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»

Лабораторная работа № 2
по дисциплине ”Анализ алгоритмов”

Тема Сравнение алгоритмов матричного умножения

Студент Калашников С.Д.

Группа ИУ7-53Б

Преподаватель Волкова Л.Л. Строганов Ю.В.

Москва, 2022

СОДЕРЖАНИЕ

Введение

Матрицы — крайне мощный инструмент, используемый в каждой точной науке: физике, математике, прораммировании и т.д. Они позволяют выполнять операции, требующие большого количества входных параметров и сложные в вычислении, с относительной легкостью: плоскопараллельный перенос и поворот точек в трехмерной графике, расчет роторов и дивергенции и др. В связи с этим встает вопрос об оптимизации основных алгоритмов, связанных с матрицами: сложения, умножения, транспонирования и т.п. В данной работе будет рассмотрена оптимизация алгоритмов матричного умножения.

Целью данной лабораторной работы является анализ трудоемкости реализаций алгоритмов матричного умножения.

Для достижения поставленной цели требуется решить ряд задач:

1) изучить алгоритмы умножения матриц — классический, Винограда, оптимизированный Винограда;
2) провести сравнительный анализ трудоёмкости алгоритмов на основе теоретических расчетов;
3) реализовать каждый из трех алгоритмов;
4) провести замеры процессорного времени для каждой из реализаций алгоритмов;
5) выполнить анализ полученных результатов;
6) по итогам работы составить отчет.

1. Аналитическая часть

1.1 Матрица

Пусть есть два конечных множества:

- номера строк $M = 1, 2, ..., m$,
- номера столбцов $N = 1, 2, ..., n$,

где m и n — натуральные числа. Тогда матрицей A размера m на n называется структура вида:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}; \tag{1.1}$$

где элемент матрицы a_{ij} находится на пересечении i-й строки и j-го столбца. При этом количество элементов матрицы равно $m * n$.

Можно выделить следующие операции над матрицами:

1) сложение матриц одинакового размера;
2) вычитание матриц одинакового размера;
3) умножение матриц в случае, если количество столбцов первой матрицы равно количеству строк второй матрицы. В итоговой матрице количество строк будет, как у первой матрицы, а столбцов – как у второй.

1.2 Описание алгоритмов

В этом разделе будут рассмотрены следующие алгоритмы матричного умножения: классический, Винограда, оптимизированный Винограда.

1.2.1 Классический алгоритм умножения матриц

Пусть даны две матрицы

$$A_{lm} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{l1} & a_{l2} & \dots & a_{lm} \end{pmatrix}, \quad B_{mn} = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{pmatrix}, \tag{1.2}$$

тогда матрица C

$$C_{ln} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{l1} & c_{l2} & \dots & c_{ln} \end{pmatrix}, \tag{1.3}$$

где

$$c_{ij} = \sum_{r=1}^{m} a_{ir} b_{rj} \quad (i = \overline{1, l}; j = \overline{1, n}) \tag{1.4}$$

будет называться произведением матриц A и B.

Стандартный алгоритм реализует данную формулу.

1.2.2 Алгоритм Винограда

Алгоритм Винограда — алгоритм умножения матриц. Начальная версия имела асимптотическую сложность алгоритма примерно $O(n^{2,3755})$, где n – размер стороны матрицы, но после доработки он стал обладать лучшей асимптотикой среди всех алгоритмов умножения матриц [1].

Рассмотрим два вектора $U = (u_1, u_2, u_3, u_4)$ и $W = (w_1, w_2, w_3, w_4)$. Их скалярное произведение равно: $U \cdot W = u_1w_1 + u_2w_2 + u_3w_3 + u_4w_4$, что эквивалентно:

$$V \cdot W = (u_1 + w_2)(u_2 + w_1) + (u_3 + w_4)(u_4 + w_3) - u_1 u_2 - u_3 u_4 - w_1 w_2 - w_3 w_4. \quad (1.5)$$

За счёт предварительной обработки данных можно получить прирост производительности: несмотря на то, что полученное выражение требует большего количества операций, чем стандартное умножение матриц, выражение в правой части равенства можно вычислить заранее и запомнить для каждой строки первой матрицы и каждого столбца второй матрицы. Это позволит выполнить лишь два умножения и пять сложений, при учёте, что потом будет сложено только с двумя предварительно посчитанными суммами соседних элементов текущих строк и столбцов. Операция сложения выполняется быстрее, поэтому алгоритм должен работать быстрее обычного алгоритма перемножения матриц.

Стоит упомянуть, что при нечётном значении размера матрицы M нужно дополнительно добавить к скалярному произведению векторов произведения крайних элементов соответствующих строк и столбцов.

1.2.3 Оптимизация алгоритма Винограда

При реализации рассмотренного выше алгоритма Винограда можно провести оптимизации.

1. Операции сложения и вычитания с присваиванием следует реализовывать при помощи соответствующего оператора $+=$ или $-=$ (при наличии данных операторов в выбранном языке программирования).

2. Операцию умножения на 2 программно эффективнее реализовывать как побитовый сдвиг влево на 1.

3. Занесение в циклах вычисления множителей вычисления первых двух элементов во внутренний цикл j.

2. Конструкторская часть

2.1 Описание алгоритмов

В данном разделе будут рассмотрены схемы алгоритмов классического умножения матриц (рис. 2.1), Винограда (рис. 2.2, 2.3, 2.4), оптимизированного Винограда (рис. 2.5, 2.6, 2.7).

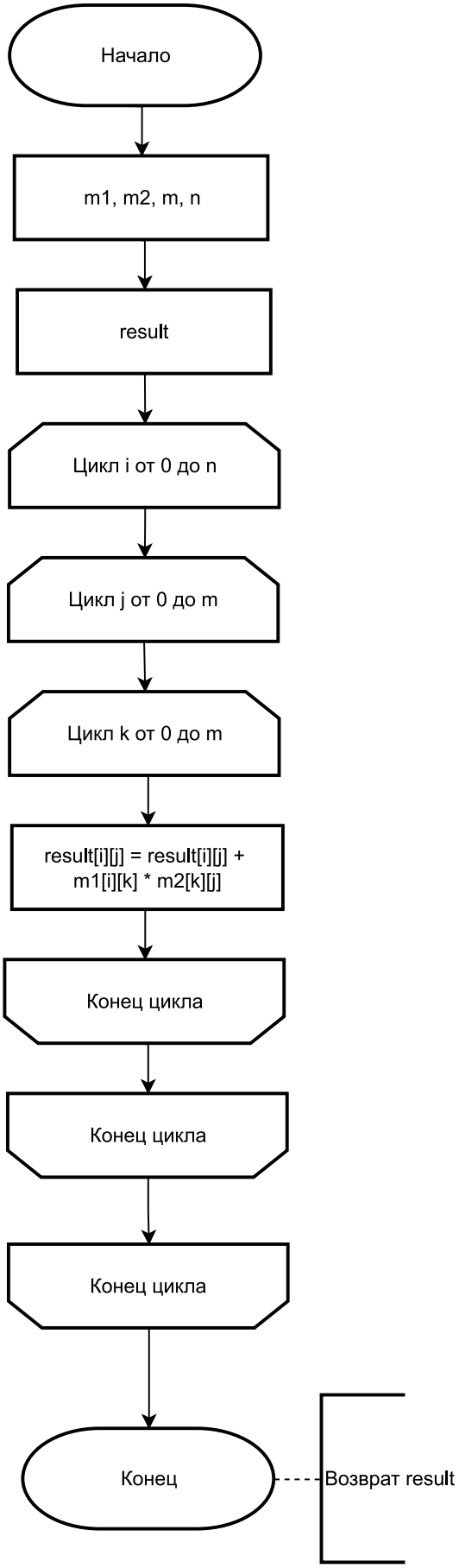

Рис. 2.1 – Схема стандартного алгоритма

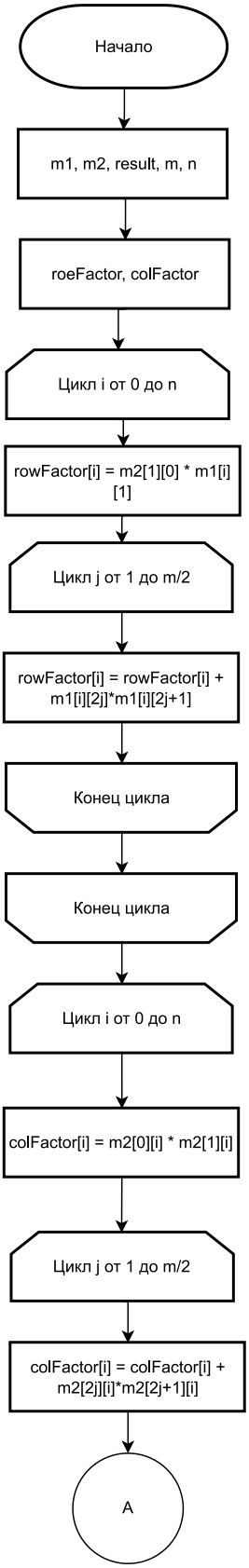

Рис. 2.2 – Схема алгоритма Винограда, часть 1

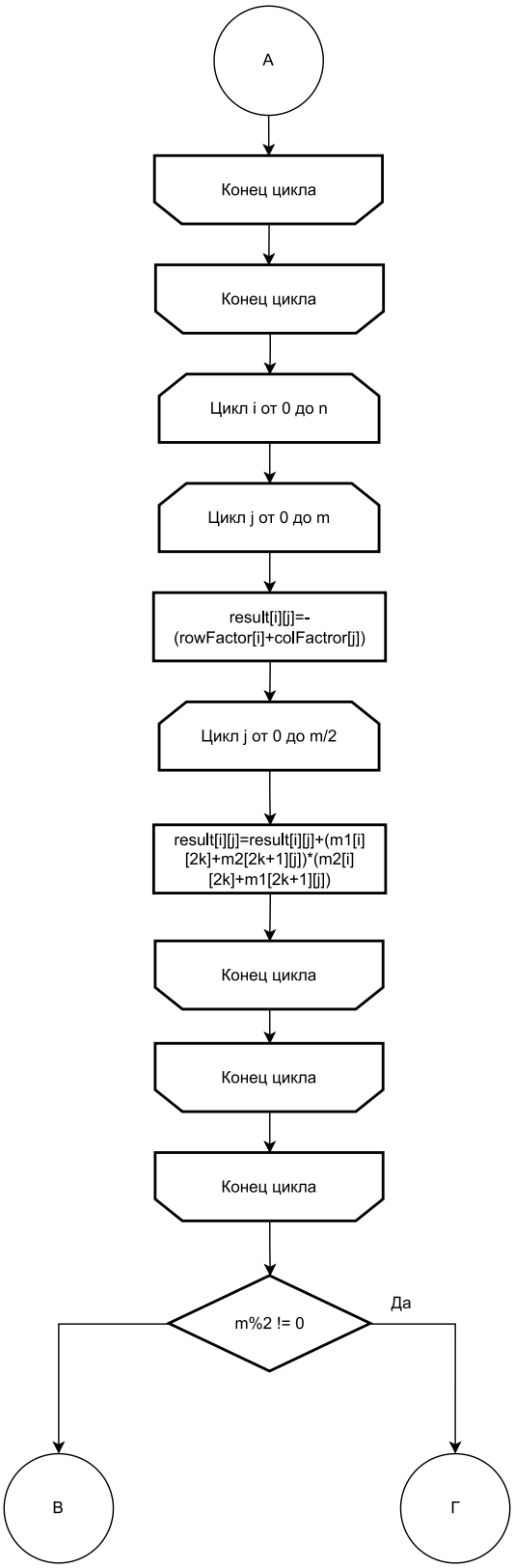

Рис. 2.3 – Схема алгоритма Винограда, часть 2

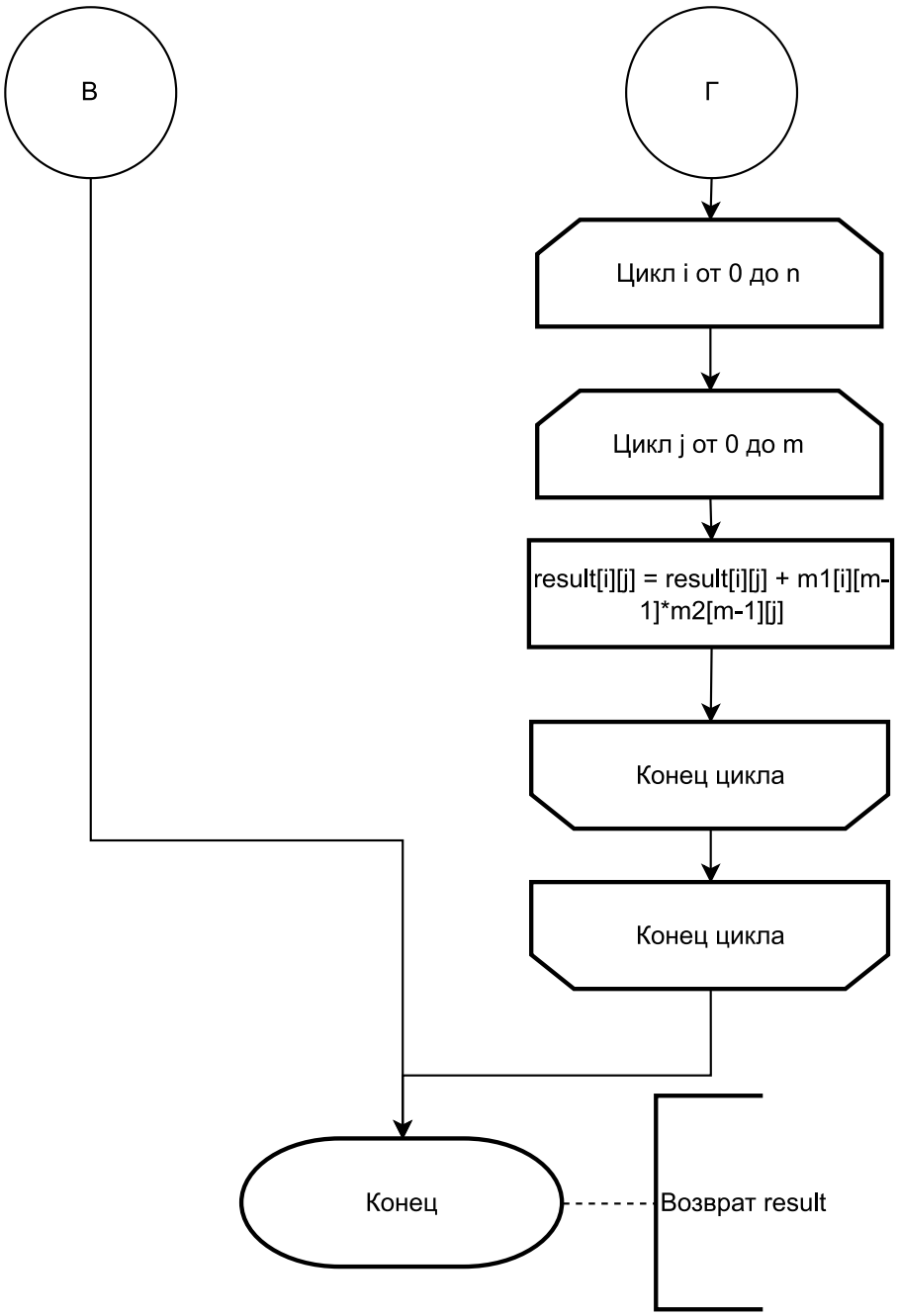

Рис. 2.4 – Схема алгоритма Винограда, часть 3

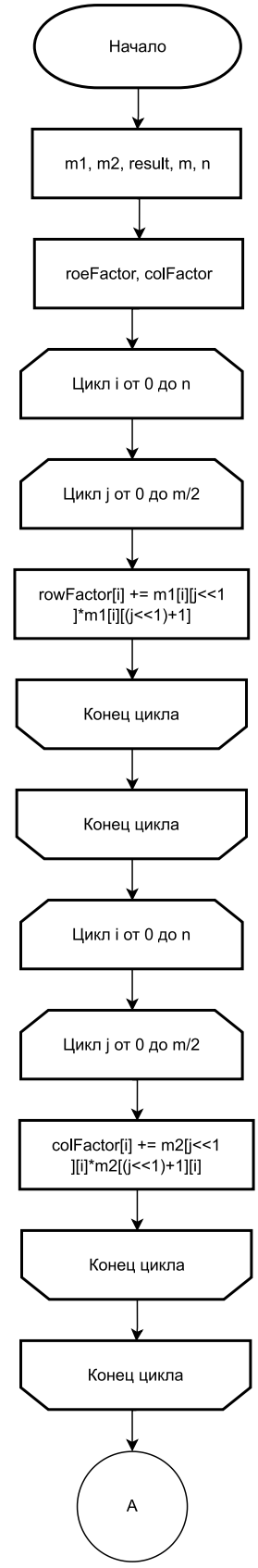

Рис. 2.5 – Схема оптимизированного алгоритма Винограда, часть 1

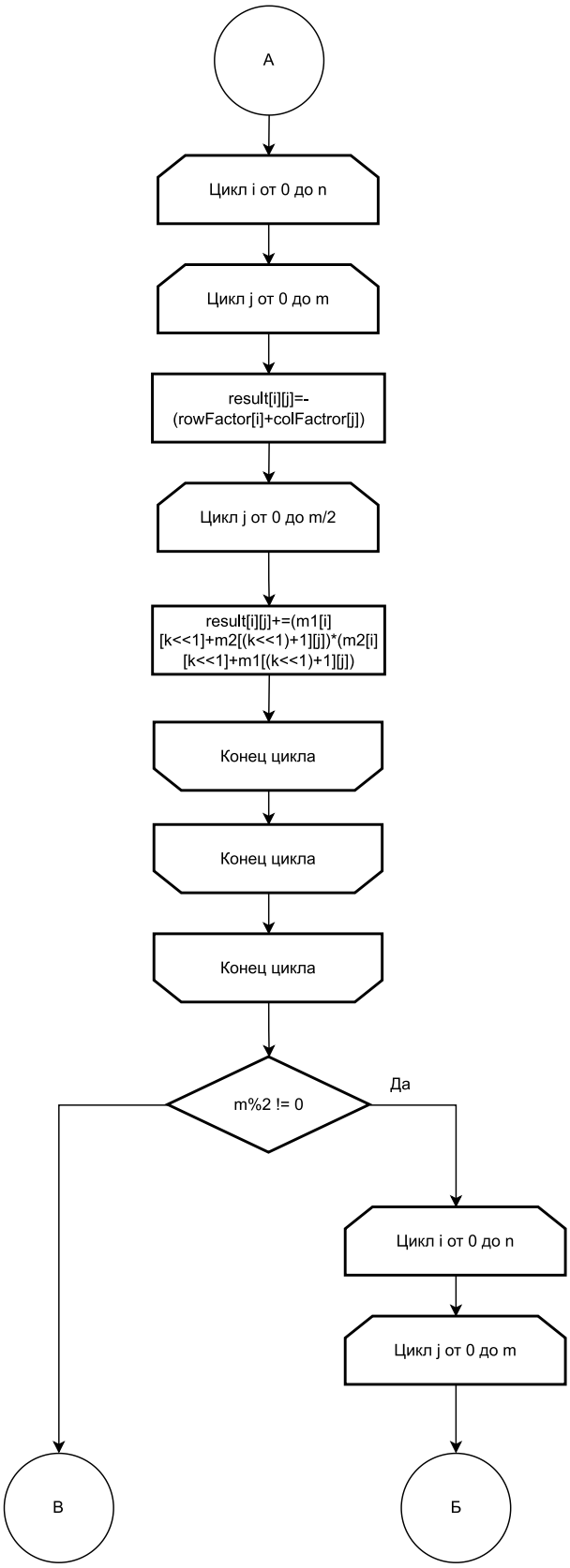

Рис. 2.6 – Схема оптимизированного алгоритма Винограда, часть 2

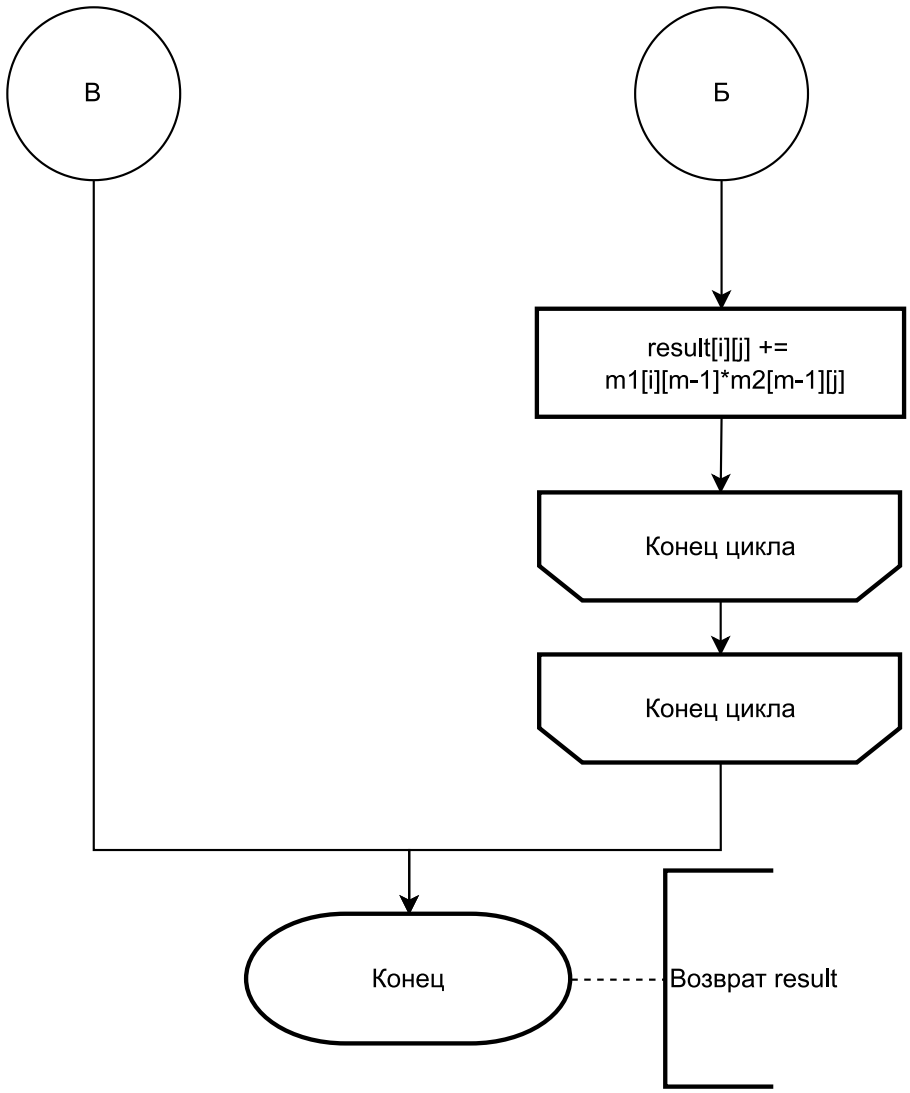

Рис. 2.7 – Схема оптимизированного алгоритма Винограда, часть 3

2.2 Трудоемкость алгоритмов

Введем модель вычисления трудоемкости.

1. Операции из списка (2.1) имеют трудоёмкость, равную 1:

$$+, -, /, *, \%, =, + =, - =, * =, / =, \% =, ==$$

$$! =, <, >, <=, >=, [], ++, --$$ (2.1)

2. Трудоёмкость оператора выбора if условие then A else B рассчитывается как

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

3. Трудоёмкость цикла из NN шагов рассчитывается как

$$f_{for} = f_{\text{инициализации}} + f_{\text{сравнения}} + NN(f_{\text{тела}} + f_{\text{инкремент}} + f_{\text{сравнения}})$$ (2.3)

4. Трудоёмкость вызова функции равна 0.

Далее будет приведены оценки трудоемкости алгоритмов.

2.2.1 Стандартный алгоритм умножения матриц

Для стандартного алгоритма умножения матриц трудоемкость будет слагаться из трех пунктов:

- внешнего цикла по $i \in [1..M]$, трудоёмкость которого $f = 2 + M \cdot (2 + f_{body_j})$;
- цикла по $j \in [1..N]$, трудоёмкость которого $f = 2 + N \cdot (2 + 2 + f_{body_k})$;
- цикла по $k \in [1..K]$, трудоёмкость которого $f = 2 + 13K$.

Поскольку трудоемкость стандартного алгоритма равна трудоемкости внешнего цикла, то

$$f_{standard} = 2 + M \cdot (4 + N \cdot (4 + 13K)) = 2 + 4M + 4MN + 13MNK \quad (2.4)$$

$$\approx 13MNK \quad (2.5)$$

2.2.2 Алгоритм Винограда

Чтобы вычислить трудоемкость алгоритма Винограда, нужно учесть следующие пункты.

- Создание и инициализация массивов a_{tmp} и b_{tmp} трудоёмкостью

$$f_{init} = L + N \quad (2.6)$$

- Заполнение массива a_{tmp}, трудоёмкость:

$$f_{a_{tmp}} = 2 + L(4 + 7 + (\frac{M}{2} - 1) \cdot 12) \quad (2.7)$$

- Заполнение массива b_{tmp}, трудоёмкость:

$$f_{b_{tmp}} = 2 + N(4 + 7 + (\frac{M}{2} - 1) \cdot 12) \quad (2.8)$$

- Цикл заполнения для чётных размеров, трудоёмкость которого

$$f_{cycle} = 2 + L \cdot (4 + N \cdot (11 + \frac{M}{2} \cdot 23)) \quad (2.9)$$

- Цикл, который необходим для рассчёта при нечётном размере матрицы, трудоемкость которого

$$f_{last} = \begin{cases} 2, & \text{чётная,} \\ 4 + L \cdot (4 + 14N), & \text{иначе.} \end{cases} \tag{2.10}$$

Тогда для худшего случая (нечётный общий размер матриц) имеем трудоемкость:

$$f_{worst} = f_{a_{tmp}} + f_{b_{tmp}} + f_{cycle} + f_{last} \approx 11.5 \cdot LNM \tag{2.11}$$

Для лучшего случая (чётный общий размер матриц) —

$$f_{best} = f_{a_{tmp}} + f_{a_{tmp}} + f_{cycle} + f_{last} \approx 11.5 \cdot LNM \tag{2.12}$$

2.2.3 Оптимизированный алгоритм Винограда

Оптимизация заключается в следующих пунктах.

- Использование побитового сдвига вместо умножения на 2.
- Замена операции сложения и вычитания на операции $+=$ и $-=$ соответственно.
- Занесение в циклах вычисления множителей вычисления первых двух элементов во внутренний цикл j.

Тогда трудоемкость оптимизированного алгоритма Винограда состоит из нижележащих пунктов.

1. Создание и инициализацие массивов a_{tmp} и b_{tmp} (2.6).
2. Заполнение массива a_{tmp}, трудоёмкость:

$$f_{a_{tmp}} = 2 + L(4 + \frac{M}{2} \cdot 12) \tag{2.13}$$

3. Заполнение массива b_{tmp}, трудоёмкость:

$$f_{b_{tmp}} = 2 + N(4 + \frac{M}{2} \cdot 12) \tag{2.14}$$

4. Цикла заполнения для чётных размеров, трудоёмкость которого (2.3).

5. Условие, которое нужно для дополнительных вычислений при нечётном размере матрицы, трудоемкость которого (2.10).

Тогда для худшего случая — при нечётной размерности матриц M — трудоемкость равна

$$f_{worst} = f_{MH} + f_{MV} + f_{cycle} + f_{last} \approx 11.5 \cdot LNM \tag{2.15}$$

Для лучшего случая — при чётной размерности матриц M — трудоемкость равна

$$f_{best} = f_{MH} + f_{MV} + f_{cycle} + f_{last} \approx 11.5 \cdot LNM \tag{2.16}$$

Вывод

В данном разделе были описаны и проанализированы три алгоритма умножения матриц. Рассчитаны трудоёмкости алгоритмов в лучшем случае (л.с.) и в худшем случае (х.с.).

Стандартный алгоритм: трудоемкость $O(13n^3)$.

Алгоритм Винограда: л.с. – $O(11.5n^3)$, х.с. – $O(11.5n^3)$.

Оптимизированный алгоритм Винограда: л.с. – $O(11.5n^3)$, х.с. – $O(11.5n^3)$.

3. Технологическая часть

В данном разделе будут рассмотрены средства реализации, а также представлены листинги сортировок.

3.1 Средства реализации

В данной работе для реализации был выбран язык программирования $c++$. В текущей лабораторной работе требуется замерить процессорное время для выполняемой программы. Для визуализации результатов использовался язык $Python$.

Время работы реализаций алгоритмов было замерено с помощью функции *GetProcessTimes(...)* [2] из библиотеки $Windows.h$. Функция возвращает пользовательское процессорное временя типа float. Использовать функцию приходится дважды, затем из конечного времени нужно вычесть начальное, чтобы получить результат.

3.2 Сведения о модулях программы

Программа состоит из следующих модулей:

- $AA2.cpp$ — файл, содержащий весь служебный код;
- $MatrixMull.cpp$ — файл, содержащий код всех алгоритмов матричного умножения;
- $TimeCompare.cpp$ — файл, производящий замеры времени;
- $GetCPUTime.cpp$ — файл, определяющий функцию замера времени;
- $Gen.cpp$ — файл, генерирующий входную матрицу;

- *MatrixMull.hpp* — заголовочный файл модуля *MatrixMull.cpp*;
- *TimeCompare.hpp* — заголовочный файл модуля *TimeCompare.cpp*;
- *GetCPUTime.hpp* — заголовочный файл модуля *GetCPUTime.cpp*;
- *Gen.hpp* — заголовочный файл модуля *Gen.cpp*.

3.3 Реализация алгоритмов

В листингах 3.1, 3.2, 3.3, представлены реализации алгоритмов матричного умножения — стандартного, Винограда, оптимизированного алгоритма Винограда.

Листинг 3.1 — Стандартный алгоритм умножения матриц

```
data_t matrix_multiplication(const data_t m1, const data_t m2)
{
  data_t result;
  result.n = m1.n;
  result.m = m2.m;
  result.matrix = create_matrix(m1.n, m2.m);
  if (!result.matrix)
  return result;
  for (int i = 0; i < m1.n; i++)
  for (int j = 0; j < m2.m; j++)
  {
    result.matrix[i][j] = 0;
    for (int k = 0; k < m1.m; k++)
    result.matrix[i][j] += m1.matrix[i][k] * m2.matrix[k][j];
  }
  return result;
}
```

Листинг 3.2 — Алгоритм Винограда умножения матриц

```
data_t matrix_multiplication_Vinograd(const data_t m1, const data_t m2)
{
  data_t result;
  result.n = m1.n;
  result.m = m2.m;
  result.matrix = create_matrix(m1.n, m2.m);
  if (!result.matrix)
  return result;
  int n = m1.n;
  int m = n;

  int *rowFactor = (int *)calloc(n, sizeof(int)), *colFactor = (int*)
     calloc(n, sizeof(int));
    for (int i = 0; i < n; ++i)
  {
    rowFactor[i] = m1.matrix[i][0] * m1.matrix[i][1];
    for (int j = 1; j < d; ++j)
    rowFactor[i] +=  m1.matrix[i][j<<1] * m1.matrix[i][(j<<1) + 1];
  }

  for (int i = 0; i < n; ++i)
  {
    colFactor[i] = m2.matrix[0][i] * m2.matrix[1][i];
    for (int j = 1; j < d; ++j)
    colFactor[i] += m2.matrix[j<<1][i] * m2.matrix[(j<<1) + 1][i];
  };

  for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j)
  {
    result.matrix[i][j] = -(rowFactor[i] + colFactor[j]);
    for (int k = 0; k < d; ++k)
    {
      result.matrix[i][j] += (m1.matrix[i][k << 1] + m2.matrix[(k<<1) +
          1][j]) * (m1.matrix[i][(k<<1) + 1] + m2.matrix[ k<<1][j]);
    }
  }

```

```
37    if (m % 2 != 0)
38    {
39      for (int i = 0; i < n; ++i)
40      for (int j = 0; j < n; ++j)
41      {
42        result.matrix[i][j] += m1.matrix[i][m - 1] * m2.matrix[m - 1][j];
43      }
44    }
45
46    free(rowFactor);
47    free(colFactor);
48    return result;
49  }
```

Листинг 3.3 — Оптимизированный алгоритм Винограда умножения матриц

```
1  data_t matrix_multiplication_VinogradOptimase(const data_t m1, const
       data_t m2)
2  {
3    data_t result;
4    result.n = m1.n;
5    result.m = m2.m;
6    result.matrix = create_matrix(m1.n, m2.m);
7    if (!result.matrix)
8    return result;
9    int n = m1.n;
10   int m = n;
11   int d = n>>1;
12   int *rowFactor = (int *)calloc(n, sizeof(int)), *colFactor = (int*)
       calloc(n, sizeof(int));
13     for (int i = 0; i < n; ++i)
14   {
15     for (int j = 0; j < m / 2; ++j)
16     rowFactor[i] = rowFactor[i] + m1.matrix[i][2 * j] * m1.matrix[i][2
         * j + 1];
17   }
18
19   for (int i = 0; i < n; ++i)
20   {
```

```
21      for (int j = 0; j < m / 2; ++j)
22      colFactor[i] = colFactor[i] + m2.matrix[2 * j][i] * m2.matrix[2 * j
            + 1][i];
23    };
24
25    for (int i = 0; i < n; ++i)
26    for (int j = 0; j < n; ++j)
27    {
28      result.matrix[i][j] = -(rowFactor[i] + colFactor[j]);
29      for (int k = 0; k < m / 2; ++k)
30      {
31        result.matrix[i][j] = result.matrix[i][j] + (m1.matrix[i][2 * k]
              + m2.matrix[2 * k + 1][j]) * (m1.matrix[i][2 * k + 1] + m2.
              matrix[2 * k][j]);
32      }
33    }
34
35    if (m % 2 != 0)
36    {
37      for (int i = 0; i < n; ++i)
38      for (int j = 0; j < n; ++j)
39      {
40        result.matrix[i][j] = result.matrix[i][j] + m1.matrix[i][m - 1] *
              m2.matrix[m - 1][j];
41      }
42    }
43    free(rowFactor);
44    free(colFactor);
45    return result;
46  }
```

3.4 Функциональное тестирование

В таблице 3.1 приведены тесты для функций, реализующих алгоритмы матричного умножения. Тесты для всех реализаций алгоритмов пройдены успешно.

Таблица 3.1 — Функциональные тесты

Матрица 1	Матрица 2	Ожидаемый результат
$\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
$\begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{pmatrix}$
$\begin{pmatrix} 1 \end{pmatrix}$	$\begin{pmatrix} 2 \end{pmatrix}$	$\begin{pmatrix} 2 \end{pmatrix}$
$\begin{pmatrix} 1 \end{pmatrix}$	$\begin{pmatrix} \end{pmatrix}$	Сообщение об ошибке
$\begin{pmatrix} 1 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$	Сообщение об ошибке

Вывод

В данном разделе были представлены реализации следующих алгоритмов: стандартного матричного умножения, Винограда, Винограда с оптимизацией. Выполнено тестирование реализаций алгоритмов.

4. Исследовательская часть

В данном разделе будут приведены примеры работы программы, а также проведен сравнительный анализ процессорного времени работы реализаций алгоритмов при различных ситуациях на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры времени представлены далее:

- операционная система Windows 11 Pro Версия 22H2 (22621.674) [3];
- память 16 ГБ;
- процессор 11th Gen Intel(R) Core(TM) i5-11400 2.59 ГГц [4].

При тестировании компьютер был включен в сеть электропитания. Во время замеров процессорного времени устройство было нагружено только встроенными приложениями окружения, а также системой тестирования.

4.2 Демонстрация работы программы

На рисунке 4.1 представлен результат работы программы. На экран выводятся результаты заемров времени для разных размеров матриц и разных видов алгоритмов матричного умножения в мс.

	Time in ms		
Size	Easy	Vinograde	VinogradeOptimise
10	0	0.015625	0
20	0.03125	0.03125	0.015625
30	0.0625	0.03125	0.03125
40	0.046875	0.0625	0.0625
50	0.34375	0.28125	0.296875
60	0.671875	0.4375	0.4375
70	1.04688	0.65625	0.78125
80	1.57812	1.15625	1.14062
90	2.34375	1.6875	1.625
100	3.07812	2.23438	2.23438
200	25	18.0781	18.1406
300	85.7031	63.2656	61.4531
400	206.312	150.75	150.516
500	411.062	301.531	302.391

Рис. 4.1 – Пример работы программы

4.3 Время выполнения реализаций алгоритмов

Как было сказано выше, используется функция замера процессорного времени GetProcessTimes(...) из библиотеки Windows.h.

Входные данные: линейная размерность квадратной матрицы от 10 до 500 для л.с., от 11 до 501 для х.с., элементы матрицы – целые числа от 0 до 200.

Результаты замеров времени работы реализаций алгоритмов матричного умножения на различных входных данных (в мс) приведены в таблицах 4.1, 4.2.

Таблица 4.1 — Процессорное время работы реализаций алгоритмов для четной размерности M

Размер	Классический	Винограда	Оптимизированный
10	0	0.015625	0
20	0.03125	0.03125	0.015625
30	0.0625	0.03125	0.03125
40	0.046875	0.0625	0.0625
50	0.34375	0.28125	0.296875
60	0.671875	0.4375	0.4375
70	1.04688	0.65625	0.78125
80	1.57812	1.15625	1.14062
90	2.34375	1.6875	1.625
100	3.07812	2.23438	2.23438
200	25	18.0781	18.1406
300	85.7031	63.2656	61.4531
400	206.312	150.75	150.516
500	411.062	301.531	302.391

Таблица 4.2 — Процессорное время работы реализаций алгоритмов для нечетной размерности M

Размер	Классический	Оптимизированный Винограда	Винограда
11	0.015625	0.015625	0.015625
21	0.0625	0.046875	0.046875
31	0.15625	0.109375	0.125
41	0.34375	0.265625	0.296875
51	0.65625	0.484375	0.515625
61	1.04688	0.8125	0.90625
71	1.76562	1.3125	1.32812
81	2.85938	1.92188	1.95312
91	3.54688	2.6875	2.70312
101	4.71875	3.57812	3.67188
201	37.7031	29.2188	29.6719
301	145.453	108.859	111.656
401	358.453	282.562	350.109
501	810.516	590.203	636.422

Также на рисунках 4.2 и 4.3 приведены графические результаты замеров времени работы алгоритмов в зависимости от линейного размера входной матрицы M.

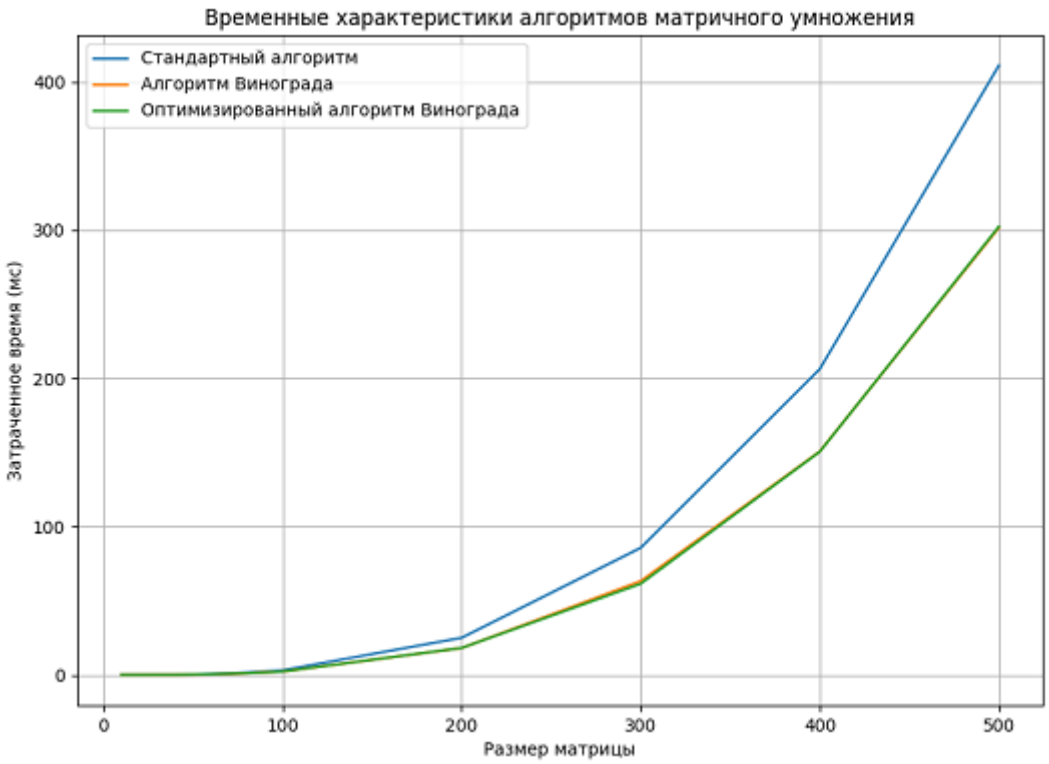

Рис. 4.2 – Процессорное время вычислений: четная размерность

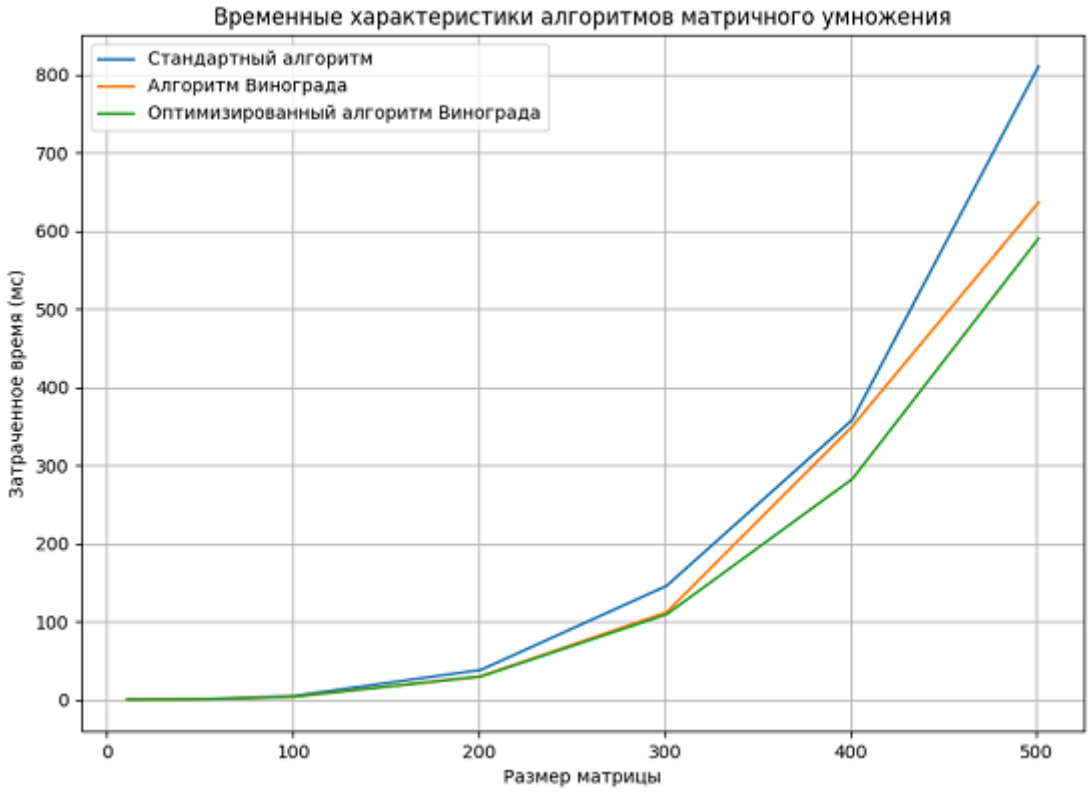

Рис. 4.3 – Процессорное время вычислений: нечетная размерность

4.4 Вывод

Теоретические результаты оценки трудоемкости и полученные практически результаты замеров процессорного времени совпадают. Алгоритмы Винограда выполняются быстрее, чем стандартный алгорим умножения матриц, примерно в 1.3 раза. Различия между временами выполнения реализаций алгоритмов Винограда и стандартного алгоритма становятся более различимы при наступлении худшего случая по трудоёмкости для алгоритма Винограда — при нечётном N.

Заключение

Цель, которая была поставлена в начале лабораторной работы, была достигнута: выполнен анализ трудоемкости реализаций алгоритмов матричного умножения.

В ходе выполнения лабораторной работы были решены все задачи:

1) изучены алгоритмы умножения матриц — классический, Винограда, оптимизированный Винограда;
2) проведен сравнительный анализ трудоёмкости алгоритмов на основе теоретических расчетов;
3) реализован каждый из трех алгоритмов;
4) проведены замеры процессорного времени для каждой из реализаций алгоритмов;
5) выполнен анализ полученных результатов;
6) по итогам работы составлен отчет.

В ходе проделанной работы было выявлено, что различия в процессорном времени выполнения двух версий алгоритма Винограда не являются существенными и не превосходят разрыва в 0.9 раз. В это же время стандартный алгоритм заметно отстает во времени выполнения при увеличении размеров матрицы. Из полученных данных видно, что время работы реализаций алгоритмов возрастает кубически с увеличением линейной размерности матриц.

Список использованных источников

1. Coppersmith D., Winograd S. Matrix multiplication via arithmetic progressions / Symbolic Computation, 1990, no. 3, vol. 9. — City: Publisher, 1990. — Pp. 251–280.
2. GetProcessTimes function [Эл. ресурс]. Режим доступа: https://clck.ru/32NCYi (дата обращения: 13.10.2022).
3. Windows 11, version 22H2 [Эл. ресурс]. Режим доступа: https://clck.ru/32NCXx (дата обращения: 14.10.2022).
4. Процессор Intel® Core™ i7 [Эл. ресурс]. Режим доступа: https://clck.ru/yeQa8 (дата обращения: 14.10.2022).