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 №2 по дисциплине "Анализ алгоритмов"

Тема Умножение матриц

Студент Байрамгалин Я.Р.

Группа ИУ7-53Б

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

Москва — 2022 г.

Оглавление

Введение

Матрицей размера $m \cdot n$ называют прямоугольную таблицу из элементов произвольного типа, имеющую m строк и n столбцов.

Целью работы является сравнение трудоемкости различных алгоритмов перемножения матриц, а именно обычного алгоритма, алгоритма Винограда и алгоритма винограда с оптимизациями, согласно варианту.

Для достижения цели ставятся следующие задачи:

- изучить классический алгоритм умножения матриц, алгоритм Винограда и модифицированный алгоритм Винограда;
- реализовать классический алгоритм умножения матриц, алгоритм Винограда и оптимизированный алгоритм Винограда;
- дать оценку трудоёмкости алгоритмов;
- замерить время работы алгоритмов;
- провести сравнительный анализ на основе полученных экспериментально данных.

1 Аналитическая часть

В этом, а также во всех последующих разделах матрицей размера $m \cdot n$ будем называть прямоугольную таблицу., содержащую m строк и n столбуов.

Для начала дадим определение умножения матриц. Пусть есть матрица A размера $m \cdot n$ и матрица B размера $n \cdot l$. Тогда матрицей $= A \cdot B$ будем называть матрицу

$$C = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1l} \\ a_{21} & a_{22} & \dots & a_{2l} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{ml} \end{pmatrix}$$

где

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

1.1 Стандартный алгоритм

Стандартный алгоритм заключается в подсчете значения для каждого c_{ij} согласно определению.

1.2 Алгоритм Винограда

Заметим, что значение каждого элемента c_{ij} равно скалярному произведению вектора-строки

$$V = (v_1, v_2, \dots, v_n) \tag{1.1}$$

на вектор-столбец

$$W = (w_1, w_2, \dots, w_n) \tag{1.2}$$

Таким образом получим, что:

$$c_{ij} = V \cdot W = \sum_{i=0}^{n} v_i w_i \tag{1.3}$$

Иначе это выражение можно записать как:

$$c_{ij} = V \cdot W = \sum_{i=0,2}^{n} ((v_i + w_i)(v_{i+1} + w_{i+1})) - \sum_{i=0,2}^{n} (v_i v_{i+1} + w_i w_{i+1}) \tag{1.4}$$

Также заметим, что второе слагаемое выражения 1.4 можно превычислить заранее для каджой строки левой матрицы и для каждого столбца правой.

Из-за такого подхода алгоритм Винограда в среднем будет использовать меньше умножений, чем стандартный алгоритм.

1.3 Алгоритм Винограда с оптимизациями

Согласно варианту необходимо работу алгоритма со следующими оптимизациями:

- заменить = на + = там, где это возможно;
- предвычислять некоторые выражения;
- использовать битовый сдвиг вместо умножения на 2.

Вывод

Рассмотрены теоретические аспекты алгоритмов умножения матриц. Получены достаточные знания для программной реализации.

2 Конструкторская часть

2.1 Схемы алгоритмов

Время выполнения стандартного алгоритма не зависит от каких-либо характеристик взржныхх матриц, из-за чего рассматривать лучший и худший случай будеи излишне.

В случае алгоритма Винограда стоит выделить 2 класса эквивалентности: четное и нечетное количество столбцов левой матрицы.

На рисунке 2.1 приведена схема стандартного алгоритма умножения матриц.

На рисунках 2.2 и 2.3 представлена схема алгоритма Винограда и схема оптимизированного алгоритма Винограда.

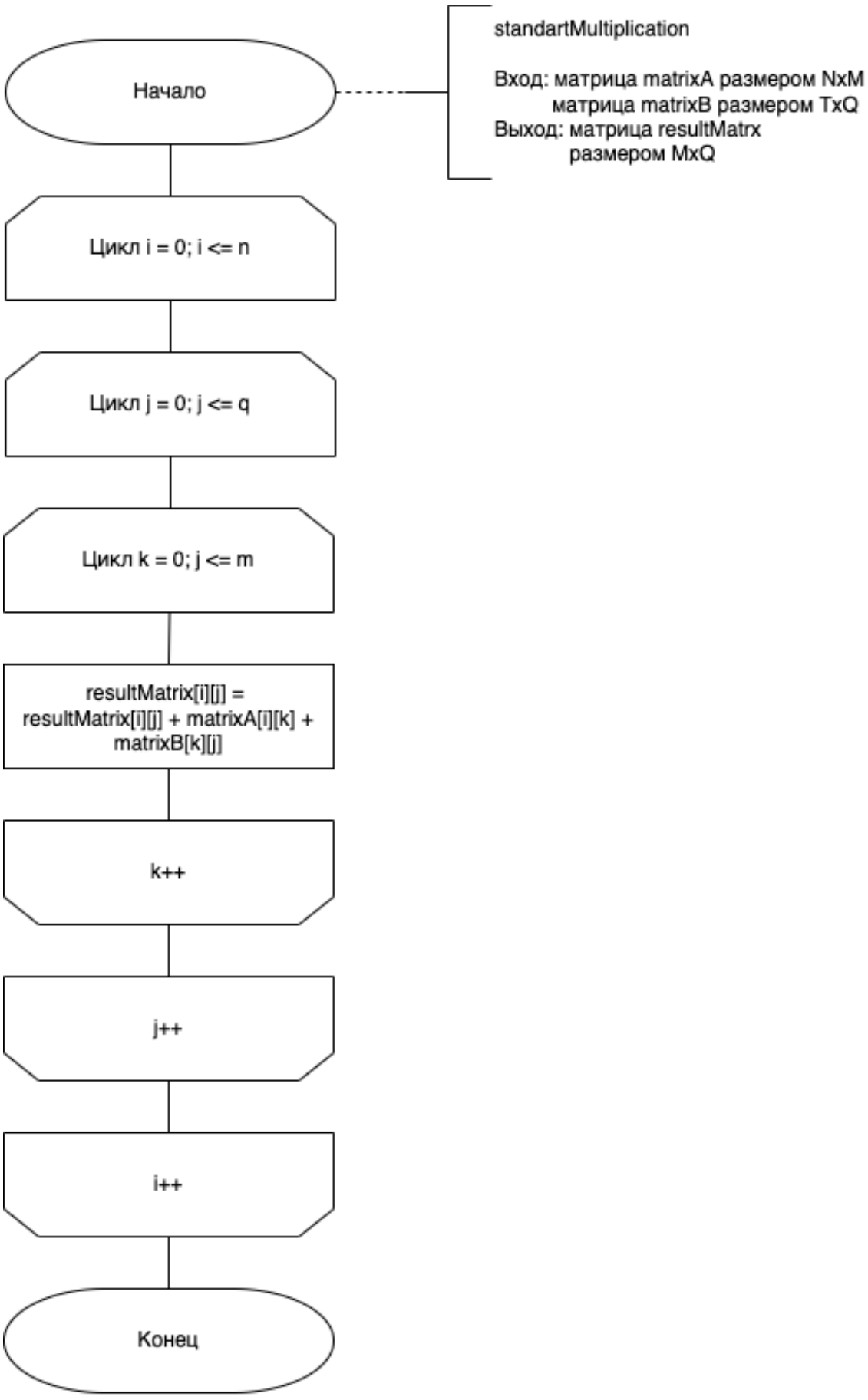

Рис. 2.1: Схема стандартного алгоритма умножения матриц

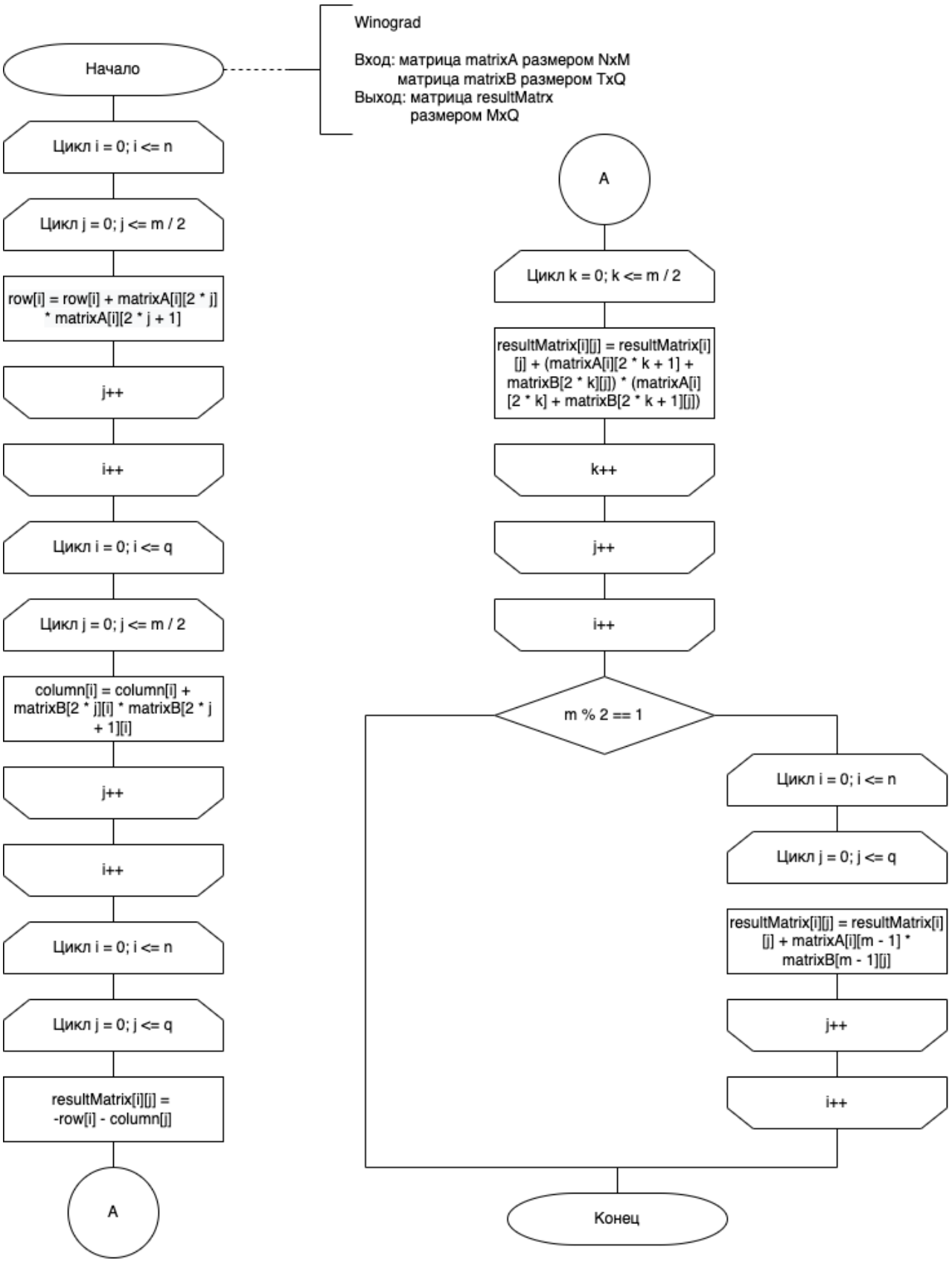

Рис. 2.2: Схема алгоритма Винограда

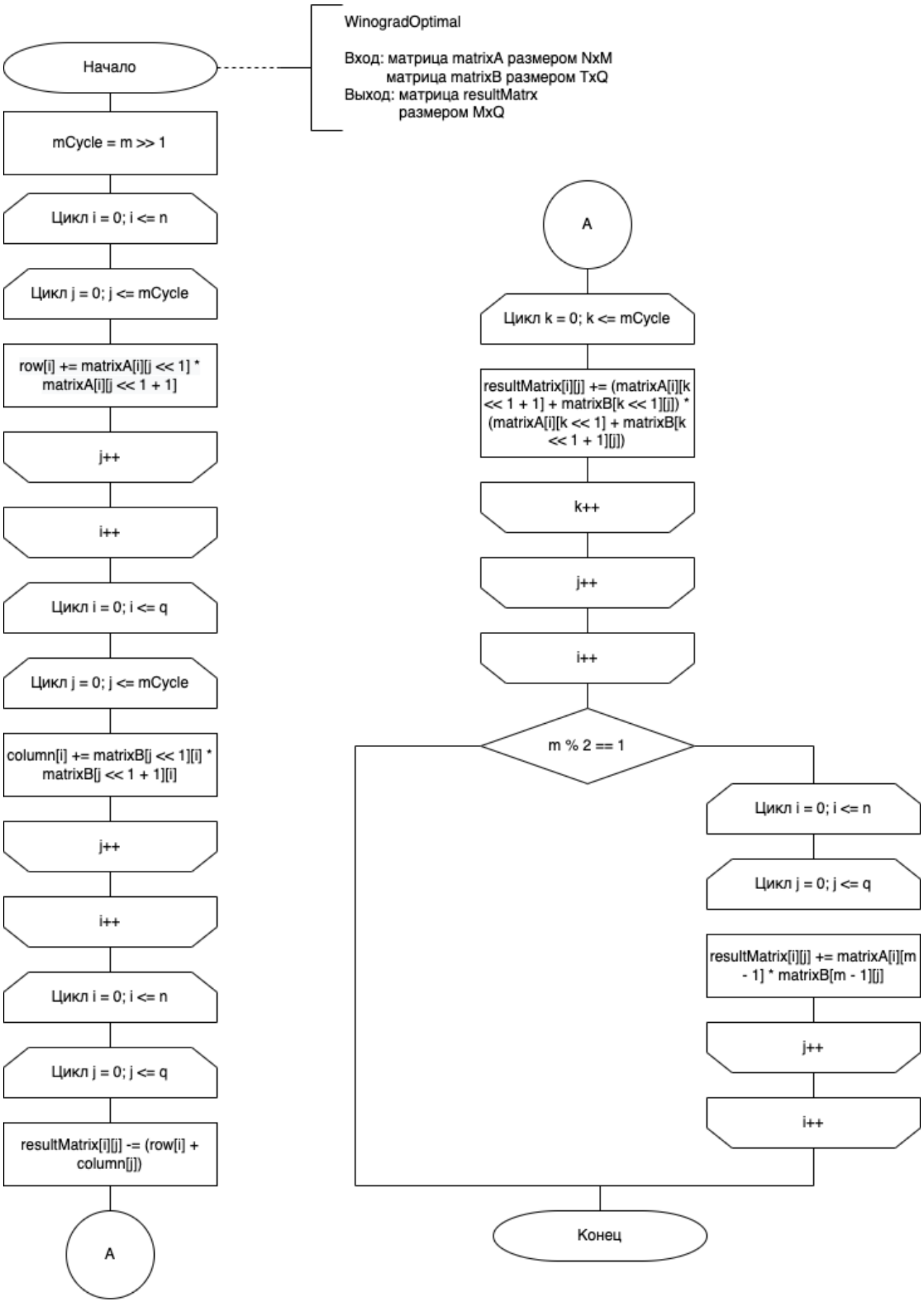

Рис. 2.3: Схема функций оптимизированного алгоритма Винограда

2.2 Модель вычислений

Для последующего вычисления трудоемкости введём модель вычислений.

1. Операции из списка (2.1) имеют трудоемкость 1.

$$+, -, \%, ==, !=, <, >, <=, >=, [], ++, -- \tag{2.1}$$

2. Операции из списка (2.2) имеют трудоемкость 2.

$$*, *=, /, /= \tag{2.2}$$

3. Трудоемкость оператора выбора if условие then A else B рассчитывается по формуле (2.3).

$$f_{if} = f_{\text{условия}} + \begin{cases} f_A, & \text{если условие выполняется,} \\ f_B, & \text{иначе.} \end{cases} \tag{2.3}$$

4. Трудоемкость цикла рассчитывается по формуле (2.4).

$$f_{for} = f_{\text{инициализации}} + f_{\text{сравнения}} + N(f_{\text{тела}} + f_{\text{инкремента}} + f_{\text{сравнения}}) \tag{2.4}$$

5. Трудоемкость вызова функции равна 0.

6. Трудоемкость выделения произвольного блока памяти на куче равна 10.

2.3 Трудоёмкость алгоритмов

2.3.1 Стандартный алгоритм умножения матриц

1. Трудоёмкость проверки на корректность рамеров перемножаемых матриц составляет:
$$f_{check} = 1 \tag{2.5}$$

2. Трудоёмкость создания матрицы:
$$f_{create} = 10 \tag{2.6}$$

3. Трудоёмкость 1 цикла составляет:
$$f_{c1} = 1 + 2m + m \cdot f_{c2} \tag{2.7}$$

4. Трудоёмкость 2 цикла составляет:
$$f_{c2} = 1 + 2l + l \cdot f_{c3} \tag{2.8}$$

5. Трудоёмкость 2 цикла составляет:
$$f_{c3} = 1 + 2n + n \cdot 9 \tag{2.9}$$

Таким образом получим, что согласно введенной модели трудоемкость стандартного алгоритма составит:

$$f_{standart} = 12 + 3m + 3ml + 11mnl = O(mnl)$$

2.3.2 Алгоритм Винограда

1. Трудоёмкость проверки на корректность рамеров перемножаемых матриц составляет:
$$f_{check} = 1 \tag{2.10}$$

2. Трудоёмкость создания матрицы:

$$f_{create} = 10 \tag{2.11}$$

3. Трудоемкость подсчета сумм (предвычисление) для строк левой матрицы:

$$f_{row-factors} = 1 + 3m + 10mn \tag{2.12}$$

4. Трудоемкость подсчета сумм (предвычисление) для столбцов правой матрицы:

$$f_{column-factors} = 1 + 3n + 10nl \tag{2.13}$$

5. Трудоемкость выполнения основного цикла:

$$f_{cycle} = 9mnl + 8mn + 2m + 2 \tag{2.14}$$

6. Трудоемкость выполнения цикла при нечетном количестве строк второй матрицы:

$$f_{cycle} = 11ml + 3m + 3 \tag{2.15}$$

Для худшего случая имеем:

$$f_{va-best} = 16 + 5m + 3n + 18mn + 13ml + 11nl + 9mnl = O(mnl) \tag{2.16}$$

Для лучшего случая имеем:

$$f_{va-best} = 16 + 5m + 3n + 18mn + 11nl + 9mnl = O(mnl) \tag{2.17}$$

2.3.3 Оптимизированный алгоритм Винограда

Трудоемкость оптимизированного алгоритма винограда рассчитывается аналогично обычному алгоритму винограда и составляет:

$$f_{va-best} = 16 + 5m + 3n + 15mn + 10nl + 8mnl = O(mnl) \tag{2.18}$$

Вывод

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

3 Технологическая часть

В данном разделе будут приведены требования к программному обеспечению, используемые технологии и реализации алгоритмов.

3.1 Требования к программному обеспечению

К программе предъявляется ряд требований:

- на вход подаются 2 произвольные матрицы с численным типом элементов;
- при невозмодности выполнить умножение кидается исключение;
- в случае успешного выполнения программы на выходе матрица, которая является результатом умножения первой матрицы на вторую (в порядке их поступления на вход).

3.2 Выбор средств реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язые программирования C++[1].

Язык позволяет управлять всеми ресурсами компьютера и, тем самым позволяет писать эффективные алгоритмы.

Время работы алгоритмов было замерено с помощью функции из листинга 3.1, разработанной самостоятельно.

```
template <typename F, typename... Args>
auto GetExecutionTime(F function, Args&&... args) {
  const auto start_time = std::chrono::high_resolution_clock::now();
  const auto return_value = function(std::forward<Args>(args)...);
  const auto end_time = std::chrono::high_resolution_clock::now();
  return std::pair(
      std::chrono::duration_cast<TimeUnit>(end_time - start_time),
      return_value);
}
```

Листинг 3.1: Функция для замера времени исполнения функции

3.3 Реализация алгоритмов

В листингах 3.1, 3.2, 3.3 представлены реализации алгоритма обычного умножения матриц, алгоритма Винограда, алгоритма Винограда с оптимизациями соответственно.

```
template<typename ValueType>
models::Matrix<ValueType> MultiplySimple(
const models::Matrix<ValueType>& first,
const models::Matrix<ValueType>& second) {
  if (first.GetSize().second != second.GetSize().first) {
    throw std::runtime_error("Inappropriate matrix dimensions to execute multiplication");
  }

  const auto result_size = std::pair(first.GetSize().first, second.GetSize().second);
  models::Matrix<ValueType> result{result_size.first, result_size.second};
  for (std::size_t i = 0; i < result_size.first; ++i) {
    for (std::size_t j = 0; j < result_size.second; ++j) {
      for (std::size_t k = 0; k < first.GetSize().second; ++k) {
        result[i][j] += first[i][k] * second[k][j];
      }
    }
  }

  return result;
}
```

Листинг 3.2: Функция стандартного умножения матриц

```
template<typename ValueType>
models::Matrix<ValueType> MultiplyVinograd(
const models::Matrix<ValueType>& first,
const models::Matrix<ValueType>& second) {
  if (first.GetSize().second != second.GetSize().first) {
    throw std::runtime_error("Inappropriate matrix dimensions to execute multiplication");
  }

  typename models::Matrix<ValueType>::Row row_factors(first.GetSize().first);
  for (std::size_t i = 0; i < first.GetSize().first; ++i) {
    for (std::size_t j = 0; j < first.GetSize().second - 1; j = j + 2) {
      row_factors[i] = row_factors[i] + first[i][j] * first[i][j + 1];
    }
  }
  typename models::Matrix<ValueType>::Row
  column_factors(second.GetSize().second);
  for (std::size_t i = 0; i < second.GetSize().second; ++i) {
    for (std::size_t j = 0; j < second.GetSize().first - 1; j = j + 2) {
      column_factors[i] = column_factors[i] + second[j][i] * second[j + 1][i];
    }
  }

  const auto result_size = std::pair(first.GetSize().first,
  second.GetSize().second);
  models::Matrix<ValueType> result{result_size.first, result_size.second};
  for (std::size_t i = 0; i < result_size.first; ++i) {
    for (std::size_t j = 0; j < result_size.second; ++j) {
      result[i][j] = -row_factors[i] - column_factors[j];
      for (std::size_t k = 0; k < first.GetSize().second - 1; k = k + 2) {
        result[i][j] = result[i][j] + (first[i][k + 1] + second[k][j]) *
        (first[i][k] + second[k + 1][j]);
      }
    }
  }

  if (first.GetSize().second % 2 == 1) {
    for (std::size_t i = 0; i < result_size.first; ++i) {
      for (std::size_t j = 0; j < result_size.second; ++j) {
        result[i][j] =
        result[i][j] + first[i][first.GetSize().second - 1] *
        second[second.GetSize().first - 1][j];
      }
    }
```

```
44      }
45
46      return result;
47    }
```

Листинг 3.3: Функция алгоритма Винограда умножения матриц

```
1   template<typename ValueType>
2   models::Matrix<ValueType> MultiplyVinogradOptimized(
3   const models::Matrix<ValueType>& first,
4   const models::Matrix<ValueType>& second) {
5     if (first.GetSize().second != second.GetSize().first) {
6       throw std::runtime_error("Inappropriate matrix dimensions to execute
      "
7       "multiplication");
8     }
9
10    typename models::Matrix<ValueType>::Row row_factors(first.GetSize().
     first);
11    for (std::size_t i = 0; i < first.GetSize().first; ++i) {
12      const auto upper_index = first.GetSize().second - 1;
13      for (std::size_t j = 0; j < upper_index; j += 2) {
14        row_factors[i] += first[i][j] * first[i][j + 1];
15      }
16    }
17    typename models::Matrix<ValueType>::Row
18    column_factors(second.GetSize().second);
19    for (std::size_t i = 0; i < second.GetSize().second; ++i) {
20      const auto upper_index = second.GetSize().first - 1;
21      for (std::size_t j = 0; j < upper_index; j += 2) {
22        column_factors[i] += second[j][i] * second[j + 1][i];
23      }
24    }
25
26    const auto result_size = std::pair(first.GetSize().first,
27    second.GetSize().second);
28    models::Matrix<ValueType> result{result_size.first, result_size.second
     };
29    for (std::size_t i = 0; i < result_size.first; ++i) {
30      for (std::size_t j = 0; j < result_size.second; ++j) {
31        result[i][j] = -row_factors[i] - column_factors[j];
32        const auto upper_index = first.GetSize().second - 1;
33        for (std::size_t k = 0; k < upper_index; k += 2) {
34          result[i][j] += (first[i][k + 1] + second[k][j]) *
35          (first[i][k] + second[k + 1][j]);
36        }
37      }
38    }
39
```

```
40      if (first.GetSize().second % 2 == 1) {
41        for (std::size_t i = 0; i < result_size.first; ++i) {
42          for (std::size_t j = 0; j < result_size.second; ++j) {
43            result[i][j] += first[i][first.GetSize().second - 1] *
44            second[second.GetSize().first - 1][j];
45          }
46        }
47      }
48
49      return result;
50    }
```

Листинг 3.4: Функция оптимизированного алгоритма Винограда умножения матриц

3.4 Тестрование

В таблице 3.1 рассмотрены случаи ожидаемого поведения программы. Все тесты были успешно пройдены.

Таблица 3.1: Функциональные тесты

Матрица 1	Матрица 2	Ожидаемый результат
$\begin{pmatrix} \end{pmatrix}$	$\begin{pmatrix} \end{pmatrix}$	Сообщение об ошибке
$\begin{pmatrix} 1 & 5 & 7 \\ 2 & 6 & 8 \\ 3 & 7 & 9 \end{pmatrix}$	$\begin{pmatrix} \end{pmatrix}$	Сообщение об ошибке
$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$	Сообщение об ошибке
$\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$	Сообщение об ошибке
$\begin{pmatrix} 1 & a & 3 \end{pmatrix}$	$\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$	Сообщение об ошибке
$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 1 \\ a \\ 1 \end{pmatrix}$	Сообщение об ошибке
$\begin{pmatrix} 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$
$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 6 & 6 \\ 15 & 15 \\ 24 & 24 \end{pmatrix}$
$\begin{pmatrix} 2 \end{pmatrix}$	$\begin{pmatrix} 2 \end{pmatrix}$	$\begin{pmatrix} 4 \end{pmatrix}$

Вывод

Сформированы критерии для функции перемножения матриц. Разработаны 3 алгоритма умножения с использованием языка программирования C++. Разработанные алгоритмы удовлтворяют сформированным требованиям.

4 Исследовательская часть

В данном разделе будут приведены примеры работы программ, постановка эксперимента и сравнительный анализ алгоритмов на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование, следующие.

- Операционная система: macOs Monterey 12.4[2].
- Память: 16 GiB.
- Процессор: 2,6 GHz 6-ядерный процессор Intel Core i7[3].

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

4.2 Время выполнения алгоритмов

В таблице 4.1 представлены замеры времени работы для каждого из алгоритмов для квадратных матриц небольшого размера (1-9). Здесь и далее: СА — стандартный алгоритм, АВ — алгоритм Винограда, ОАВ – оптимизированный алгоритм Винограда. Время в микросекундах.

Также наглядной иллюстрацией зависимости времени выполнения от размера матрицы является график 4.1.

Таблица 4.1: Результаты замеров времени алгоритмов (миллисекунды)

Размер	CA	AB	OAB
10	57	48	29
50	5764	3996	2161
100	38218	25298	13236
200	290076	201203	107843
300	974186	791162	434455
400	2645162	1833453	1174284
500	5143356	3717723	2208044

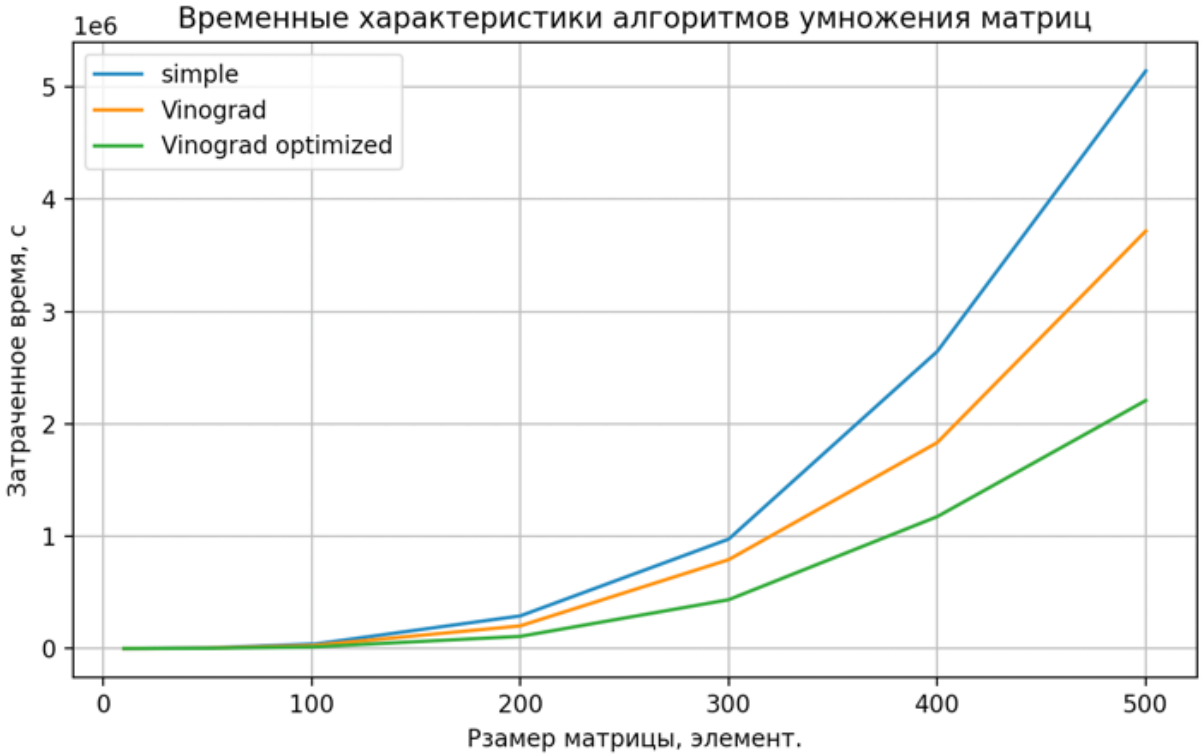

Рис. 4.1: График зависимости времени выполнения влгоритма от размеров квадратной матрицы)

Вывод

При увеличении размера матриц, увеличивается и эффективность работы алгоритма Винограда в сравнении со стандартным алгоритмом. В среднем реализация по Винограду работает быстрее в 1.7 раз. Оптимизированная реализация алгоритма Винограда также позволяет уменьшить время работы при больших размерностях: например, для размерности матрицы, равной 200, эксперимент показал, что алгоритм Винограда быстрее

стандартного алгоритма чуть менее, чем в 2 раза, когда оптимизированная версия выигрывает у обычного умножения более, чем в 2 раза.

Заключение

В ходе выполнения лабораторной работы были решены следующие задачи:

- были реализованы 3 алгоритма умножения матриц: обычный, Винограда, оптимизированный Винограда;
- был произведен анализ трудоёмкости алгоритмов на основе теоретических расчетов и выбранной модели вычислений;
- алгоритмы проанализированы на основе экспериментальных данных: выявлено, что реализация по Винограду работает в среднем в 1.7 раз быстрее стандартной, этот показатель улучшается при увеличении размера матриц;
- подготовлен отчет о лабораторной работе.

Список использованных источников

[1] C++ language [Электронный ресурс]. Режим доступа: https://en.cppreference.com/w/cpp/language (дата обращения: 04.10.2022).

[2] macos Monterey 12.03 [Электронный ресурс]. Режим доступа: https://www.apple.com/ru/macos/monterey/ (дата обращения: 04.10.2022).

[3] Процессор Intel® Core™ i5-1135G7 [Электронный ресурс]. Режим доступа: https://www.intel.ru/content/www/ru/ru/products/sku/208658/intel-core-i51135g7-processor-8m-cache-up-to-4-20-ghz/specifications.html (дата обращения: 04.10.2022).