограда. Выполнено сравнение зависимости всех рассматриваемых алгоритмов от числа параллельных потоков и размера матриц. В ходе исследования было установлено, что многопоточный алгоритм Винограда выполняется быстрее, чем стандартный алгоритм.