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

Название: Алгоритмы умножения матриц

Дисциплина: Анализ алгоритмов

Студент ИУ7-55Б _____ В.Г. Горячев
(Группа) (Подпись, дата) (И.О. Фамилия)

Преподаватель _____ Л.Л. Волкова
(Подпись, дата) (И.О. Фамилия)

Москва, 2020

Оглавление

Введение

Умножение матриц – одна из самых широко используемых операций в вычислениях, например, в таких областях, как физические расчёты и компьютерная графика. И эта распространённость, при достаточной сложности самого процесса, вынудила программистов и математиков искать способы быстрого умножения матриц. Одним из таких является алгоритм Ш. Винограда, который и будет изучаться в настоящей лабораторной работе.

1 Аналитическая часть

1.1 Цель и задачи

Целью данной лабораторной работы является реализация и сравнение по временной эффективности алгоритма стандартного умножения матриц, на основе математического определения, классического алгоритма Винограда и самостоятельно оптимизированной его версии. Также требуется изучить расчёт сложности алгоритмов. Задачи:

1) реализовать алгоритмы умножения матриц;
2) оптимизировать алгоритм Винограда;
3) дать теоретическую оценку трудоёмкости стандартного алгоритма умножения матриц, алгоритма Винограда и модифицированного алгоритма Винограда;
4) провести сравнительный анализ реализаций по затраченному времени.

1.2 Описание и формулы

Матрица — математический объект, записываемый в виде прямоугольной таблицы элементов кольца или поля (например, целых, действительных или комплексных чисел), которая представляет собой совокупность строк и столбцов, на пересечении которых находятся её элементы. Количество строк и столбцов задает размер матрицы. Хотя исторически рассматривались, например, треугольные

матрицы, в настоящее время говорят исключительно о матрицах прямоугольной формы, так как они являются наиболее удобными и общими.

1.2.1 Стандартное умножение матриц

В математике произведение матриц определяется формулой (1.1). Она же лежит в основе стандартного алгоритма умножения матриц. Пусть даны две прямоугольные матрицы A и B размером $[n * m]$ и $[m * q]$. В результате произведения матриц A и B получим матрицу C размером $[n * q]$, в которой:

$$c_{i,j} = \sum_{k=1}^{m} a_{ik} b_{kj} \qquad (i = 1, 2, ...n; j = 1, 2, ...q) \tag{1.1}$$

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

1.2.2 Алгоритм 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[2].

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 вычисляется по формуле (1.2):

$$V * W = v_1 w_1 + v_2 w_2 + v_3 w_3 + v_4 w_4 \tag{1.2}$$

Это равенство можно переписать, и тогда выражение примет вид, представленный

формулой (1.3):

$$V * W = (v_1 + w_2)(v_2 + w_1) + (v_3 + w_4)(v_4 + w_3) - v_1 v_2 - v_3 v_4 - w_1 w_2 - w_3 w_4 \quad (1.3)$$

Менее очевидно, что 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. На практике это означает, что над предварительно обработанными элементами придется выполнять лишь первые два умножения и последующие пять сложений, а также дополнительно два сложения. Это и сокращает общий объём вычислений, необходимый для умножения матриц.

1.3 Вывод

В этом разделе были сформулированы цели и задачи работы, а также даны математические описания стандартного алгоритма умножения матриц и алгоритма Винограда. Отличительными чертами второго являются предварительная обработка и количество операций умножения.

2 Конструкторская часть

Входные данные - размеры матриц и их содержимое - формируются программно с помощью генератора случайных чисел. После вывода демонстрационного примера программа запрашивает у пользователя ввод любого символа, кроме перевода строки, что играет роль паузы перед началом измерения скорости работы алгоритмов умножения.

2.1 Требования к программному обеспечению

Программа должна выводить пример умножения матриц небольшого размера с целью контроля правильности работы алгоритмов, а также информацию об измерениях времени.

Для удобства пользователя дополнительно выводятся проценты, по которым можно наглядно увидеть соотношение времени работы алгоритмов.

2.2 Схемы алгоритмов

На рисунках 2.1 и 2.2 представлены схемы стандартного алгоритма умножения матриц и алгоритма Винограда соответственно. На рисунке 2.3 изображена схема самостоятельно оптимизированного алгоритма Винограда.

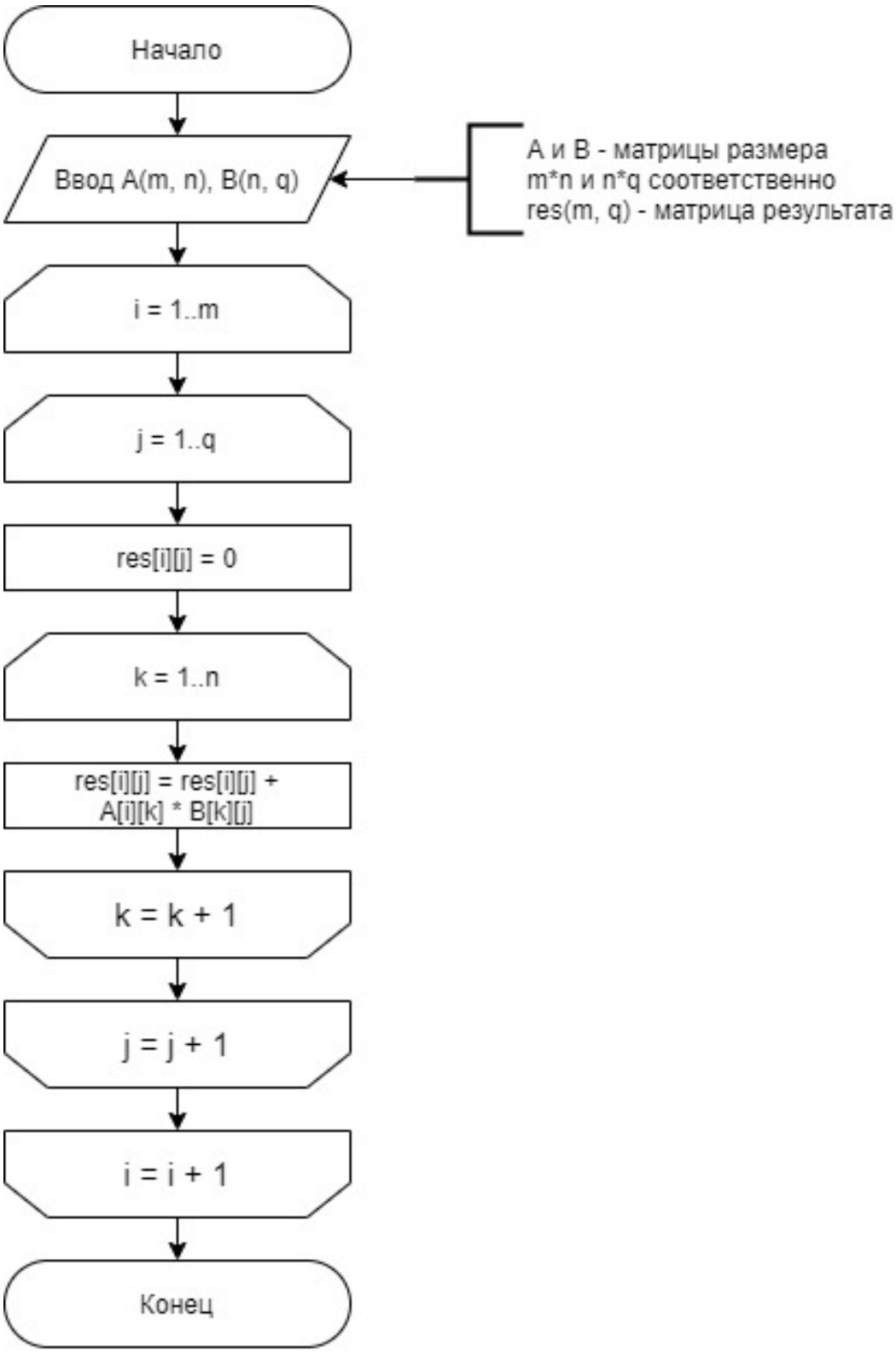


Рис. 2.1: Схема стандартного алгоритма умножения матриц

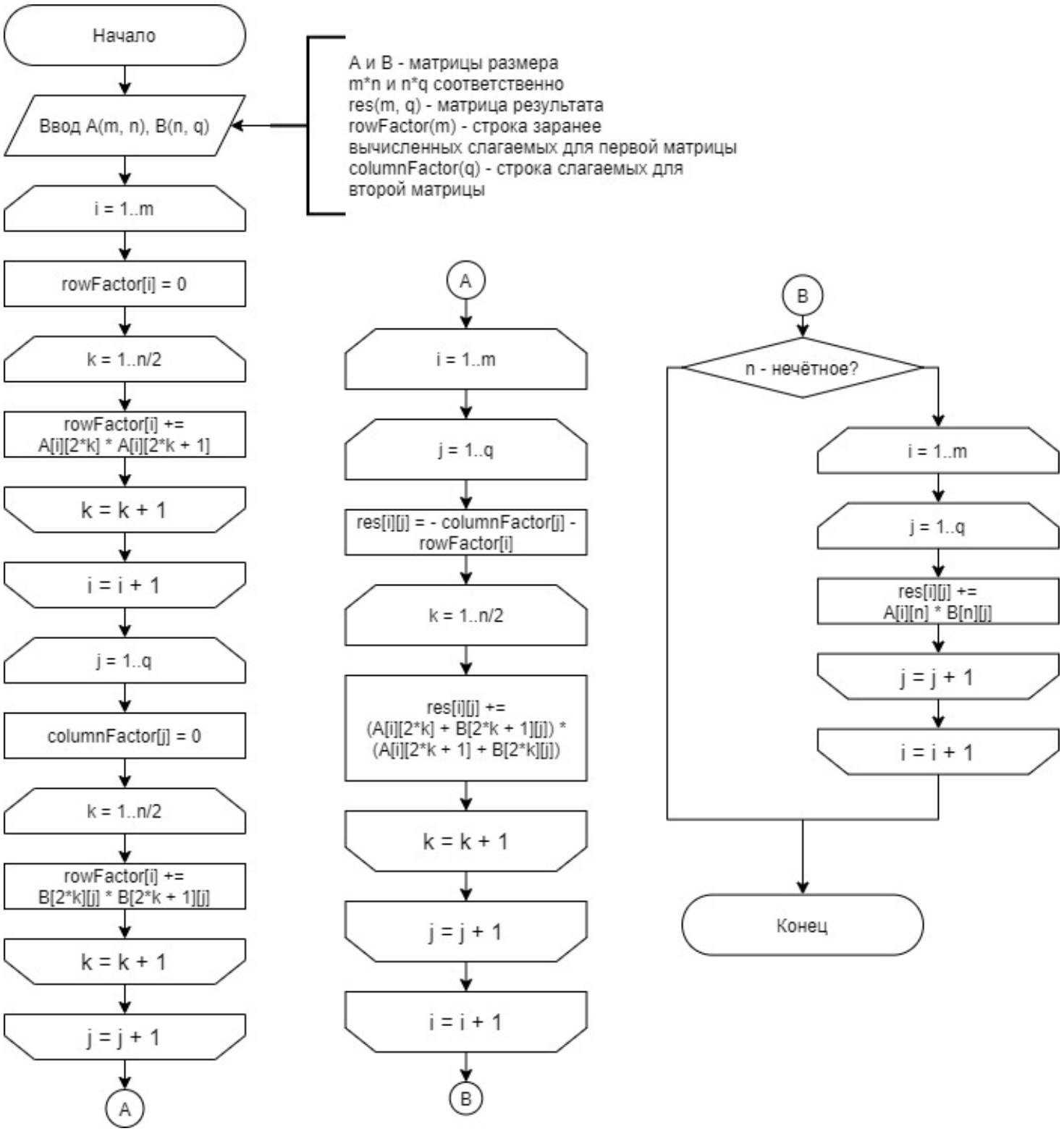


Рис. 2.2: Схема алгоритма Винограда

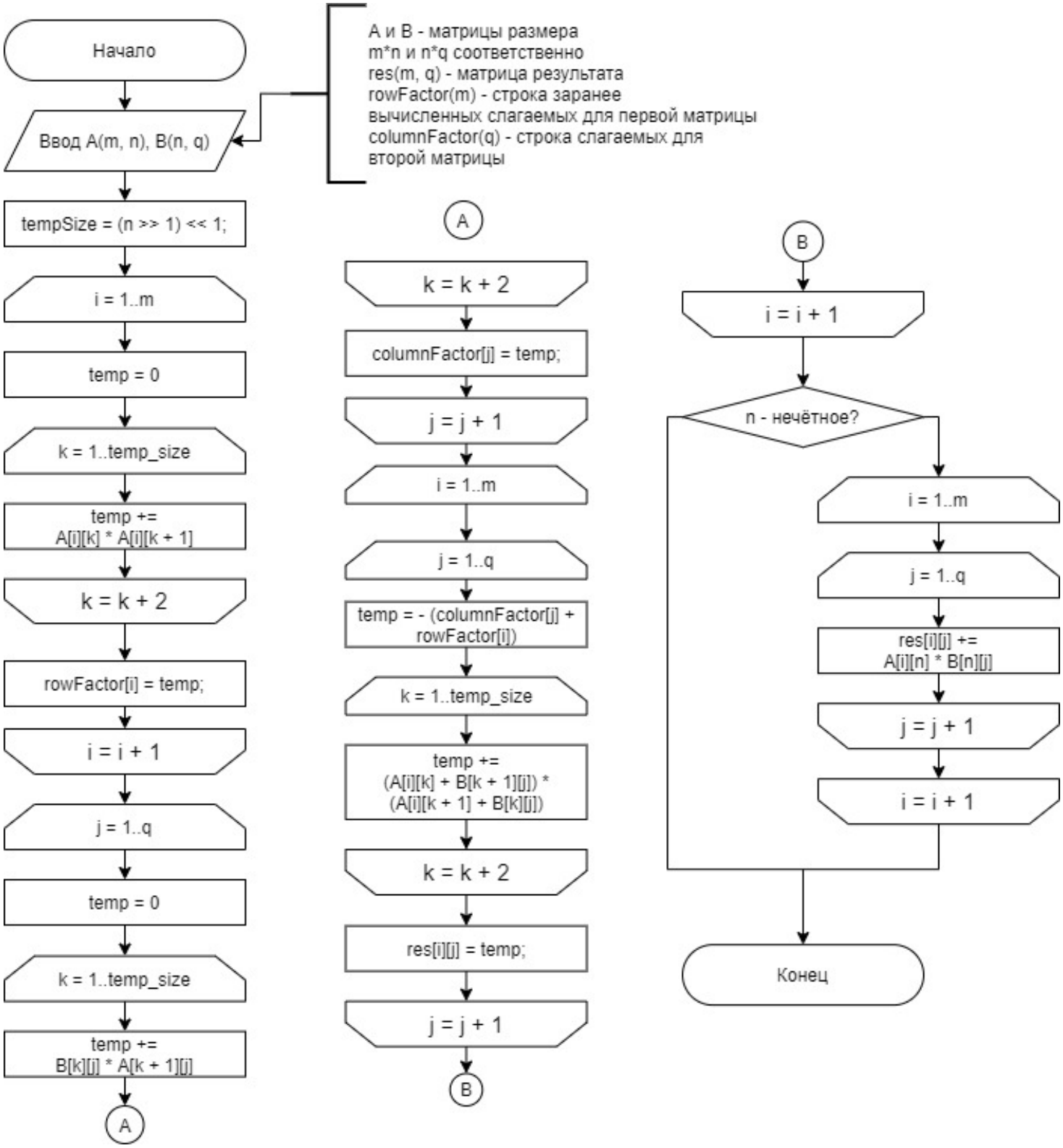


Рис. 2.3: Схема оптимизированного алгоритма Винограда

2.3 Оптимизация классического алгоритма Винограда

В целом, проведённая мной оптимизация не затрагивает общую структуру алгоритма. Изменения состоят в следующем:

1) все умножения заменены сложением - переменная, по которой происходит итерирование, увеличивается на 2 на каждом шаге, а максимальный предел увеличен в два раза;

2) вводятся дополнительные переменные:

   - для циклов по k;
   - для размеров матрицы (для удобства чтения кода);
   - буферная переменная для подсчёта значений ячеек векторов и результирующей матрицы, что сокращает количество обращений к ним;

3) остальные отличия слишком малы, чтобы выделять под них новый пункт - это другие способ определения чётности матрицы и сложение элементов векторов перед отрицанием в основном цикле.

Стоит заметить, что у первого пункта есть варианты - можно было заменить умножения не на сложение, а на сдвиги. Однако стоит заметить, чуть забежав вперёд, что вариант со сдвигами был хуже по времени на 0,2-0,4%. Несущественно, но раз уж такое улучшение было замечено, я решил его оставить.

2.4 Модель трудоемкости

Модель трудоемкости для оценки алгоритмов:

1) стоимость базовых операций - 1:

   $=, +, \simeq, <, >, \geq, \leq, ==, !=, [], ++, --, <<, >>$;

2) операции повышенной стоимости - 2:

   $+=, -=, [i][j]$ - операция обращения к ячейке матрицы;

3) дорогие операции - 4 и 5 (если со "встроенным"присваиванием):

   $*, /, \%, *=, /=$;

4) стоимость цикла:

   $f_{for} = f_{init} + f_{comp} + M(f_{body} + f_{increment} + f_{comp})$

   Пример: $for(i = 0; i < N; i++)/ * body * /$,

   Результат: $2 + N * (2 + f_{body})$ (в итоговых расчётах может отличаться из-за другого инкремента);

5) стоимость условного оператора:

   Пусть переход к одной из ветвей стоит 0, тогда

$$f_f = f_{cond} + \begin{cases} min(f_A, f_B), & \text{лучший случай} \\ max(f_A, f_B), & \text{худший случай} \end{cases}$$

2.5 Расчёт трудоёмкости алгоритмов

2.5.1 Трудоемкость стандартного алгоритма

Подсчет: $2 + M(2 + 2 + Q(2 + 2 + 1 + 2 + N(2 + 2 + 2 + 4 + 2 + 2))) = 2 + M(4 + Q(7 + N * 14)) = 2 + M(4 + 7Q + 14NQ) = 2 + 4M + 7MQ + 14MNQ$

В рамках выбранной модели, трудоёмкость данного алгоритма описывается формулой (2.1).

$$2 + 4M + 7MQ + 14MNQ \tag{2.1}$$

Считая, что M, N и Q одного порядка, получаем итоговую оценку $14N^3$.

2.5.2 Трудоемкость алгоритма Винограда

Всего необходимого подсчитать трудоёмкость четырёх циклов, один из которых - условный. Трудоёмкости циклов и развилки на случай нечётного N описываются формулами (2.2), (2.3), (2.4) и (2.5) по порядку.

$$2 + 10M + \frac{19}{2}MN \tag{2.2}$$

$$2 + 10Q + \frac{19}{2}NQ \tag{2.3}$$

$$2 + 4M + 15MQ + \frac{37}{2}MNQ \tag{2.4}$$

$$\begin{bmatrix} 5 & , \text{ невыполнение} \\ 2 + 4M + 16MQ & , \text{ выполнение} \end{bmatrix} \tag{2.5}$$

Итоговое значение описывается формулой (2.6):

$$\frac{37}{2}MNQ + 15MQ + \frac{19}{2}MN + \frac{19}{2}NQ + 10M + 10Q + 6+$$

$$\begin{bmatrix} 5 & , \text{невыполнение} \\ 2 + 4M + 16MQ & , \text{выполнение} \end{bmatrix} \tag{2.6}$$

В итоге получается $\frac{37}{2}N^3$. Пока что это значение хуже, чем у стандартного алгоритма. Необходимо оптимизации.

2.5.3 Трудоемкость оптимизированного алгоритма Винограда

В начале алгоритма производится несколько операций вне циклов. Эти числа будут прибавлены к оценке первого цикла.

Всего необходимого подсчитать трудоёмкость четырёх циклов, один из которых - условный. Трудоёмкости циклов и развилки на случай нечётного N описываются формулами (2.7), (2.8), (2.9) и (2.10) по порядку.

$$5 + 5M + 7MN \tag{2.7}$$

$$2 + 5Q + 7NQ \tag{2.8}$$

$$2 + 4M + 12MQ + \frac{19}{2}MNQ \tag{2.9}$$

$$\begin{bmatrix} 1 & , \text{ невыполнение} \\ 3 + 4M + 14MQ & , \text{ выполнение} \end{bmatrix} \tag{2.10}$$

Итоговое значение описывается формулой (2.11):

$$\frac{19}{2}MNQ + 12MQ + 7MN + 7NQ + 9M + 5Q + 9+$$

$$\begin{bmatrix} 1 & , \text{невыполнение} \\ 3 + 4M + 14MQ & , \text{выполнение} \end{bmatrix} \tag{2.11}$$

В итоге получается $\frac{19}{2}N^3$. Это значение превосходит аналогичное для стандартного алгоритма.

2.6 Вывод

В ходе конструкторской части были сформулированы требования к программе, разработаны схемы алгоритмов, с помощью которых можно осуществить написание программы, а также проведена теоретическая оценка в рамках выбранной модели трудоёмкости алгоритмов.

3 Технологическая часть

3.1 Выбор языка программирования

Для написания программных реализаций алгоритмов был выбран язык программирования C++, ввиду изучения оного на прошлых курсах, наличия некоторых элементов программы (реализация универсальной матрицы).

Среда разработки – Visual Studio 2019.

В компиляторе полностью отключена автоматическая оптимизация, насколько это позволяют сделать настройки, чтобы продемонстрировать полезные эффекты ручной оптимизации.

3.2 Реализация алгоритмов

В приводимых ниже листингах используется при подстановке в шаблоны тип Word– псевдоним для целочисленного типа short int, размер которого составляет 2 байта.

Привычная реализация умножения матриц, основанная на математической формуле, представлена на листинге 3.1.

Листинг 3.1: Стандартный алгоритм умножения матриц

```
1 template <typename T>
2 Matrix<T> Matrix<T>::multiplication(const Matrix<T>& mat) const {
3     {
4         time_t _time = time(NULL);
5         if (!this->isSuitForMult(mat))
```

```
6               throw MultiplicationException(__FILE__, typeid(*this).
                    name(), __LINE__, ctime(&_time));
7       }
8       Matrix<T> res(this->m_rows, mat.m_columns);
9       for (size_t i = 0; i < this->m_rows; ++i) // n
10          for (size_t j = 0; j < mat.m_columns; ++j) { // m
11              res(i, j) = 0;
12              for (size_t k = 0; k < this->m_columns; ++k) // l
13                  res(i, j) += (*this)(i, k) * mat(k, j); }
14      return res;
15  }
```

Алгоритм Винограда, реализованный по классической его записи, представлен на листинге 3.2.

Листинг 3.2: Реализация алгоритма Винограда

```
1   template <typename T>
2   Matrix<T> Matrix<T>::multiplicationVinograd(const Matrix<T>& mat)
        const {
3       {
4           time_t _time = time(NULL);
5           if (!this->isSuitForMult(mat))
6               throw MultiplicationException(__FILE__, typeid(*this).
                    name(), __LINE__, ctime(&_time));
7       }
8       Matrix<T> res(this->m_rows, mat.m_columns);
9
10      MatrixLine rowFactor(this->m_rows);
11      for (size_t i = 0; i < this->m_rows; i++) {
12          rowFactor[i] = 0;
13          for (size_t k = 0; k < this->m_columns / 2; k++)
14              rowFactor[i] = rowFactor[i] + (*this)[i][2 * k] * (*this)
                    [i][2 * k + 1];
15      }
16
17      MatrixLine columnFactor(mat.m_columns);
18      for (size_t i = 0; i < mat.m_columns; i++) {
19          columnFactor[i] = 0;
```

```
20          for (size_t k = 0; k < this->m_columns / 2; k++)
21              columnFactor[i] = columnFactor[i] + mat[2 * k][i] * mat[2
                    * k + 1][i];
22      }
23
24      for (size_t i = 0; i < this->m_rows; i++)
25          for (size_t j = 0; j < mat.m_columns; j++) {
26              res[i][j] = -rowFactor[i] - columnFactor[j];
27              for (size_t k = 0; k < this->m_columns / 2; k++)
28                  res(i, j) += ((*this)(i, 2 * k) + mat[2 * k + 1][j])
29                  * ((*this)[i][2 * k + 1] + mat[2 * k][j]);
30          }
31
32      if ((this->m_columns % 2) == 1)
33          for (size_t i = 0; i < this->m_rows; ++i)
34              for (size_t j = 0; j < mat.m_columns; ++j)
35                  res[i][j] += (*this)[i][this->m_columns - 1] * mat[
                        this->m_columns - 1][j];
36
37      return res;
38  }
```

Алгоритм Винограда, оптимизированный самостоятельно, представлен на листинге 3.3.

Листинг 3.3: Оптимизированная реализация алгоритма Винограда

```
1  template <typename T>
2  Matrix<T> Matrix<T>::multiplicationVinogradOptimised(const Matrix<T>&
       mat) const {
3      {
4          time_t _time = time(NULL);
5          if (!this->isSuitForMult(mat))
6              throw MultiplicationException(__FILE__, typeid(*this).
                   name(), __LINE__, ctime(&_time));
7      }
8      //      this      a x b       mat
9      //      b x c     .             res      a x c
10     //       rowFactors    G
```

```
11      const size_t m = this->m_rows, q = mat.m_columns;
12      Matrix<T> res(m, q);
13      size_t temp_size = (this->m_columns >> 1) << 1;
14      MatrixLine rowFactor(m);
15      MatrixLine columnFactor(q);
16
17      T temp;
18
19      for (size_t i = 0; i < m; ++i) {
20          temp = 0;
21          for (size_t k = 0; k < temp_size; k += 2)
22              temp += ((*this)(i, k) * (*this)(i, k + 1));
23          rowFactor[i] = temp;
24      }
25
26      for (size_t i = 0; i < q; ++i) {
27          temp = 0;
28          for (size_t k = 0; k < temp_size; k += 2)
29              temp += mat(k, i) * mat(k + 1, i);
30          columnFactor[i] = temp;
31      }
32
33      for (size_t i = 0; i < m; ++i)
34          for (size_t j = 0; j < q; ++j) {
35              temp = -(rowFactor[i] + columnFactor[j]);
36              for (size_t k = 0; k < temp_size; k += 2) {
37                 temp += ((*this)(i, k) + mat(k + 1, j))
38                       * ((*this)(i, k + 1) + mat(k, j));
39              }
40              res(i, j) = temp;
41          }
42
43      if ((temp_size) != this->m_columns)//((this->m_columns % 2) == 1)
44          for (size_t i = 0, t = temp_size; i < m; ++i)
45              for (size_t j = 0; j < q; ++j)
46                  res(i, j) += (*this)(i, t) * mat(t, j);
47
48      return res;
```

```
49 }
```

3.3 Тестирование

Программа не подразумевает ручной ввод, что отбрасывает ситуации с заведомо ошибочными данными. Правильность расчётов можно отследить, сравнивая результаты между собой и с самостоятельно полученным расчётом по выведенным исходным данным. Пример такой проверки можно увидеть в таблице 3.1.

Таблица 3.1: Результаты тестирования программы

Исходн. матр.	Ожидается	Стандарт. алг.	Алг. Винограда	Опт. алг. Винограда
Чётное N				
6 1 2 11 14 7 * 4 1 7 2 6 4 13 7	30 10 55 19 74 46 157 81 98 42 189 77	30 10 55 19 74 46 157 81 98 42 189 77	30 10 55 19 74 46 157 81 98 42 189 77	30 10 55 19 74 46 157 81 98 42 189 77
Нечётное N				
0 1 3 12 3 2 3 0 1 7 11 3 10 7 3 * 2 0 11 14 0 0 5 6 1 2	74 92 49 62 93 90	74 92 49 62 93 90	74 92 49 62 93 90	74 92 49 62 93 90

Все результаты, полученные в ходе тестирования, соответствуют ожидаемым.

3.4 Вывод

В ходе технологической части были успешно реализованы и протестированы требуемые алгоритмы умножения матриц, что позволит произвести измерения

времени работы и перейти к следующей части.

4 Исследовательская часть

4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

Для удобства берутся две квадратные матрицы одинаковой размерности. Для замеров быстродействия проводится несколько десятков повторных вызовов функций, количество повторений рассчитывается динамически с учётом размерности матрицы - чтобы сократить время ожидания результатов до разумных пределов. Получается следующий ряд чисел: 755, 80, 42, 30, 23, 20, соответствующий размерам 10-11, 100-101, 200-201, 300-301, 400-401, 500-501. Быстродействие алгоритмов измеряется в процессорном времени, которое более точно отражает время работы алгоритмов, с помощью функции _rdtsc()[1].

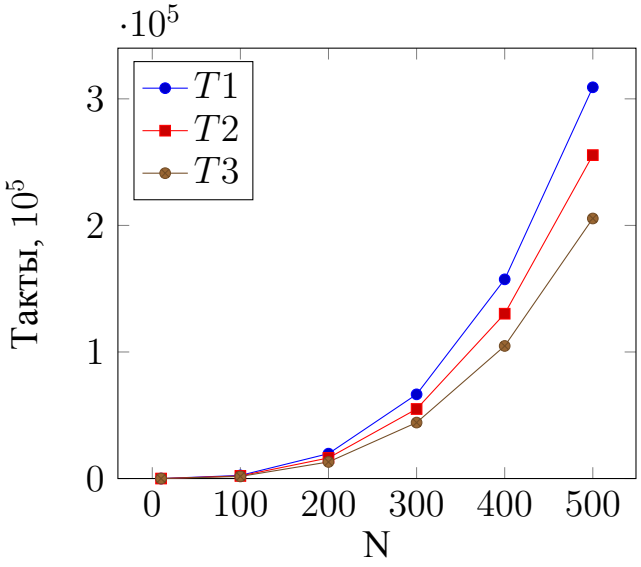
Полученные данные представлены в табл. 4.1. Всё время в таблице разделено на 10^5 - чтобы упростить и сократить запись.

Таблица 4.1: Результаты тестовых измерений скорости работы алгоритмов (в тактах процессора) в зависимости от размера матриц

N	T1 стд.	T2 Вин.	T3 опт. Вин	T1 / T3%
Чётное N				
10	2.73417	2.64742	2.17532	79.5604%
100	2476.08	2064.9	1667.78	67.3556%
200	19661	16289.4	13106.5	66.6624%
300	66526.4	54945.2	44215	66.4624%
400	157391	130247	104760	66.5603%
500	309136	255536	205471	66.4663%
Нечётное N				
11	3.64188	3.4966	2.93277	80.5291%
101	2588.48	2163.42	1745.67	67.44%
201	20344.3	16833.8	13553.3	66.6193%
301	67980.3	56186	45298.6	66.6349%
401	159935	132470	106642	66.6779%
501	309122	255487	205602	66.5117%

Полученные данные удобно воспринимать в виде графика, данного на рисунке 4.1. В нём представлены значения только для чётных N по той причине, что соответствующие показатели для нечётных размерностей мало отличаются от первых.

Рис. 4.1: Результаты тестовых измерений скорости работы алгоритмов (в тактах процессора) в зависимости от размера матриц



Из оценок трудоёмкости видно, что сложность всех алгоритмов пропорциональна N^3, что и можно наблюдать на графике. Однако они обладают разными константами-множителями, что и обеспечивает различие функций на изображении.

4.2 Вывод

Результаты тестирования показывают ожидаемый результат, что самым быстрым способом перемножить матрицы является модифицированный алгоритм Винограда, а самым медленным - стандартный алгоритм.

Стоит заметить, что сам по себе алгоритм Винограда, несмотря на теоретический расчёт, показывающий как будто его неэффективность относительно стандартного алгоритма, уже, начиная от матриц 100 на 100, требует около 80% от времени вычисления произведения с помощью привычной формулы. В дальнейшей оптимизации удалось добиться ускорения относительного этого значения около 20%, причём наибольшым полезным эффектом обладает использование буферных переменных: оказалось, что постоянно встречающиеся в исходном коде умножения и деления на 2 добавляют всего 1-2% лишней работы в то время, как остальная нагрузка приходится на постоянные обращения к элементам матрицы.

Необходимо учитывать, что это достигается использованием нескольких дополнительных переменных. Их немного, но это необходимо учитывать в гипотетических ситуациях, когда размеры доступной программе памяти предельно ограничены.

Заключение

В ходе лабораторной работы были реализованы и изучены различные алгоритмы умножения матриц, проведены экспериментальное сравнение по затраченному времени и теоретическая оценка трудоёмкости. Проведённое исследование позволяет сделать вывод, что классический алгоритм Винограда и его оптимизированная версия не хуже по времени при малых размерах матрицы и лучше - при больших, причём вторая является наилучшей среди всех трёх.

Чтобы проиллюстрировать приведённое утверждение в числах, посчитаем средние показания по тестам с размерностями со 100 до 500. Принимая время вычисления по стандартной формуле за эталон, увидим, что для классической и оптимизированной версии алгоритма Винограда необходимо 83% и 66,7% от эталона соответственно. Или же, записывая иначе, первая реализация быстрее обычного произведения в 1,2 раза, а вторая – в 1,5 раза.

Впрочем, необходимо помнить о том, что ускорение достигается путём использования дополнительной памяти. Если задача не накладывает особых ограничений в этом плане, можно считать оптимизированную версию алгоритма Винограда наилучшей для вычисления плотных матриц любого размера среди алгоритмов с кубической оценкой сложности.

Список литературы

1. Функция _rdtsc() – документация Майкрософт [эл. ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/cpp/intrinsics/rdtsc?view=vs-2019 (дата обращения: 27.09.2020).

2. Дж. Макконнелл ”Основы современных алгоритмов. 2-е дополненное издание”. (дата обращения: 20.10.2020).